This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

Context-based Access Control Systems for Mobile Devices

Bilal Shebaro, Oyindamola Oluwatimi, Elisa Bertino
Computer Science, Cyber Center and CERIAS,
Purdue University, West Lafayette, IN 47907, USA
Email: {bshebaro, ooluwati, bertino}@purdue.edu

Abstract—Mobile Android applications often have access to sensitive data and resources on the user device. Misuse of this data by malicious applications may result in privacy breaches and sensitive data leakage. An example would be a malicious application surreptitiously recording a confidential business conversation. The problem arises from the fact that Android users do not have control over the application capabilities once the applications have been granted the requested privileges upon installation. In many cases, however, whether an application may get a privilege depends on the specific user context and thus we need a context-based access control mechanism by which privileges can be dynamically granted or revoked to applications based on the specific context of the user. In this paper we propose such an access control mechanism. Our implementation of context differentiates between closely located sub-areas within the same location. We have modified the Android operating system so that context-based access control restrictions can be specified and enforced. We have performed several experiments to assess the efficiency of our access control mechanism and the accuracy of context detection.

Index Terms—Context-based access control, smartphone devices, security and privacy, policies, mobile applications

1 Introduction

As smartphones are becoming more powerful in terms of computational and communication capabilities, application developers are taking advantage of these capabilities in order to provide new or enhanced services to their applications. For example, on March 2013 Samsung unveiled its Galaxy $S4$ device with 8 CPU cores and 9 sensors that enrich the device with powerful resources [1]. However, the majority of these resources can collect sensitive data and may expose users to high security and privacy risks if applications use them inappropriately and without the user's knowledge [2]. The threat arises when a device application acts maliciously and uses device resources to spy on the user or leak the user's personal data without the user's consent [3]–[5]. Moreover, users carrying their smartphones in public and private places may unknowingly expose their private information and threaten their personal security as they are not aware of the existence of such malicious activities on their devices. To prevent such threats, users must be able to have a better control over their device capabilities by reducing certain application privileges while being in sensitive contexts e.g. confidential meetings. To achieve this, smartphone systems must provide device owners with configurable policies that enable users to control their device usage of system resources and application privileges according to context, mainly location and time. Since such a feature is still missing in popular smartphone systems, such as in Android systems, it is crucial to investigate approaches for providing such control to device users.

The need for configurable device policies based on context extends from high profile employees to regular smartphone users. For example, government employers, such as in national labs [6], restrict their employees from bringing any camera-enabled device to the workplace, including smartphones, even though employees might need to have their devices with them at all times as their devices may contain data and services they might need at any time. With context-based device policies, employees may be allowed to use smartphones as they can disable all applications from using the camera and any device resources and privileges that employers restrict while at work, while the user device can retain all its original privileges outside the work area. Context-based policies are also a necessity for politicians and law enforcement agents who would need to disable camera, microphone, and location services from their devices during confidential meetings while retaining these resources back in non-confidential locations. With context-based policies, users can specify when and where their applications can access their device data and resources, which reduces the hackers' chances of stealing such data.

Previous work on security for mobile operating systems focuses on restricting applications from accessing sensitive data and resources, but mostly lacks efficient techniques for enforcing those restrictions according to fine-grained contexts that differentiate between closely located subareas [7]. Moreover, most of this work has focused on developing policy systems that do not restrict privileges per application and are only effective system-wide [8]. Also, existing policy systems do not cover all the possible ways in which applications can access user data and device resources. Finally, existing location-based policy systems are not accurate enough to differentiate between nearby locations without extra hardware or location devices [7], [9], [10]. In most

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

cases, such systems assume the context as given without providing or evaluating context detection methods of mobile devices [7], [11].

The design of context based policy systems for smartphones is challenging as it should fulfill the following requirements:

1) Applications should not be able to fake the location or time of the device, as they should not be able to bypass the policy restrictions applied on the device in a specific context.
2) As users are assumed to be mobile, the policy restrictions should be applied automatically on the device as the device's location changes.
3) The accuracy of location needs to be higher than the location accuracy by GPS, as we need to apply different policies in different spots or nearby sub-areas located within the same GPS location.
4) The enforcement of context-based policies should not require the application developers to modify source code, or impose any additional requirement on their applications.
5) The applied policy should not cause significant delays in the device functionality that could negatively impact the system performance.

In this paper, we propose a context-based access control (CBAC) mechanism for Android systems that allows smartphone users to set configuration policies over their applications' usage of device resources and services at different contexts. Through the CBAC mechanism, users can, for example, set restricted privileges for device applications when using the device at work, and device applications may re-gain their original privileges when the device is used at home. This change in device privileges is automatically applied as soon as the user device matches a pre-defined context of a user-defined policy. The user can also specify a default set of policies to be applied when the user is located in a non-previously defined location.

Configured policy restrictions are defined according to the accessible device resources, services, and permissions that are granted to applications at installation time. Such policies define which services are offered by the device and limit the device and user information accessibility. Policy restrictions are linked to context and are configured by the device user. We define context according to location and time. Location is determined basically through visible Wi-Fi access points and their respective signal strength values that allows us to differentiate between nearby sub-areas within the same work space, in addition to GPS and cellular triangulation coordinates whenever available. We implement our CBAC policies on the Android operating system and include a tool that allows users to define physical places such as home or work using the captured Wi-Fi parameters. Users can even be more precise by differentiating between sub-areas within the same location, such as living rooms and bedrooms at home or meeting rooms and offices at work. Once the user configures the device policies that define device and application privileges according to context, the policies will be automatically applied whenever the user is within a pre-defined physical location and time interval.

We deployed our CBAC policies on the latest Android Google Nexus 7 tablet and the Google Nexus 4 phone to compare the effectiveness of enforcing these policies in different contexts. Our investigation involved 250 applications and several test cases in various locations on our campus grounds with several Wi-Fi access points installed. Our experimental results show the effect of CBAC policies on Android applications and the policies impact on the system performance.

Organization. This paper is organized as follows. Section 2 introduces basic concepts and background information. Section 3 discusses the design of our architecture and introduces our access control framework. Section 4 introduces our policy constructs and their classification followed by implementation and technical details in Section 5. Section 6 emphasizes our technique in managing context information and how we keep policy restrictions up-to-date with device location. Section 7 reports results of experiments to assess the accuracy of context information and the impact of policy restrictions on applications. We analyze the security of our approach in Section 8. Sections 9 and 10 discuss related work and outline future work, respectively.

2 Background

In this section, we cover related background information on Android operating system (OS) and its access control mechanism, and some basics on location services.

2.1 Android

2.1.1 Operating System and API

The Android operating systems are derived from Linux based kernels and have enhanced support for security and privacy [12]. Android is designed with a multi-layered security infrastructure, which provides developers a secure architecture to design their applications.

Android applications are sandboxed and run inside the Dalvik Virtual Machine, which isolates each application's processes and data. Each application is assigned a user ID (UID) with assigned privileges, and is given a dedicated part of the file system for its own data. The processes of applications with different UIDs run separately and are isolated from each other and from the system. While applications have limited access to device resources, developers can explicitly request access for their applications through the Android permission system.

2.1.2 Permission System

The Android permission system controls which application has the privilege of accessing certain device resources and data. Application developers that need access to protected Android APIs need to specify the permissions they need in the *AndroidManifest.xml* file which, if inaccurately

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

assigned, can increase the risks of exposing the users' data and increase the impact of a bug or vulnerability.

Each application declares the permissions listed in its *AndroidManifest.xml* file at the time of installation, and users have to either grant all the requested permissions to proceed with the installation, or cancel the installation. The Android permission system does not allow users to grant or deny only some of the requested permissions, which limits the user's control of application's accessibility.

2.1.3 Android Application Components

Every Android application is composed of four essential components: Activities, Services, Content Providers, and Broadcast Receivers. An *Activity* defines an application's user interface. The *Service* component is designed to be used for background processing. The *Content Provider* component acts as a global instance for a particular application, so that all applications on the device can use it. It stores and manages a shared set of data belonging to the owner application using a relational database interface. Finally the *Broadcast Receiver* component acts as mailboxes that allows applications to register for system or application events. All registered receivers for an event will be notified by Android once this event happens.

2.1.4 Intents

Intents are asynchronous messages that allow the three core application components *Activities, Services*, and *Broadcast Receivers* to communicate and request functionality from each other. Communication between components occurs within the application own processes or across external applications. Intents are used for many purposes such as data transfer, service requests, and application action requests.

2.2 Location Positioning Methods

In our paper, we rely on Wi-Fi-based positioning techniques to retrieve the location of the device. In addition to these techniques, we also collect location data retrieved from GPS and cellular networks for situations where there is no Wi-Fi coverage in the areas of interest. Moreover, we use public GPS location data in defining policy contexts for areas that have not been previously visited, as described in Section 6. In this section, we introduce a brief description on methods used in smartphones for locating the devices.

2.2.1 GPS Trilateration

The Global Position System (GPS) is a positioning tool available in most smartphones which uses data signals from satellites to compute the position of the device. Data received from satellites contains the time stamp of sending, the orbital information, and the position of satellites. With at least three different satellite signals, the GPS uses the trilateration method to calculate the device's location by measuring the satellite signals time difference or their received signal strengths. The location information provided from the GPS includes the latitude, longitude, altitude, and time. The accuracy of this method is estimated to be in the range of 50 to 100 meters.

2.2.2 Cellular Triangulation

Cellular triangulation (*cell ID*) is another positioning approach based on cellular technology. It refers to tracking a mobile phone's current location using radio towers. Through contacting every nearby antenna tower, cellular triangulation can make a measurement of how far away the cellular mobile device is based on the signal it is transmitting. This is done by measuring signal strength and the round-trip signal time. Once this distance is calculated, finer approximations can be done by interpolating transmitting signals between adjacent radio towers. The accuracy of this method varies according to the density of cell towers existing in an area. For instance, the accuracy in large cities can reach up to 10 meters due to the high density of cellular towers that a device could be in contact with [13].

2.2.3 Wi-Fi Positioning Method

Many smartphone devices nowadays rely on Wi-Fi-based positioning techniques due to its efficiency in computing the location of a device especially in places when GPS and cellular signals are weak or unavailable [14]. These techniques are based on comparing the device's received Wi-Fi access points (AP) with database fingerprints containing Wi-Fi access points with known geographic locations [15].

In our work, smartphone users will define their own database of Wi-Fi APs which only includes the user's areas of interest. In addition to the Wi-Fi APs, we also store the Received Signal Strength Indication (RSSI) of each AP to enhance the positing accuracy of the device, which in our case, needed to differentiate between nearby sub-areas. We describe in details how we capture and store these Wi-Fi parameters in Section 6, in addition to how we use them for detecting the device location.

3 Architecture Design

In this section, we introduce the design of our architecture through describing the components of our access control framework with the corresponding role of its entities.

Our framework consists of an access control mechanism that deals with access, collection, storage, processing, and usage of context information and device policies. To handle all the aforementioned functions, our framework design consists of four main components as shown in Figure 1.

The **Context Provider** (CP) collects the physical location parameters (GPS, Cell IDs, Wi-Fi parameters) through the device sensors and stores them in its own database, linking each physical location to a user-defined logical location. It also verifies and updates those parameters whenever the device is re-located.

The **Access Controller** (AC) controls the authorizations of applications and prevents unauthorized usage of device resources or services. Even though the Android OS has its own permission control system that checks if an application has privileges to request resources or services, the AC complements this system with more control methods

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

and specific fine-grained control permissions that better reflect the application capabilities and narrow down its accessibility to resources. The AC enhances the security of the device system since the existing Android system has some permissions that, once granted to applications, may give applications more accessibility than they need, which malicious code can take advantage of. For example, the permission READ_PHONE_STATE gives privileged applications a set of information such as the phone number, the IMEI/MEID identifier, subscriber identification, phone state (busy/available), SIM serial number, *etc.*

The **Policy Manager** (PM) represents the interface used to create policies, mainly assigning application restrictions to contexts. It mainly gives control to the user to configure which resources and services are accessible by applications at the given context provided by the CP. As an example, the user through the PM can create a policy to enable location services only when the user is at work during weekdays between 8 am and 5 pm.

The **Policy Executor** (PE) enforces device restrictions by comparing the device's context with the configured policies. Once an application requests access to a resource or service, the PE checks the user-configured restrictions set at the PM to either grant to deny access to the application request. The PE acts as a policy enforcement by sending the authorization information to the AC to handle application requests, and is also responsible to resolve policy conflicts and apply the most strict restrictions.

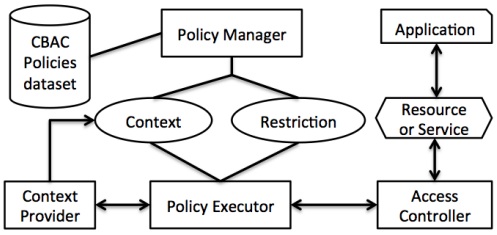

Fig. 1: **Access Control Framework.**

Through the PM, users can create CBAC policies through configuring application restrictions and linking them to contexts. When an application requests a resource or service, the AC verifies at run-time whether the application request is authorized and forwards the request to the PE. If the request is authorized, the PE then checks if there is any policy that corresponds to the application request. If such a policy exists, the PE requests from the CP to retrieve the context at the time of the application request. The PE then compares the retrieved context with the context defined in the policy. In case of a match, the PE enforces the corresponding policy restrictions by reporting back to the AC to apply those restrictions on the application request.

We carefully design the access control framework so that the user-configured policies are securely enforced with minimal processing steps and execution time to avoid any significant delays in responding back to the requesting application. As our design should securely handle policy execution, we maintain the context data provided by the CP to make sure it is accurate, precise and up-to-date.

4 Policy Core Model

In this section we describe the core policy constructs that compose our CBAC policies. We start by defining the policy constructs and then categorize our policies according to the type of restrictions and modifications that we need to apply to the Android OS.

4.1 Policy Constructs

We define our CBAC policies as a set of restrictions applied to the smartphone applications when the device is located within a specified context. Policy restrictions represent the constraints applied on the applications' privileges in accessing device resources, system methods, functions, user data, and services. Policy contexts represent to where and when the policy must be enforced.

In what follows, we assume three sets: (1) *SUB* the set of subjects representing the device applications, (2) *OBJ* the set of protected objects (objects, for short) representing the permissions, services, and functionalities available for the system or applications, and (3) *ACTION* the set of restriction actions that can be applied through the CBAC policies.

The set of subjects *SUB* is composed of the *PackageNames* of all applications installed on the device. In addition, a special character $*$ is added to the set to represent all installed applications. This character is useful for policies that need to be enforced on all applications, rather than creating the same policy for every application. Moreover, we assume that each object from set *OBJ* has an associated type from the set {Permission, Data, Intent, System_Peripheral}. Let o be an object from the set *OBJ*; notation *type(o)* denotes the type of o. The set of actions *ACTION* defined for our CBAC model includes the following actions:

revoke_Permission: denys permission(s) from being granted to application(s).

shadow_Data: conceals the actual user data stored on the device.

disable_Intent: intercepts and drops the specified intent message.

Save_State: disables toggling the state (ON/OFF) of the specified system peripheral.

Definition 1. (Restriction.) Let $s \in SUB$, $o \in OBJ$, and $a \in ACTION$. A policy restriction is defined as the tuple $[s, o, a]$ such that:

$$a = \begin{cases} revoke\_Permission & \text{if type(o) = Permission.} \\ shadow\_Data & \text{if type(o) = Data.} \\ disable\_Intent & \text{if type(o) = Intent.} \\ Save\_State & \text{if type(o) = System\_Peripheral.} \end{cases}$$

Access control policies are linked to a context that specifies where and when these policies should be enforced. In our model, the policy context is composed of a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

physical location and a time interval. A physical location corresponds to where a policy should be enforced. We retrieve location data from GPS, cellular network, and Wi-Fi device sensors to represent a physical location. Specific to Wi-Fi, we retrieve the Wi-Fi APs and their RSSI ranges to differentiate between closely located sub-areas within the same location found by the GPS and cellular triangulation. From the defined physical locations, users can assign logical location names (such as living room or work office) corresponding to the captured physical locations. Using these user-defined logical locations, users can re-assign them to multiple policies without the need to capture their location data for every policy. We discuss further details on how we accurately capture and detect physical locations in Section 6.

On the other hand, a policy time interval represents the specific time period to when a policy should be enforced. We represent the specific date and time in the format of YYYY-MM-DD-hh:mm:ss. Additionally, we introduce the *R* flag to define recurring events. The value of *R* is drawn from the set {*O,D,W,M,Y*} defining the event frequency: $O \rightarrow once$, $D \rightarrow daily$, $W \rightarrow weekly$, $M \rightarrow monthly$, and $Y \rightarrow yearly$. An event is recurred based on the value of *R* and the date/time set in the policy time interval. For example, to set an event that occurs every Monday from 5 pm to 10 pm, *R* is set to W and the time interval should be set to a sample event date-time, such as starting on $2013-04-01-17:00:00$ and ending on $2013-04-01-22:00:00$.

Definition 2. (Context.) Let LOC be the location data of a particular sub-area and $Name$ be a user-defined logical location that corresponds to this sub-area. Also let $\{ST, ET, R\}$ respectively be the starting time, ending time, and frequency to when a particular policy should be enforced. A policy context is defined as the tuple [LOC, {ST, ET, R}] such that

$$LOC = <Name>, \\ <latitude, longitude>, \\ <Wi-Fi-APs, RSSI_{min}, RSSI_{max}>$$

Definition 3. (Policy.) Let r be a restriction as defined in *Def. 1* and c be a context as defined in *Def. 2*. A policy is defined as a tuple $[r, c]$.

Below is an example of a policy that disables the Skype application from having the *Camera* permission monthly between 4.00 pm and 5.00 pm on the first of every month starting August 1, 2013 at our department meeting room $Room110$:

POLICY = [[com.skype.raider, android.permission.CAMERA, revoke_Permission], [$<Room110>, <44.127, -106.875>$, $<34:8B:43:C6:98:22, -52, -60>$, $<11:9A:C0:93:54:8E, -68, -74>$, {2013-08-01-16:00:00, 2013-08-01-17:00:00, M}]]

4.2 Policy Categories and Examples

We here classify location dependent run-time policies according to the type of restrictions and modifications that we need to apply on the Android OS. Table 1 displays examples of policy restrictions for each policy category.

Resource Restriction Policies: This category corresponds to the Android resource permissions, mainly restricting device applications from using certain resources such as *Camera, GPS, etc.* As Android applications are granted all their set of requested permissions upon installation, this category deals with policies that can manage these permissions and control the applications' access to system resources post-installation. Permission restrictions are either applied system-wide or per application, as it is also possible to revoke some or all of the previously granted permissions per application. These permissions are managed by intercepting an application's request to use a permission at run-time, and then determine if the permission should be granted or denied depending on the current context. We discuss technical details on how to achieve permission restrictions in Section 5.

In situations when a resource, such as camera, can be accessed by the application using more than one possible method, we automatically create the appropriate policies that can restrict all these methods, as users have no knowledge of what method is the application is using. For instance, the camera resource can be either requested directly through the Camera API (requires permission) or indirectly by calling a camera intent message (no permission needed). For this reason, we complement the camera-related policy with two intent-related policies, one for capturing images and another for capturing videos.

Data Access Policies: This category relates to user data stored on the device such as *Contacts, Calendar, Accounts, etc.* As device users might wish to disable their applications from accessing such data at specific contexts, we design data-specific CBAC policies that allow users to set restrictions on data access per application or system-wide. These data access policies are enforced through obfuscating the data request and returning empty lists of the request data. We discuss more details in Section 5.

Similar to the resource restriction policies, we create additional policies to cover all possible methods of accessing *Contacts*, mainly intent-related policies.

System Peripheral State Policies: This category relates to policies that restrict applications from modifying the state of the device peripherals, such as toggling the Bluetooth state. While some developers may embed functionality for changing the state of device peripherals within their applications, users can disable such functionality in specific contexts that will restrict the configured applications from overriding the user's choice of the state of these peripherals. Restrictions on system peripherals are implemented mainly through modifying the peripheral API method to intercept any application's request to enable/disable a particular system peripheral. Section 5 contains more details on how to achieve these restrictions.

Multitasking and Intercommunication Policies: This category corresponds to policies that restrict data

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

Policy Category	Example	Policy Restriction [s,o,a]
Resource Restriction Policies	Disabe Camera for Skype	[com.skype.raider, android.permission.CAMERA, revoke_Permission]
Data Access Policies	Shadow Contacts for Pandora	[com.pandora.android, CONTACTS, shadow_Data]
System Peripheral State Policies	Disbale Bluetooth toggling	[*, BLUETOOTH, Save_State]
Multitasking and Intercommunication Policies	Disable loading a browser activity	[*, Intent.ACTION_VIEW, Uri.parse, disable_Intent]
User Security Policies	Disable uninstalling applications	[*, android.intent.action.DELETE, disable_Intent]

TABLE 1: Policy categories and examples.

communication and messages between different device applications in which many system restrictions can be applied. For instance, users may choose to disable multitasking and enable only one application to run at a time. Moreover, users may choose to automatically close all running applications whenever the device location is changed. These system restrictions are achieved by intercepting system and application *Intent* messages as we discuss in Section 5.

User Security Policies: This category of policies relate to modifications of the security level of the device which allows users to specify security restrictions based on context. With these policies, users can specify the context in which to ask for a passcode for using the device. Users can also choose to block certain applications from loading and running at a specific context, while enabling them at different contexts. Finally, as some applications may have functionality for installing/uninstalling other applications and scripts, users can use these policies to disable this functionality to avoid installing malicious code in the device. Using these types of policies, a user may choose to disable untrusted applications from running while at home. As these applications may use device sensors to track its users, device owners can temporarily block these applications when located in private places, and enable them in other public locations.

5 IMPLEMENTATION

In this section, we introduce the technical details of our implementation which includes our modifications to the Android OS and the components of the *Policy Manager* custom application that acts as an intermediary between the OS and the user's desired policy configurations.

5.1 Policy Manager Components

The *Policy Manager* custom application consists of the four main Android application components: Activities, Broadcast Receivers, a Content Provider, and a Service.

Activities: The user interacts with the *Policy Manager* via activities, and through these activities, a user is able to define physical locations and subsequently configure a set of policies for these locations. The main constituents of these activities include *Application Events, Permission Access, Resource Access, System Preferences*, and *Time Restriction*.

BroadcastReceiver: We extended the Android's *BroadcastReceiver* class and created two custom classes, the *StartLocationServiceReceiver* and the *BootReceiver* classes. The *StartLocationServiceReceiver* is responsible for triggering our customized *LocationService* for retrieving device location information. The *BootReceiver*'s main task is to schedule when the *StartLocationServiceReceiver* should request the location service. Once the *BootReceiver* receives the BOOT_COMPLETED *Intent* from the system, it uses the Android's *AlarmManager* service to let the receiver schedule a pending *Intent* to be sent periodically to our *StartLocationServiceReceiver* in order to update the device location.

Service: The *LocationService* service is derived from the *IntentService* class that facilitates offloading work from the main application's thread, allowing tasks to be performed in the background on a separate thread if desired. *LocationService* determines if the device has moved to or still is in a previously registered area. Offloading the aggregation of location-based data in a separate thread reduces the performance impact of the execution of the *LocationService* on the *Policy Manager*. We use the *AlarmManager* to periodically activate the *LocationService* to ensure the device's location is always up-to-date. By default, the *LocationService* is activated once per minute, but we give the user the choice to configure how often the service is executed. The duration of the service depends on the number of snapshots of location parameters to be taken, which is currently configured to four per area.

Content Provider: The policies configured by the user are stored within the *Policy Manager* data directory. This data is private to our custom application and cannot be accessed by other applications or the system itself, as a result of Linux's kernel user ID access control mechanisms. *PolicyCP* is our custom content provider that acts as a secure intermediary between the policy database and all objects outside of the *Policy Manager*'s running process. We chose to use the SQLite database to store user-configured policies due to the support and ease of programming provided by the Android API's associated with storing and managing databases on Android devices.

5.2 Permission Management

In the Android system, all resources that require explicit access rights in the form of permissions are protected by the *ActivityManagerService* class via permission verification. When an application attempts to use any of these resources, the *ActivityManagerService*'s method called *checkComponentPermission* is invoked to verify if the calling application has the appropriate permission(s) to access the resource.

We apply our modifications to this particular method by simply intercepting the permission call before the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

system performs its standard permission verification process. Given the permission and the application name, the system subsequently calls our custom content provider's *revokeResourceAccess* to determine the next course of action. Depending on the user's policy configuration, the next course of action could either be returning the constant *PackageManager.DENIED* in the *checkComponentPermission* if the user has configured to block that permission from the requesting application, or letting the normal verification process take its course. We also give the ability to revoke any or all permissions system-wide via the *PolicyManager*'s interface.

5.3 Restrictions on User Data

Our implementation of data obfuscation complements much of the techniques used in [16] and [17], but instead under the domain of CBAC policy restrictions. We obfuscate user data from applications attempting to access it if the policy restriction applies to those applications. We modify the Android APIs that access the user data saved on the device.

Relational database systems are the common data management systems used to create, store, and manage user data. Accessing these data usually require calling the *ContentResolver's query()* method, and thus we modify it for our purposes. Instead of returning the expected *Cursor* object needed to point to the required data, a *NullCursor* object is substituted. A *NullCursor* object represents an empty dataset, such as an empty list of pictures as if pictures were not present or never stored on the device.

5.4 Managing System Peripheral State

We also give users the option to configure a policy to restrict access to peripherals (e.g. Bluetooth) when entering a particular location. Specifically, users can set up their devices to prevent applications from modifying a peripheral's current state (enabled/disabled). While it is possible to modify a peripheral's current state by using permission management, we modify the specific methods that enable/disables these peripherals in order to prevent applications from crashing that do not have code for handling exceptions resulting from revocation of permissions. As an example, for Bluetooth we modified the *BluetoothAdapter* class and for Wi-Fi we modified the *WifiManager* class so to assure that these modifications do not result in application crashes and to prevent applications from modifying peripherals current state. Whenever an application tries to modify the state of a system peripheral, our content provider *PolicyCP* checks the validity of the request and would refuse the request if the request tries to override a user-configured restriction.

5.5 Intent Management

Intent messages are one of the common forms for inter- and intra-communication between application components, sent via three methods: *startActivity(), sendBroadcast()*, and *startService()*. Preventing an application from sending intents is simply a matter of intercepting the intents when the aforementioned methods are called by applications. *Intent* interception provides the user the ability to prevent an application component from starting another activity, broadcasting any possible sensitive information, or executing a possibly suspicious background service. For example, without the need of declaring the Android permission "RECORD_AUDIO", an application can indirectly access the device's microphone recorder application by requesting the *Activity* class to send a record audio intent. Therefore, we modified the *Activity* class which hosts *startActivity()* and *startActivityForResult()*, and the *ContextWrapper* class which contains *sendBroadcast()* and *startService()*. We modified these methods to intercept the *Intents* and control the actions performed based on those *Intent* objects.

Restriction Category	Description
Application Install/Uninstall	Prevent an application from sending an intent to install or uninstall an application.
Application Multitasking	Prevent running multiple user-application simultaneously.
Services	Prevent applications from starting background services.
Broadcasts	Prevent applications from broadcasting Intents.
Application Launching	Prevent certain applications from running on the device.
Lock/Unlock Device	Preventing requesting pin code to unlock the device.

TABLE 2: Examples of policy restrictions that can be controlled via intercepting Intents.

We classify these *Intents* based on the contents and description of the intent objects. The user is given, via the *Policy Manager* interface, the ability to prevent a specific set of *Intents* from being sent. Table 2 lists few examples of these *Intent*-related restrictions.

Launching applications is achieved by intercepting those intents and preventing applications from being started either by users or by other applications. The first method is when users launch applications through the default Android *Launcher* application, which is the home screen of the device. The second method for starting another application is calling its activity within an already opened application. We extract the action and category from the *Intent* object, and verify if it is "android.intent.action.MAIN" and "android.intent.category.LAUNCHER", respectively. If those specific contents are present and if the simultaneous running of applications is restricted, we discard the intent preventing the framework to handle it.

Finally, through the *Intent* management, users can control when to request a pin code when unlocking the device. In our implementation, we modify the *KeyguardViewMediator* class in order to intercept the locking operation of the device, and thus controlling when a PIN is required.

To summarize, users will have all the options to specify applications restrictions associated with context data through the policy managers. In the policy manager, the permission management is used to configure application restrictions related to device resources (e.g. Camera). Restrictions on user data are used to shadow user data (usually saved in relational databases) and to return fake data to the application (e.g. Contacts). Managing system peripheral state is used to control applications actions in toggling the state of certain resources (e.g. enabling/disabling Bluetooth). Finally, intent management

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

is used to control the communication between applications and filter user and application actions on the system. Intent management is also used to configure restrictions on applications accessing resources, as in some cases, these applications are developed to access system resources indirectly through using an intent message rather than requesting a permission.

6 Context Management

The main source of location-related information for our access control system is the Wi-Fi APs and their corresponding signal strengths. Location information acquired from GPS and cellular towers is also aggregated to our context definition but may not be sufficient for indoor localization especially that they may become weak or unavailable inside buildings or areas within building structures [18], [19]. However, location information retrieved from Wi-Fi parameters could be more precise to differentiate between closely located sub-areas within the same GPS location [20], [21].

A spatial region is represented by combining GPS coordinates, cellular triangulation location data, and Wi-Fi APs and signal strengths. In Android, the GPS coordinates and cellular triangulation are obtained in a similar fashion by invoking the Android *LocationManager* service. Once the *LocationManager* is invoked, we request location updates by calling the *requestLocationUpdates* method that returns a *Location* object which contains latitude, longitude, timestamp, and other information.

Wi-Fi is handled differently than the previous two location methods. We obtain the Wi-Fi parameters by invoking the *WifiManager* service to retrieve the Wi-Fi access points scans. We register our *BroadcastReceiver mWifi_receiver* with an IntentFiler action to receive the broadcasted Wi-Fi scanned intent, and then request for and subsequently process the actual scanned access points data.

In our CBAC policy system, we provide users with a utility to define physical locations by either capturing snapshots of location data of the desired areas or by manually entering the area location coordinates. In the following sections, we show our design and implementation of the location capturing phase when users define and store physical locations, and the location detection phase when device detects its location and match it with a pre-defined policy context.

6.1 Location Capturing Phase

Figure 2 describes how location data is captured for each context defined by the user. Through the location scan interface, the user is able to capture several snapshots of location data in different sub-areas. For each sub-area, location data is accumulated from each snapshot; the GPS coordinates and the cellular triangulation, when applicable, import the latitude and longitude from the captured snapshots and only select those with the highest position accuracy. With respect to Wi-Fi, we noticed that the Wi-Fi access points signal strengths fluctuate even

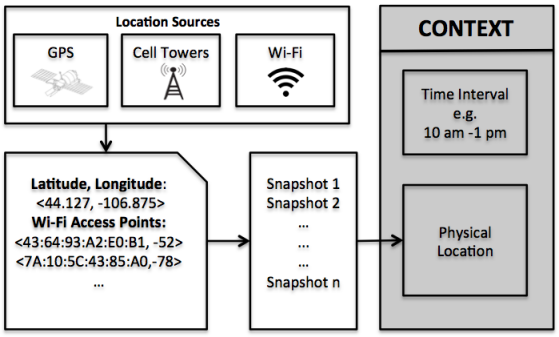

Fig. 2: **Location Capturing Phase.**

if the device is stationary or motionless. Therefore, our application scans the signal strengths of each access point for several seconds gathering the RSSI values at each particular sub-area. Finally, the accumulated data, which mainly consists of Wi-Fi access points with signal strength ranges in addition to GPS and cellular triangulation data as supporting location information, will represent one physical defined location to the user.

Any location that is not defined by the user or does not have location information saved on the device will be considered "Unregistered". Therefore, we designate a default policy restrictions for the user to configure whenever the device is located in an unregistered location. In addition, we allow users to register locations that have not been previously visited. This is achieved through either manually entering the publicly known longitude and latitude of the desired location, or by acquiring the fine-grained Wi-Fi parameters from other devices who have saved those parameters. This becomes very practical when the user is switching between two devices and needs to import previously saved policy contexts to the new device.

Our implementation does not store all the GPS or triangulated cellular coordinates acquired, rather a subset of those coordinates that bound into a convex hull and their associated precision. The points in the interior of the convex hull are discarded. We also only store the RSSI range for each distinct Wi-Fi access point scanned. This range is the minimum and maximum RSSI values aggregated from all the sub-areas for each access point. A sub-area is therefore represented as a range of Wi-Fi signal strength values at the least, and if with high position accuracy, also a representation of a convex hull of GPS or triangulated cellular coordinates.

6.2 Location Detection Phase

Figure 3 describes how device context is detected and matched with pre-defined context. Periodically, the location background service is re-instantiated to accumulate location-context data to determine the device's current whereabouts. Like when registering and scanning a sub-area in the location capturing phase, we scan the device's location-related data. The list of user-registered areas that have a subset of the scanned neighboring access points are extracted from the database first. Matching distinct access

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

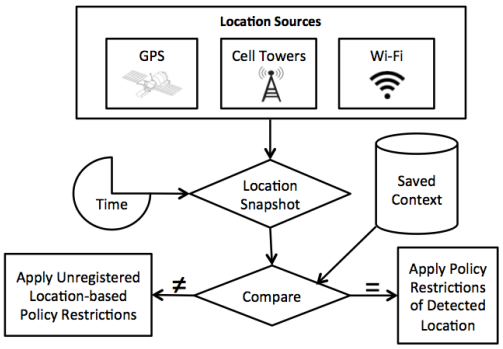

Fig. 3: **Location Detection Phase.**

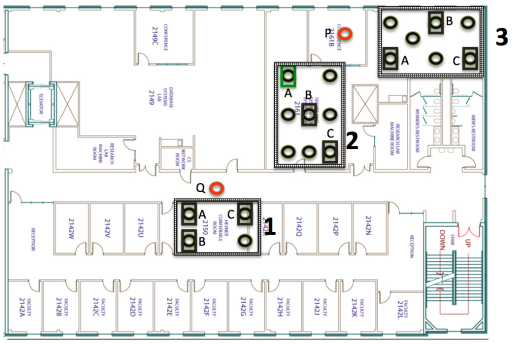

Fig. 4: **Tested areas in one of our campus buildings.**

points is computationally less expensive than determining if coordinate position falls within the boundaries of a convex hull. Then, using the current signal strengths of the access points, we reduce the list to only a set of "best-match" list of physical locations whose access points fall within the current captured signal strength values. If the current scanned GPS or cell network coordinates fall within the convex hull of the associated sub-area, then it is highly likely that the sub-area has been located.

In the unlikely situation when the "best-match" list of physical locations contain more than one location, the user is given the list to confirm his/her location. Even though this event is unlikely to happen, it may still occur because Wi-Fi access point's signal strengths are volatile; their signal strengths fluctuate at a given location. As a consequence, an access point's signal may be, at some point, too weak for the Android's device sensor regardless of whether the device is in motion or stationary. In the location capturing phase, we aggregate the retrieved data that records a range of RSSI values from different sub-areas within the same location, for example, instead of just a single snapshot. In the detection phase, however, we take a snapshot of the location data, which may have only a subset of the previously aggregated data. We compare the previously stored values with the snapshot data. Specifically, with respect to Wi-Fi, we determine if the captured RSSI value of a particular access point is within the stored range. We perform this operation for each access point captured in the snapshot and count the number of tests passed, which is the basis in determining the physical location of the device.

7 Experimental Results

In this section, we report experimental results about the CBAC mechanism and evaluate its impact on the device system and applications. Our modifications to the Android source code were tested on the Android Nexus 4 cellular device and Android Nexus 7 tablet running the Android 4.2.2 OS (API level v. 17). We ran the top 250 applications from the Google Play market for testing and evaluating our modifications. Each experiment has been carried out with the help of the Android Debug Bridge (ADB) utility by using the command "adb logcat". We inserted logging commands in various parts of the operating systems where modifications were made to observe, for example, application access events.

Experiment 1: Location Detection Accuracy. The goal of this experiment is to evaluate the accuracy of the location detection algorithm used in our CBAC mechanism. We measure the number of success and failure detections per sub-area. Figure 4 displays the schematics of one building where we performed some of our experiments. The large, grid-pattern rectangles point out main locations or areas, identified by numbers. Areas outside the rectangles are considered "unregistered." The black circles indicate the specific sub-areas examined during the location capturing phase. All other colors indicate other sub-areas examined during the detection phase.

Figure 4 shows three tested rooms located on the same floor. However, our experimentation included several buildings and areas. In each room, we chose at least four spots to participate in the location capturing phase to accumulate location-related data, in order to construct a robust set of location parameters per room to be stored in the database. In each location, we analyzed three sub-areas, indicated by 'A', 'B', or 'C' and measure the detection rate in each of these subareas. For that particular floor which contains over 15 Wi-Fi access points, we captured the top 5 Wi-Fi access points per snapshot with the highest signal strength for each tested sub-area.

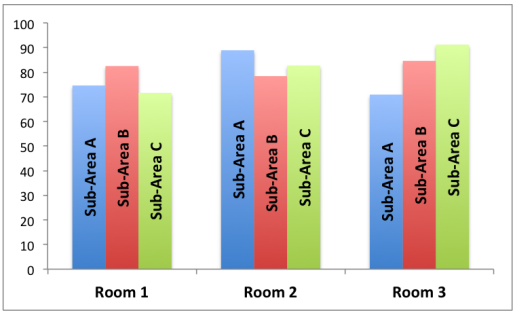

Fig. 5: **Detection accuracy rate of closely located areas.**

Figure 5 displays the detection accuracy rate in the 3 sub-areas of rooms 1, 2 and 3. At each of the sub-areas of each room, we performed 50 location tests and counted the number of successful detections. Our experimental results show that the successful detection rates were up to 91%, and in the worst case scenario we had up to 29% of incorrect detections. This experimental result

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

was complemented by testing several "unregistered" areas around the registered rooms. We detected 16% of false positives, that is, unregistered areas that appeared to be user-defined. Within the registered areas, the values of the signal strengths of matching Wi-Fi access points fell within the range of signal strengths first acquired during the location capturing phase. However, in the unregistered areas, especially the further away the device was when the snapshots at the location capturing phase were taken, the values fell outside the stored range because of building structures hindering the Wi-Fi signal strengths.

Experiment 2: Impact of Permission Restrictions. The purpose of this experiment is to observe the impact of permission-related policy restrictions on applications. Specifically, we are interested in whether or not an application crashes as a result of being denied a permission that was initially granted at installation time. Therefore, we performed a stress test on each application and observed the impact on the application upon revoking its permissions when requesting a service or resource. We performed our experiment on 245 Android applications and used the ADB logging utility to view the permission being revoked when the *checkComponentPermission()* method is called.

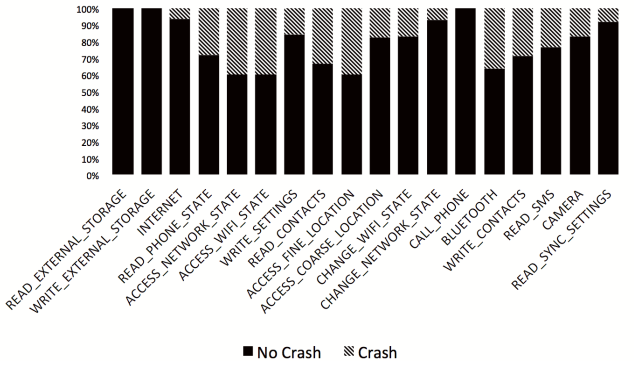

Fig. 6: **Impact of permission revoking on applications.**

Figure 6 shows the percentage of application crashes upon performing the stress test on each permission. We counted an application as crashing even if it crashed during the execution of one minor functionality. The main cause of these crashes is due to the developers' mishandling the denial of previously granted permissions. Since application crashes are due to developers' mishandling the denial of previously granted permissions to their applications, application crashes can be prevented if error-handling is added whenever an application attempts to access a resources or request a service. In fact, throughout testing several application versions, we realized that the number of application crashes has been decreasing over time. This is because developers are now aware that not having the permission error-mishandling script is causing several application crashes. A script is thus being added in their application updates especially with the evolvement of many permission restriction techniques.

Experiment 3: Performance Overhead. The purpose of this experiment is to evaluate the timing overhead introduced by our modifications to the Android OS. We calculate the amount of time it takes for our modified methods to fully execute once called by applications. We also compare the execution times of these methods before our modifications to estimate the overhead introduced by our modifications. Specifically, we measure the overhead time caused by intercepting application permissions, user data accesses (e.g. *Contacts*), *Intent* messages, and access to system peripherals (e.g. Bluetooth).

Method	Overhead
checkComponentPermission(..)	12.220
Intent-startActivity(..)	12.708
Intent-startService(..)	5.402
Intent-sendBroadcast(..)	5.208
User Data-ContentResolver(..)	12.300
Device Peripherals-setEnable(..)	8.351

TABLE 3: **Time overhead (in milliseconds) for some of the core Android methods that were modified.**

Table 3 reports in milliseconds the time imposed on these methods. As the results show, the overall delay introduced by enforcing our CBAC policies is not perceivable by the end-user.

Experiment 4: System Memory Overhead. The purpose of this experiment is to measure the amount of memory overhead placed on the system after our modifications. Mainly, we aim to observe the changes in memory usage caused by our application restrictions and by the *LocationService* method that continuously run in the background for context updates.

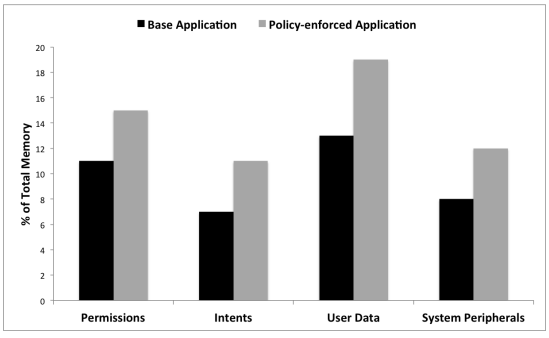

Fig. 7: **Total memory overhead comparison with and without our CBAC policy restrictions.**

Figure 7 shows that the memory usage when enforcing our CBAC policies closely matches the memory usage when these policies are not enforced. Even though our experiments test the different restriction categories separately as shown in the figure, we believe that the observed memory overheard is due to the *LocationService* that is instantiated periodically to keep the device's context up-to-date.

Experiment 5: Battery Consumption. The purpose of this experiment is to observe the Android device's battery consumption change when CBAC policies are enforced compared to when they are not. For this purpose, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

monitored the device's battery percentage when running both the unmodified OS and our customized system, separately. In both cases, we forced the device's screen to never turn off with Wi-Fi and GPS enabled, a representation of a somehow worst case scenario as when the user is continuously using the device for that duration. Since the period for which the *LocationService* method that is responsible for checking the device's location can be customized by the user, we tested the battery consumption for different time periods for the purpose of getting a fair evaluation.

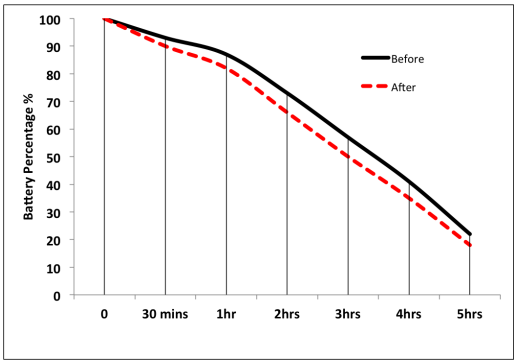

Fig. 8: **Comparison of device battery consumption when checking for context updates every 30 seconds.**

We started our experiment by setting the device to check for any device location updates every 30 seconds. Figure 8 shows that the battery percentage displayed on the device when enforcing our CBAC policies drops $\sim 5\%$ less per hour compared to when the policies are not enforced. We achieved similar results when context was updated every 45 seconds. However, when we set the timer to check for device context every 60 seconds or above, the battery usage percentage of when the CBAC policies are enforced closely matches the battery consumption when the policies are not enforced.

8 Security Analysis

In this section, we present a security analysis of our implementation of the CBAC system to analyze possible threats from a malicious user or applications that can bypass our policy restrictions. The aim of our security analysis is to identify possible threats by malicious users or applications that can bypass CBAC policies and to mitigate these threats.

Colluding Applications. In Android, each device application is assigned a unique UserID (UID) that the system uses to refer to an application. However, if two applications are created and signed by the same developer, the system will give both applications the same UID, which gives these applications the ability to share the same processes if needed [22]. The stock Android OS applies its security policies not based on the application label or its package name, but rather on the process UID. In our modifications of the OS, we obtain the name of the package (application) which is performing an action by calling the PackageManager's *getPackagesForUid(int uid)*. This way, our restrictions are not based on UID but are transformed in order to refer to the package name. As an example of such threat, consider two applications, Application A and Application B, that share the same UID. Suppose that the user blocks access to GPS capabilities from application A. However, although successfully blocked, application A may still be able to acquire information about the user's or device's location because Application B was not denied access to GPS. In our system, we prevent such threat by blocking *all* package names associated with a UID using the *getPackageForUid()* method.

Circumventing Application Multitasking Restriction. A malicious developer may attempt to bypass the restriction that disallows multiple applications from running simultaneously by creating a custom launcher-like app. Android is modular, and thus the default home screen can be replaced. Our system is not vulnerable to such an attack because we check all possible intents as our system is not limited to intents related to the stock *Launcher*. Thus any intent with "android.intent.action.MAIN" and "android.intent.category.LAUNCHER" will be intercepted and processed to disable multitasking of any launcher the user decides to use.

Protection of Policies. As users can configure policy restrictions based on time and location, these restrictions are either applied system-wide or per application. If these policy restrictions can be altered by applications, then any malicious application can perform specific attacks based on policy configurations. To protect policies, we thus do not allow write privileges to be granted on policies so to prevent policies from being modified.

Malicious applications that are aware of our CBAC policies may try to drop a policy or modify the device's detected context so that the wrong policy is applied. However, in our implementation we retrieve context information directly from the system protected APIs that cannot be altered by applications. Moreover, context information is managed by our *Content Provider* that gathers such information regardless of which applications are running on the device or services requested by applications. This independency from the *Content Provider* gives robustness in gathering context data that is forwarded to the *Policy Manager* as discussed in Section 3.

Sensitive Information Disclosure. Some applications may maliciously leak private user information once they detect that a previously granted privileged application is revoked. As an example, a malicious application that was granted access to *Camera* and *GPS* at installation time may upload the device GPS location to the application server once it detects that the *CAMERA* permission is revoked, leaking the user-private location. For this reason, our implementation provides the user, at the time of policy configurations, with a list of privileges that are still granted to the configured applications, acting as a warning for the user to be aware of the sensitive information still accessible by the configured applications in a particular context.

Continuous Running Application Processes. When an

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2014.2320731, IEEE Transactions on Dependable and Secure Computing

application requests access to a resource, the Android OS checks if the application has the appropriate permission(s) just at the time of the request. If the user-configured policy grants such permission to the requesting application in a given context, some processes associated with certain resources may continuously run even if the device is later located in a different context for which the user has denied access to such resource. The reason is that permission granting is not checked continuously while the process is running, rather is only checked when the request is issued. Malicious applications may take advantage of this, for example by continuously recording audio in one context while transitioning to another context.

Audio recording using the *Microphone* resource is one example of a continuously running process that will not terminate until the recording is stopped. Take for example a user who attends private meetings in a same meeting room that he configured a policy to disable the *Microphone* resource. A malicious application can begin recording outside of the meeting room area without alerting the user, and continue recording when the user enters the restricted meeting room. The Android OS does not continuously verify whether an application has audio recording permission during recording. It verifies each time a request is made, and thus when approved the application can continue using the peripheral for that specific session. Our implementation prevents this type of attack. Once a registered area is associated with a restriction on video or record audio access, the location service forces the applications with the associated permissions in their *AndroidManifest.xml* to close.

9 Related Work

The area of location-based policies on mobile phones is related to work done in the areas of access control policies and techniques for reliable location information. Also, since our work applies policy restrictions on Android applications, our work is also related to work on Android restriction techniques.

Several approaches have been proposed for context-based access control. Generalized Role Based Access Control (GRBAC) is an approach that incorporates the concept of environment information (such as time) into access control [23]. GRBAC is very expressive and thus suitable for context aware authorization [24], [25]. However GRBAC may not be feasible in practice due to the large amounts of environment roles that make the system very hard to maintain. Zhang *et al.* proposed dynamic RBAC (DRBAC) that dynamically adjusts for roles and permissions based on context information [26]. GEO-RBAC is another approach that incorporates spatial awareness into role-based access control [27]–[29]. However, these models are conceptual and just focus on high level abstraction that does not specify on how to deploy these approaches on real implementations. For the purpose of applying those models to real implementations, Sandhu *et al.* proposed the notion of PEI (policy, enforcement, implementation) models that define a usable structure for creating an implementation of enforcement mechanisms [30]. Kirkpatrick *et al.* defined such enforcement mechanism and emphasized the importance of authenticating the user's claim to a particular context through incorporating the NFC technology into an access control mechanism for information systems [10]. Gupta et al. proposed a context profiling framework based on the surrounding environment captured by the mobile device sensors to estimate the familiarity of a place [11]. They used such context to create context profilers that are used to configure access control policies on mobile devices.

Policy-specification languages also relate to our work as they are intended to simplify the task of specifying and enforcing security policies on untrusted software. Examples of expressive policy-specification languages are Ponder [31], XACML [32], PoET/PSLang [33], Naccio [34], Polymer [35] and Deeds [36]. However, none of these languages provide user constructs to manage location information. On the other hand, OpenAmbient [37] and GEO-RBAC [38] are examples of location-based policy-specification languages able to manipulate location information but may fail in dynamically changing roles or location that we require in our CBAC policies.

Our work also relates to techniques for providing reliable location information that can provide adequate security guarantees once merged with security policies. GPS is the most commonly used technology for location purposes. However it is not very reliable when used indoor due to its low signal power [14]. *CellID, Bluetooth, or NFC* technologies have been extensively tested for proximity of device location [39]. However Madlmayr *et al.* [40] have shown that these technologies are not always reliable as they are vulnerable to eavesdropping and collusion attacks. Therefore, our choice of using the Wi-Fi access points *RSSI* values seems to be the cheapest and most accurate as it does not require any extra hardware and is capable of differentiating between closely located sub-areas. We benefit from techniques used for indoor localization and tracking purposes to define the exact location parameters needed to form an accurate indoor location [18]–[21], [41].

Our work is also related to techniques implemented for the Android OS to restrict device applications for protecting the user privacy and device security. Past work has focused on detecting applications that are over-privileged [42] and categorized them based on their permission requirements before installation, and their potential threats in case such permissions are granted and abused. Enck *et al.* [43] also built the Kirin tool that uses a formal model of policy mechanisms to determine whether or not the privileges requested by applications match the desired functionality. However, even though these mechanisms may solve the problem of over-privileged applications, but they may not be able to accurately differentiate between over-privileged and malicious or buggy applications, which represent our main threat. Roesner *et al.* [44] introduced user-driven access control gadgets based on user intent, allowing Android users to grant resource privileges to

their applications at the time of use. Our mechanism offers these controls with a larger scope of restrictions that give users a comprehensive control over their device, with all restrictions tied to context. Other researchers have implemented detection mechanisms for malicious applications [45]–[47], some of which are responsible for user private information.

There are few approaches that have similar goals of our. The first [48] defines a language for specifying location-dependent runtime security policies. However our approach supports a wider range of policy restrictions and enforces them more efficiently on the Android system. In addition, our technique is able to acquire more accurate location information. Other closely related approches are AppFence [16], Apex [49], TISSA [50], and IdentiDroid [17] that have developed modifications to the Android OS in order to limit data leakage and restrict application permissions. Our work complements these techniques by adding more user controls and device restrictions (such as intent management) and ties these configurations to context-based policies that dynamically apply device restrictions. Our work also complements research efforts in protecting user and application data applied at the middleware and kernel layers of the Android OS, such as FlaskDroid [51], Moses [52], Saint [53], and TrustDroid [54].

10 Conclusion and Future Work

In this work, we proposed a modified version of the Android OS supporting context-based access control policies. These policies restrict applications from accessing specific data and/or resources based on the user context. The restrictions specified in a policy are automatically applied as soon as the user device matches the pre-defined context associated with the policy. Our experimental results show the effectiveness of these policies on the Android system and applications, and the accuracy in locating the device within a user-defined context.

Our approach requires users to configure their own set of policies; the difficulty of setting up these configurations require the same expertise needed to inspect application permissions listed at installation time. However we plan to extend our approach to give network administrators of organizations the same capabilities once a mobile device connects to their network. In this way, network administrators are able to block malicious application accesses to resources and services that may affect the security of their network. We believe that such an approach is critical for assuring security of corporate networks when organizations allow users to "bring their own devices".

References

[1] Wikipedia, "Samsung galaxy s4 specifications," http://en.wikipedia.org/wiki/Samsung_Galaxy_S4, May 2013.

[2] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10, Berkeley, CA, USA, 2010.

[3] J. Leyden, "Your phone may not be spying on you now - but it soon will be," http://www.theregister.co.uk/2013/04/24/kaspersky_mobile_malware_infosec/, April 2013.

[4] R. Templeman, Z. Rahman, D. J. Crandall, and A. Kapadia, "Placeraider: Virtual theft in physical spaces with smartphones," *CoRR*, vol. abs/1209.5982, 2012.

[5] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones," in *Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2011.

[6] L. L. N. Laboratory, "Controlled items that are prohibited on llnl proporty," https://www.llnl.gov/about/controlleditems.html.

[7] M. Conti, V. T. N. Nguyen, and B. Crispo, "Crepe: context-related policy enforcement for android," in *Proceedings of the 13th international conference on Information security*, ser. ISC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 331–345.

[8] A. Kushwaha and V. Kushwaha, "Location based services using android mobile operating system," *International Journal of Advances in Engineering and Technology*, vol. 1, no. 1, pp. 14–20, 2011.

[9] S. Kumar, M. A. Qadeer, and A. Gupta, "Location based services using android," in *Proceedings of the 3rd IEEE international conference on Internet multimedia services architecture and applications*, ser. IMSAA'09, 2009, pp. 335–339.

[10] M. S. Kirkpatrick and E. Bertino, "Enforcing spatial constraints for mobile rbac systems," in *Proceedings of the 15th ACM symposium on Access control models and technologies*, ser. SACMAT '10. New York, NY, USA: ACM, 2010, pp. 99–108.

[11] A. Gupta, M. Miettinen, N. Asokan, and M. Nagy, "Intuitive security policy configuration in mobile devices using context profiling," in *IEEE International Conference on Social Computing*, ser. SOCIALCOM-PASSAT '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 471–480.

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

[13] E. Trevisani and A. Vitaletti, "Cell-id location technique, limits and benefits: an experimental study," in *Mobile Computing Systems and Applications, 2004. WMCSA 2004. Sixth IEEE Workshop on*, 2004, pp. 51–60.

[14] J. LaMance, J. DeSalas, and J. Jarvinen, "agps: A low-infrastructure approach," http://www.gpsworld.com/innovation-assisted-gps-a-low-infrastructure-approach/, March '02.

[15] "Sky hook," http://www.skyhookwireless.com/.

[16] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *18th ACM conference on Computer and communications security*, ser. CCS '11, NY, USA.

[17] P. under submission, "Identidroid: Android can finally wear its anonymous suit."

[18] I. F. Progri, "Wireless-enabled gps indoor geolocation system," in *Position Location and Navigation Symposium (PLANS), 2010 IEEE/ION*, 2010, pp. 526–538.

[19] C. Feng, W. Au, S. Valaee, and Z. Tan, "Received-signal-strength-based indoor positioning using compressive sensing," *Mobile Computing, IEEE Transactions on*, vol. 11, no. 12, pp. 1983–1993, 2012.

[20] S. Ali-Loytty, T. Perala, V. Honkavirta, and R. Piche, "Fingerprint kalman filter in indoor positioning applications," in *Control Applications, (CCA) Intelligent Control, (ISIC)*, 2009, pp. 1678–1683.

[21] A. S. Paul and E. Wan, "Rssi-based indoor localization and tracking using sigma-point kalman smoothers," *Selected Topics in Signal Processing, IEEE Journal of*, vol. 3, no. 5, pp. 860–873, 2009.

[22] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, Feb. 2012.

[23] M. Moyer and M. Abamad, "Generalized role-based access control," in *Distributed Computing Systems, 2001. 21st International Conference on.*, 2001, pp. 391–398.

[24] M. J. Covington, W. Long, S. Srinivasan, A. K. Dev, M. Ahamad, and G. D. Abowd, "Securing context-aware applications using environment roles," in *Proceedings of the sixth ACM symposium on Access control models and technologies*, ser. SACMAT '01. New York, NY, USA: ACM, 2001, pp. 10–20.

[25] C. Wullems, M. Looi, and A. Clark, "Towards context-aware security: an authorization architecture for intranet environments,"

in *Pervasive Computing and Communications Workshops*, 2004, pp. 132–137.

[26] G. Zhang and M. Parashar, "Dynamic context-aware access control for grid applications," in *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, 2003, pp. 101–108.

[27] M. L. Damiani, E. Bertino, B. Catania, and P. Perlasca, "Geo-rbac: A spatially aware rbac," *ACM Trans. Information System and Security*, vol. 10, no. 1, 2007.

[28] D. Kulkarni, "Context-aware role-based access control in pervasive computing systems," in *SACMAT08 Proceedings of the 13th ACM Symposium on Access control Models and Technologies*, 2008.

[29] R. J. Hulsebosch, A. H. Salden, M. S. Bargh, P. W. G. Ebben, and J. Reitsma, "Context sensitive access control," in *Proceedings of the tenth ACM symposium on Access control models and technologies*, ser. SACMAT '05. New York, NY, USA: ACM, 2005, pp. 111–119.

[30] R. Sandhu, K. Ranganathan, and X. Zhang, "Secure information sharing enabled by trusted computing and pei models," in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, ser. ASIACCS '06. New York, NY, USA: ACM, 2006, pp. 2–12.

[31] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, ser. POLICY '01. London, UK, UK: Springer-Verlag, 2001, pp. 18–38.

[32] U. Erlingsson, "Oasis extensible access control markup language (xacml)," https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[33] Erlingsson and F. Schneider, "Irm enforcement of java stack inspection," in *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, 2000, pp. 246–255.

[34] D. Evans and A. Twyman, "Flexible policy-directed code safety," in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, 1999, pp. 32–45.

[35] L. Bauer, J. Ligatti, and D. Walker, "Composing expressive runtime security policies," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 3, pp. 9:1–9:43, Jun. 2009.

[36] G. Edjlali, A. Acharya, and V. Chaudhary, "History-based access control for mobile code," in *Proceedings of the 5th ACM conference on Computer and communications security*, ser. CCS '98. New York, NY, USA: ACM, 1998, pp. 38–48.

[37] C. A. Ardagna, M. Cremonini, E. Damiani, S. D. C. di Vimercati, and P. Samarati, "Supporting location-based conditions in access control policies," in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, ser. ASIACCS '06. New York, NY, USA: ACM, 2006, pp. 212–222.

[38] E. Bertino, B. Catania, M. L. Damiani, and P. Perlasca, "Geo-rbac: a spatially aware rbac," in *Proceedings of the tenth ACM symposium on Access control models and technologies*, ser. SACMAT '05. New York, NY, USA: ACM, 2005, pp. 29–37.

[39] "Nfc forum tag type technical specification," http://www.nfc-forum.org/home/.

[40] G. Madlmayr, J. Langer, C. Kantner, and J. Scharinger, "Nfc devices: Security and privacy," in *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, 2008, pp. 642–647.

[41] A. Kushki, K. Plataniotis, and A. Venetsanopoulos, "Intelligent dynamic radio tracking in indoor wireless local area networks," *Mobile Computing, IEEE Transactions on*, vol. 9, no. 3, pp. 405–419, 2010.

[42] T. Vidas, N. Christin, and L. Cranor, "Curbing Android permission creep," in *Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011)*, Oakland, CA, May 2011.

[43] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09, NY, USA.

[44] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 224–238.

[45] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11, NY, USA, 2011, pp. 3–14.

[46] W. Enck, "Defending users against smartphone apps: techniques and future directions," in *Proceedings of the 7th international conference on Information Systems Security*, ser. ICISS'11. Springer-Verlag, 2011, pp. 49–70.

[47] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 21–21.

[48] J. Ligatti, B. Rickey, and N. Saigal, "Lopsil: A location-based policy-specification language."

[49] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '10. New York, NY, USA: ACM, 2010, pp. 328–332.

[50] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *Proceedings of the 4th international conference on Trust and trustworthy computing*, ser. TRUST'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 93–107.

[51] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies," in *22nd USENIX Security Symposium (USENIX Security '13)*. USENIX, Aug. 2013.

[52] G. Russello, M. Conti, B. Crispo, and E. Fernandes, "Moses: supporting operation modes on smartphones," in *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, ser. SACMAT '12. New York, NY, USA: ACM, 2012, pp. 3–12.

[53] M. Ongtang, S. Mclaughlin, W. Enck, and P. Mcdaniel, "Semantically rich application-centric security in android," in *In ACSAC 09: Annual Computer Security Applications Conference*.

[54] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 51–62.

Bilal Shebaro is a post doctoral researcher at Purdue University affiliated with Purdue Cyber Center (Discovery Park) and Purdue Center for Education and Research in Information Assurance and Security (CERIAS). He received his Ph.D. in May 2012 from the Computer Science department at the University of New Mexico. During his graduate studies, he was also a research assistant at the Los Alamos National Laboratories and a research scholar at the Universidad de Vigo in Spain. His recent research focuses on mobile security, digital forensics, and sensor networks.

Oyindamola Oluwatimi, under the DNIMAS administration, graduated with a Bachelor of Science from Norfolk State Universitys Computer Science (CS) undergraduate program in 2011. In the fall of that year, with both GEM and Purdue Doctoral fellowships, he began attending Purdue University's CS Ph.D. academic program. His focus is in computer security, and has recently picked up an interest in mobile security and privacy. He is currently working under Dr. Elisa Bertino as a research assistant, collaborating in projects involving enhancements of mobile operating systems with privacy-related features.

Elisa Bertino is professor of computer science at Purdue University, and Director of the Purdue Cyber Center (Discovery Park). She also serves as Research Director of the Center for Information and Research in Information Assurance and Security (CERIAS). Prior to joining Purdue, she was a professor and department head at the Department of Computer Science and Communication of the University of Milan. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose, at the Microelectronics and Computer Technology Corporation, at Rutgers University, at Telcordia Technologies. Her recent research focuses on database security, digital identity management, policy systems, and security for web services. She is a Fellow of ACM and of IEEE. She received the IEEE Computer Society 2002 Technical Achievement Award and the IEEE Computer Society 2005 Kanai Award. She a member of the editorial board of IEEE Transactions on Dependable and Secure Computing, and IEEE Security and Privacy. She has served as chair of the ACM Special Interest Group on Security, Audit and Control (ACM SIGSAC).