A Distributed Android Security Framework

***Abstract*—As mobile devices continue to become increasingly ubiquitous in society, the demand for incorporating their use in the healthcare industry is concurrently rising. However, the current security models on mobile devices do not provide the level of protection that is needed for the devices to handle sensitive data, such as protected healthcare information. In this paper, we introduce a Distributed Android Security Framework (DASF). DASF is a custom security framework for Android-based mobile devices that allows dynamic security policies to be enforced on an application's privileges and sensitive data. DASF allows a server to dynamically impose the security policies by utilizing an application-layer message protocol that is implemented in the system. The security policies enforced by DASF will allow for system-wide privilege restrictions to be enforced on untrustworthy applications and sensitive data. Ultimately, DASF will allow the organization that issues the mobile devices to dynamically dictate the security policies enforced by the system and retain control of the sensitive data that they send to the device.**

I. Introduction

The incorporation of mobile devices in the healthcare industry can prove to be another beneficial tool for medical professionals in a variety ways. For example, medical professionals can use mobile devices as a general reference tool to assist them with accurately diagnosing patients, as an informative tool to educate patients on the treatment of their condition, as a communicative tool to collaborate with other medical professionals, or to allow quick access to a patients medical records in an emergency. Furthermore, utilizing mobile devices to allow medical professionals to access a patients medical records also provides the additional benefit of obviating the need of paper-based medical records during a patients visitation. In regard to the sensitivity of medical records, electronic medical records are superior to paper-based medical records because the file server can record a comprehensive access log of every individual that has requested the medical record and can also enforce automated authorization controls to prevent unnecessary or unauthorized access.

As the healthcare practice is collaborative in nature, doctors and nurses frequently communicate with each other to provide a diagnosis and treatment. While the appeal of modern communication through mobile devices, e.g., smartphones or tablets, has reached almost every industry, the following observations witness the status of mobile devices in healthcare industry. First, a significant number of physicians and nurses are still relying on traditional clinical communication equipment, including pagers, phones and fax machines, yet a last generation wireless communication technology. Second, even though as many as 81% healthcare prefessionals own and use the smartphones themselves every day [2], their devices either are not integrated into the healthcare communication system (so that pagers are still needed), or cannot be trusted as the system components due to lacking of the security mechanism compliant to Health Insurance Portability and Accountability Act (HIPAA). It is reported [14] that lost, stolen, and non-HIPAA compliant healthcare mobile applications are blamed for protected health information (PHI) breach on electronic health record (EHR) system.

To take advantage of modern communication means supported by the mobile devices, such as multimedia (text, image, and video) messaging and conferencing, and fully employ the emerging technologies in healthcare industry with HIPAA compliance, a system-wide security safeguard must be included to defend against the potential security breaches. Checking the HIPAA privacy rule [1], we consider the following three security challenges for the deployment of the mobile devices in healthcare industry.

- *Network security:* The user authentication and data encryption schemes must be robust to protect PHI transmitted in the communication channel and defend against various security attacks.
- *Dynamic security demands and fine-grained access control policies:* Access control policies should be flexible enough to meet the dynamic security demands in the various scenarios that the mobile devices will be utilized. Furthermore, the healthcare organization (e.g., hospital) should be able to dynamically enforce system-wide access control policies on the mobile devices to allow the organizations to dictate their own security policies.
- *PHI propagation control and revocation:* Healthcare organizations should be able to control the propagation of the PHI and revoke access to the PHI that is sent to the mobile device. For example, the healthcare organization should be able to prevent PHI from being saved to the device (e.g., saving a medical record as a file on the device) or forwarded to another device (e.g., sending a medical record to another device).

To address the aforementioned security challenges and meet the requirements of the HIPAA regulations, we select the Android mobile operating system as the platform to develop our custom security framework. However, the existing security model on the Android platform fails to address the security challenges. In particular, Androids current security model supports static installation-time privileges and does not permit the dynamic restriction of previously granted privileges. Furthermore, Androids security model also does not include a mechanism to control the propagation and revocation of sensitive data on the device.

This paper introduces a Distributed Android Security Framework (DASF). DASF is a custom security framework for

Android-based mobile operating systems designed to provide dynamic privilege restrictions on applications and security policies on sensitive data on the device. Moreover, DASF includes an application-layer networking protocol (in regard to the OSI model) that allows the organization (e.g., hospital, clinic, etc.) to dictate their own security policies on the device and on data sent to the device. DASF allows an applications access privileges to be flexibly set up upon invocation and adjusted on-the-fly to meet the various security demands when dealing with sensitive information. In mobile devices that utilize DASF, the organization ultimately controls the security policies that the system enforces on the users device and on the sensitive data that applications receive from the server.

Our contributions in this paper can be summarized as follows. First, we use the medical communication system as an example to study the security problems and the HIPAA compliance issues caused by the mobile devices. Second, we design a distributed Android security framework that not only provides dynamic system-wide security provisioning, but also protects the privacy of the received sensitive data and therefore defeats the PHI breach on mobile devices. Third, we implement a prototype Android-based medical communication system and deploy our proposed security framework. Our evaluation shows DASF is capable of defeating a various security attacks while only imposing a reasonable system overhead.

II. Attack Model and Security Assumptions

This work focuses on the security challenges of enforcing security policies on sensitive data sent to Android-based mobile devices and enforcing dynamic privilege restrictions. Moreover, this work is focused on providing the security guarantees at the system-level rather than at the application-level. Thus, the security policies of the server and database management system are out of scope of this paper. Therefore, we assume that the data is protected in the server and the database management system by the appropriate security measures. Furthermore, we assume that the server ensures that the communication channel between the server and the application running on the mobile device is secure (e.g., using SSL) and each individual involved in the communication has his/her security credentials (including digital certificates, public and private key, etc.) distributed through an out-of-band security channel.

The mobile devices are not trusted. Once the mobile device is lost or stolen, the adversaries (who have the possession of the missing devices) can capture all stored data (both in flash memory and main memory) and may try all possible means to recover the sensitive data. Further, all other third party Android applications installed on the mobile devices (including medical applications) are not trusted and may launch attacks to try to gather sensitive information. Since the Android platform that we augmented with DASF is transparent to application packages, malicious applications may be installed and attempt to steal sensitive data from the device or the environment. We assume the communication parties are semi-trusted. That is, the user may misbehave by themselves (e.g., trying to save or forward a patients MRI image without proper authorization) but do not conspire with the server.

Therefore, we consider the following four different types of security threats for which the system must provide protection.

- *Legitimate user misbehavior:* The system must prevent legitimate users from manually attempting to save or forward sensitive information.
- *Malicious users:* The system must prevent sensitive information from being recovered from the device if it is lost or stolen.
- *Application misbehavior:* The system must prevent legitimate applications from attempting to save or forward sensitive information that is not authorized by the organizations policies. For example, it is typical to write an image to a file when it is received rather than storing the entire image in RAM. However, writing the image to a file may violate the organizations policy and must be prevented by the system.
- *Malicious applications:* The system must prevent malicious applications from attempting to steal sensitive information stored on the device (e.g., PHI information authorized to be saved to the device) or from the environment (e.g., verbal conversations occurring between patients and doctors).

III. Motivation

We begin with a brief overview of Android and then discuss the limitations of Android's security model. Afterward, we present how DASF addresses the limitations of the existing security model on Android.

A. Android Background

Android is a software stack for mobile devices that is developed and managed by the Open Handset Alliance. The Android software stack is comprised of a Linux-based kernel, a middleware layer consisting of the native libraries and the runtime environment, and an application layer consisting of the application framework and applications Fig. 1.

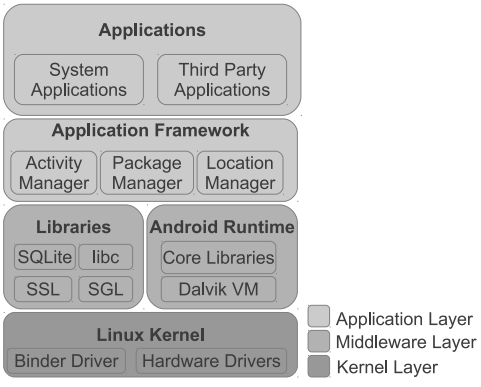


Fig. 1. Android Software Stack

The Linux-based kernel is located at the lowest layer of the Android software stack. The kernel is responsible for

providing the core functionality of Android, such as process management, networking, memory management, and device drivers to allow for the software to communicate with the hardware resources. Furthermore, the Linux kernel provides Android with the security functionality needed to enforce process isolation, hardware access privileges, and file access privileges through utilizing user identifiers (UIDs), group identifiers (GIDs) and file system access permissions. Applications that are installed typically have a unique UID, however, Android also allows applications that are signed with the same signature to share a UID to allow two different applications to access each others resources (e.g., files). However, since applications typically run in an isolated process under a unique UID, Android includes a mechanism in the Linux kernel (Binder) that allows inter-process communication.

The middleware layer is located above the kernel layer of the Android software stack, which consists of the native libraries and the runtime environment. The native libraries provide the system with machine independent functionality, such as the libc library, a SSL implementation, and an SQLite implementation. The runtime environment consists of the core libraries and the Dalvik virtual machine. The core libraries provide the system with an implementation of the Java API, the Android libraries, and the Dalvik Virtual Machine's libraries. The Dalvik Virtual Machine is a process-level virtual machine that is responsible for executing Android applications that were compiled into Android's custom bytecode (DEX). The middleware layer also provides the system with a reference monitor that enforces mandatory access control (MAC) for inter-component communications.

The application layer is located at the top of the Android software stack and contains the application framework and applications. The application framework provides Android application developers with basic blocks with which applications directly interact, such as the activity manager, package manager, location manager, and content providers. Applications are at the very top of the Android software stack and provides the user with software to perform specific tasks, such as a contact manager, a web browser, or a text messaging interface. Android applications are distributed as an application package file (APK), which contains a manifest file that declares the application's requested permissions and components, the DEX bytecode of the application, resources, and application certificates. Applications may also define their own custom permissions in the manifest file to enforce access control policies on the components or data that they expose to other applications. Furthermore, applications may either be *system applications* (signed with the same signature as the system image) or *third-party applications*.

The access control policies of applications in the Android system are achieved through the static installation-time permission model, which are declared in the application's manifest file and enforced by the reference monitor in the middleware layer or by the Linux kernel. An application's request to access sensitive data (e.g., a user's contacts) or hardware resources (e.g., the microphone, camera, etc.) must be granted by the user at the time of the application's installation. Once the requested permissions are approved and the application is installed, the application will always be granted these permissions.

B. Limitations of the Android Security Model

While Android's permission-based security model suffices for most applications, these static access controls can not address the security requirements of applications targeted towards the healthcare industry.

First, Android's current permission model does not support the dynamic restriction of an application's privileges. Since PHI is not limited to medical records and images (e.g., MRI images), but rather includes conversations between a doctor and a patient during an appointment, malicious applications may exploit the environment in which the mobile device is located to gather PHI instead of attempting to steal data stored on the device. For example, a doctor may download a third-party voice memo recorder to record reminders of non-sensitive information. However, the application may be malicious and launch a background service to secretly record conversations that include details of PHI and transmit the recorded audio to an external server. Therefore, a mechanism to dynamically restrict an application's privileges is necessary to prevent the exposure of PHI by malicious applications.

Second, Android's current framework also lacks the system-level mechanism that is required to enforce security policies on PHI data flows and revoke access to PHI. Due to the sensitivity of PHI, the system is required be able to identify and revoke access to the PHI and enforce security policies on the data (e.g., whether the PHI can be saved or forwarded). On Android's current framework the organization loses control the PHI once it is sent to the device, which is unacceptable when dealing sensitive data.

C. Addressing the Limitations of the Android Security Model

To address the limitations of Android's existing security model and the previously discussed security challenges for deploying mobile devices in the healthcare industry, DASF consists of three new modifications to the Android platform.

First, DASF provides a system-level mechanism for dynamically imposing and enforcing system-wide privilege restrictions on applications. Applications can programmatically impose dynamic system-wide privilege restrictions on other applications if they request permission to restrict a privilege.

Second, DASF utilizes dynamic taint tracking to identify sensitive data and track the propagation of the sensitive data throughout the system. We utilize TaintDroid [9] to provide the mechanism for tagging data and propagating the data's tag throughout the system. Further, we also leverage CleanOS' [22] extension of TaintDroid to provide the mechanism for revoking sensitive data. DASF uses the data's privacy tags to enforce system-level security policies on the sensitive data. DASF augments these models by allowing system-level security policies to be imposed on the data (e.g., preventing the data from being saved or forwarded).

Third, we designed a application-layer message protocol at the system-level that operates over an SSL connection to tag sensitive data that is received from the server to allow the server to impose system-level security policies on the data that it sends to the device. Further, our message protocol also allows the server to dynamically impose system-wide privilege restrictions on applications running on the device.

Therefore, our contributions are providing system-level support for classifying sensitive data and enforcing security policies on that sensitive data, and allowing a server, or an application, to dynamically enforce system-wide privilege restrictions on the device.

IV. Security Framework Overview

In this section, we begin by explaining the security policy model of DASF that we designed. Afterward, we focus on the architecture and implementation of the DASF prototype on Android.

A. Security Model

On an Android device, there are a set of applications, $A := \{a_1, a_2, \ldots, a_n\}$. Each application, $\forall a_i \in A$, where $1 \leq i \leq n$, contains a set of permissions that the application requests, $P_{a_i} := \{p_1, p_2, \ldots, p_m\}$, and a set of components, $C_{a_i} := \{c_1, c_2, \ldots, c_q\}$. Each application is either a system application, S_A, or a third-party application, T_A, such that $S_A \subset A$, and $T_A \subset A$, but $S_A \cap T_A = \{\}$.

Dynamic Privilege Restriction Security Policy. A permission, p, is restricted if the restricted permission, denoted as $R(p)$, is imposed by a specific application a_i. In Android, permissions are granted at the granularity of the entire application. If the permissions of a specific application, a_i, is granted, its corresponding components, C_{a_i} are allowed to be launched. Obviously, for any application that can successfully run, each of its components has to have the permission to be launched. We use $L(C_{a_i})$ to denote the launching of a_i's components: $c_1, c_2, ..., c_q$. Furthermore, we denote killing the process that an application, a_i, is running under as $K(a_i)$.

Therefore, the security policy for an application is a procedure of imposing the restricted permissions which determine whether or not the application and its components can be launched. Given an application a_i, its components launching, $L(C_{a_i})$, will be successful as long as one the security policy satisfy any one of the following conditions:

$$\begin{cases} a_i \in S_A \\ a_i \in T_A \wedge R(p) \wedge p \in P_{a_i} \wedge a_i = a_j \\ a_i \in T_A \wedge \neg R(p) \wedge p \in P_{a_i}. \end{cases}$$

More significantly, DASF is able to impose the policy on application that are already running when a permission is dynamically restricted for a security purpose. In that case, any running application a_i with the following updated policy:

$$(a_i \in T_A) \wedge R(p) \wedge (p \in P_{a_i})$$

will lead to $K(a_i)$, i.e., be terminated. And the running application status with the following new policy:

$$\begin{cases} a_i \in S_A \\ a_i \in T_A \wedge \neg R(p) \wedge p \in P_{a_i} \wedge a_i = a_j \end{cases}$$

will become $\neg K(a_i)$, i.e., not be affected.

In other words, an application's components are always allowed to start if they belong to a system application. Moreover, system applications are never forcibly closed if a permission is dynamically restricted. If the application is a third-party application, however, the following policies are enforced. First, a third-party application's components are not permitted to start if the application requests a permission that is currently restricted and the application is not the one that imposed the permission restriction. Second, a third-party application's components are permitted to start if the application does not request any permissions that are currently restricted or if the permission is currently restricted and the application is the one that imposed the permission restriction. Furthermore, if a permission is dynamically restricted and a third-party application that is currently running has requested that permission and was not the application that imposed the permission restriction, the application will be forcibly closed.

Since DASF provides support for third-party applications to programmatically restrict permissions, the application, a_i, must define a set of permissions that it is permitted to restrict, $X_{a_i} := p_1, p_2, \ldots, p_m$, during the application's installation and approved by the user to prevent the abuse of dynamic permission restrictions. Thus, a third-party application, a_i, is allowed to restrict a permission, p, if $p \in X_{a_i}$.

Security Policies on Sensitive Data. A mobile device contains a set of data (e.g., strings, integers, arrays, etc.) in RAM, $\{d_1, d_2, \ldots, d_n\}$, each of which has its corresponding sensitivity level. In our framework, we use a privacy tag, $T(d_i)$, with the possible values of $\{t_1, t_2, \ldots, t_m\}$, to represent data d_i's sensitivity level. Thus, we can denote the data set as a set containing a privacy tag and the data: $D := \{\{d_1, T(d_1)\}, \{d_2, T(d_2)\}, \ldots, \{d_n, T(d_n)\}\}$.

It is possible for two pieces of data, d_j, d_k, where $1 \leq j \leq n$ and $1 \leq k \leq n$, with different privacy tags to be combined (e.g., concatenating two strings with different sensitivity levels). The general propagation rules of the privacy tags are as follows when data with two different privacy tags are combined, $d_j + d_k$.

$T(d_j) \geq T(d_k) \rightarrow T(d_j + d_k) = T(d_j)$
$T(d_j) < T(d_k) \rightarrow T(d_j + d_k) = T(d_k)$

These propagation rules ensure that when two pieces of data are combined, the highest sensitivity level is propagated to the combined data. This means that if the sensitivity level of d_j is greater than or equal to the sensitivity level of d_k, the combined data has the sensitivity level of d_j. Further, if the sensitivity level of d_j is less than the sensitivity level of d_k, the combined data has the sensitivity level of d_k.

We define two operations on the data that our security policy must handle. First, $W(d_j)$, denotes writing the data to flash memory (e.g., to a file or an SQLite database).

Second, $F(d_j)$ denotes forwarding the data from the device (e.g., through a socket, bluetooth socket, SMS, and MMS). Furthermore, data, d_j, that can not be forwarded from the device nor written to flash memory is denoted as $FW(d_j)$. The operations that are permitted on the data are determined by the data's privacy tag. The privacy tag, t_m, denotes that $T(d_j) = t_m \rightarrow \neg FW(d_j)$. The privacy tag, t_{m-1}, denotes $T(d_j) = t_{m-1} \neg W(d_j)$. Finally, the privacy tag, t_{m-2}, denotes $T(d_j) = t_{m-2} \rightarrow \neg F(d_j)$. Furthermore, we reserve the privacy tag, t_{m-3} to denote sensitive information that have no security policies enforced by the system.

To ensure that the security policies are enforced correctly when combining two pieces of data with different sensitivity levels, the propagation rules of the data's sensitivity levels must include an escalation of sensitivity levels. For example, if data with the privacy tag t_{m-1} is combined with data with the privacy tag t_{m-2}, the combined data should have the privacy tag t_m rather than t_{m-1}. Thus, the propagation of privacy tags are adjusted as follows when data is combined, $d_j + d_k$.

$T(d_j) = t_{m-1} \wedge T(d_k) = t_{m-2} \rightarrow T(d_j + d_k) = t_m$

This propagation rules ensures that if data that is restricted from being forwarded from the device is combined with data the is restricted from being written to flash memory, the combined data will be restricted from being written to the flash memory and forwarded from the device.

Application-Layer Message Protocol. DASF also includes an application-layer message protocol that allows a server to dynamically enforce privilege restrictions on the device and enforce the previously mentioned security policies on data that is sent to the device. The message protocol assumes that it is built over an SSL connection to ensure a secure communication channel between the client (e.g., mobile device) and the server.

The message protocol consists of applications sending a request to the server (e.g., requesting data) and the server responding back to the request (e.g., sending the data). The message protocol allows the server to impose a sensitivity level on the data in its response, and thus, impose a security policy on the data that it sends to the client. We call the server responding to the client's request for data as a *data message*. The header of the *data message* denotes the type of message, sensitivity level, and the length of the payload. The contents of a *data message* is shown in Fig. 2.

Fig. 2. Data Message

When an application requests data from the server, the server responds back to the client with $data\_message(\{sensitivity\_level, response\})$. The system extracts sensitivity level and applies that sensitivity level to the response before passing it to the application. The sensitivity level, s, corresponds to one of the four privacy tags that we previously defined. Thus, $s \in \{t_{m-3}, t_{m-2}, t_{m-1}, t_m\}$.

The server can also dynamically impose any privilege restrictions on the device without asking the user for permission. To prevent the abuse of server-imposed privilege restrictions, the application must be declared as a *medical application* during its installation and the system insures that the client application connects to the organization's trusted server. The server may dynamically impose privilege restrictions, or unrestricted previously imposed privilege restrictions, on the device by prepending a *privilege control message* to the *data message*. Furthermore, the payload of the *privilege control message* must be encrypted using a one-time key to prevent applications from abusing this functionality. The payload must be encrypted with a one-time key so that DASF can verify that the policy originated from the server and to prevent replay attacks. Our security framework fetches the initial secret key from the server when the device is first booted.

The payload of the *privilege control message* contains a list of privilege restrictions, privilege unrestrictions, and a new secret key. The entire payload is encrypted using the previous key, $E(key, payload)$. The header of the *privilege control message* contains the type of message, a flag denoting that a data message is appended, and the length of the payload. The contents of a *privilege control message* is shown in Fig. 3. When the system receives a *privilege control message*, it passes the payload to DASF. If the security framework successfully decrypts the payload, $D(key, payload)$, the set of permission restrictions and unrestrictions are imposed and the secret key is updated.

Fig. 3. Privilege Control Message

B. Prototype Implementation

We implemented the prototype of DASF on Android 4.1.1 (*Jelly Bean*) and TaintDroid. We have also implemented a server program that utilizes our message protocol to transfer data and dynamic privilege restrictions and unrestrictions to DASF and applications running on our prototype. We assume that DASF utilizes the CleanOS extension of TaintDroid, which provides a mechanism for revoking access to tainted data. However, our prototype is actually directly implemented over the TaintDroid platform because the source code for the CleanOS project was not available. Regardless of that fact, our security model is designed to be easily integrated into the CleanOS architecture to take advantage of the data revocation mechanism that it offers.

Specifically, the four sensitivity levels that our security model defines to enforce security policies on sensitive can can be defined as the highest sensitivity level of secure data objects on CleanOS. Further, CleanOS' propagation logic would need to be modified to handle the escalating of sensitivity levels that correlate to security policies when combining data with different sensitivity levels as mentioned in our security model.

Also, the organization that sends the sensitive data to the device would need to control CleanOS' cloud service that is used for revoking sensitive data. However, since we implement our prototype directly over TaintDroid, we emulate the sensitivity levels mentioned above by defining four custom privacy tags and enforce the system restrictions based off of those tags.

The architecture of the prototype of DASF consists of three modifications to the Android platform and TaintDroid. First, we defined the four privacy tags mentioned in the previous section and placed hooks in the Java libraries to enforce the security policies on sensitive data. Second, we implemented a system service that is responsible for managing the dynamic system-wide privilege restrictions and placed hooks in the Android framework libraries to enforce the privilege restrictions. Third, we implemented the message protocol in the system to tag sensitive data that is received from the server and modified the Java libraries to force *medical applications* to use our message protocol.

Restriction Policy Manifest File. For applications to utilize DASFk, the application must include a special manifest file that is parsed during the application's installation. As stated in our security model, DASF allows applications to dynamically enforce system-wide permission restrictions that may be programmatically set in an application's code or by receiving messages from the trusted server using DASF's message protocol. In order to prevent malicious applications from abusing the system-wide permission restrictions that are imposed by the server, DASF limits their use to applications that are declared as a *medical applications* in their *restriction_policy.xml* file. Furthermore, we force all applications that are declared as *medical applications* to use our message protocol and communicate directly with the trusted server. All system-wide permission restrictions that are set programmatically in an application must be declared at the time of the application's installation and approved by the user to prevent application programmers from maliciously imposing restrictions on the device. However, the trusted server can dynamically restrict any permission on the device at runtime without the user's permission to allow organizations to dictate the security policies on the device on-the-fly.

The *restriction_policy.xml* manifest file is an XML file that must be included in the */assets/* directory in the application's APK in order for an application to utilize DASF. The *restriction_policy.xml* file is responsible for declaring the system-wide permission restrictions that may be set programmatically by the application and declaring the application as a *medical application* to allow for system-wide permission restrictions imposed by the server. An example *restriction_policy.xml* file that allows the trusted server to dynamically restrict any permissions on the device and allows the application to programmatically restrict the access to the microphone and camera is shown in Fig. 4.

Dynamic Privilege Restrictions Implementation. We implemented a system service, named the *Privilege Restriction Service*, that is responsible for enforcing and imposing the dynamic system-wide privilege restrictions on applications.

```
<?xml version="1.0" encoding="UTF-8"?>
<restriction-policy>
  <application medical-application="true"/>
  <restrict-permission permission="android.permission.INTERNET"/>
  <restrict-permission permission="android.permission.RECORD_AUDIO"/>
</restriction-policy>
```

Fig. 4. An example of a policy restriction file.

The *Privilege Restriction Service* contains four hash maps to store necessary information that DASF needs to enforce privilege restrictions on applications. The four hash maps store the following information.

- *Medical application hash map:* Stores packages that are tagged as medical applications.
- *Requested permissions hash map:* Stores all of the *uses-permissions* that an application requests.
- *Restriction policy hash map:* Stores the privileges that an application can programmatically restrict.
- *Restricted privileges hash map:* Stores the system-wide permission restrictions currently imposed on applications.

We decided to store the package information in RAM rather than on flash memory in order to prevent unauthorized modification of our framework's data (e.g., pulling a file via a USB connection, changing the contents and then pushing it back to the device). Furthermore, DASF can repopulate the hash maps if the device is shutdown since Android reparses all of the installed packages during the boot process, which makes main memory the ideal location to store our framework's data. Moreover, the entries that are enforced by the server in the *restriction policy hash map* can be restored to its previous state since our *Privilege Restriction Service* communicates with the trusted server during the boot process. The *Privilege Restriction Service* is started by Android's *Service Manager* when the device boots. When the *Privilege Restriction service* starts, it communicates with the server to fetch the secret key needed to decrypt security policies received from the server.

We placed a hook in Android's *Package Parser* to parse the *restriction_policy.xml* manifest file. If the application is being installed, the user is notified if the APK's restriction policy file includes the *medical application* tag and the restricted permissions policy that it declares. For the application to continue installation, the user must approve the policies that are declared in the restriction policy file. If the user approves the policies, a hook in the *Package Manager Service* invokes the *Privilege Restriction Service* to add the *uses-permissions* that an application requests in its manifest file to the *requested permissions hash map*. Furthermore, the hook in the *Package Manager Service* also invokes the *Privilege Restriction Service* to populate the *medical applications hash map* and the *restriction policy hash map* if the application included the *restriction_policy.xml* manifest file. We have also restricted applications that are declared as *medical applications* from sharing a UID to ensure that they run in their own isolated process and are the only application that may access their application's directory and files. Medical applications must have a unique UID because a developer that creates a medical

application could maliciously create a different application signed with the same developer signature to access the sensitive files of the medical application.

We have provided an *Privilege Restriction* class in the *android.* package that allows applications to programmatically enforce a privilege restriction or unrestrict a previously restricted permission. The application can call the *restrictPermission()* method with the permission name as the argument of the method to restrict a permission. The *restrictPermission()* method invokes the *Privilege Restriction Service* to check whether the application that called the method has requested to restricted the permission by checking the *restriction policy hash map*. If the permission is in the *restriction policy hash map* under the calling application's entry, the permission restriction is imposed by adding an entry to the *restricted privileges hash map* and all currently running applications that request that permission are closed. Furthermore, an application may unrestrict a previously imposed permission restriction by calling the *unrestrictPermission()* method with the permission name as the argument of the method. The *unrestrictPermission()* method invokes the *Privilege Restriction Service* to ensure that the application attempting the unrestrict the permission was the application that imposed the restriction. If the application's UID matches the UID of the application that imposed the restriction, the permission is unrestricted by removing the entry from the *restricted permissions hash map*.

We have placed hooks in the *Activity Manager Service* so when an application's component (e.g., activities, services, broadcast receivers, or content providers) is attempting to start, our *Privilege Restriction Service* is called to enforce the security policies for starting applications as defined in our security model.

Sensitive Data Security Policies Implementation. Our security framework allows an application developer to programmatically classify the sensitivity level of data and also allows a server to classify the sensitivity level of data that it sends to the device. An application developer can programmatically classify data by calling methods in the *Data Classification* class that DASF provides. For example, an application developer can prevent a string containing PHI data from being written to the disk by calling the *preventDiskWrite()* method with the string as the argument. The *Data Classification* class also provides the following methods, *preventForwarding()* and *preventDiskWriteAndForwarding()* to enforce the security policies on data as discussed in our security model. These methods add the security policy's corresponding privacy tag to the data.

We have placed hooks in the Android system to enforce the security policies imposed on the sensitive data. To prevent data from being written to flash memory, we have placed a hook in the *Posix* class to enforce restrictions on data that is written to a file and the *SQLite* library to enforce restrictions on data that is written to an SQLite database through insert and update statements. These hooks check the privacy tag of the data attempted to be written to flash memory and blocks writing the data if the privacy tag matches the tags that prevent disk writes. We assume that the CleanOS SQLite patch for retaining privacy tags on retrievals from the database is implemented to ensure correct propagation of the privacy tags on individual entries. To prevent data from being forwarded from the device, we have placed hooks in the *Posix* class, the *BluetoothSocket* class, and the *SmsManager* class to enforce whether PHI data is permitted to be sent over a socket, a bluetooth connection, or through an SMS message. These hooks check the privacy tag of the data attempted to be forwarded from the device and blocks forwarding the data if the privacy tag matches the tags that prevent forwarding data.

Message Protocol Implementation. As stated in our security model, our message protocol consists of two different types of messages. First, the *data message* is used by the server to send data and the data's corresponding sensitivity level to a medical application. Second, the server may prepend a *permission control message* to a *data message* to impose system-level permission restrictions, or unrestrictions, on the device.

Whenever a medical application requests data from the server the server sends the requested data back to the application as the payload of a *data message*. The *data message* contains a 4-byte header and a payload. The header of the *data message* contains 1-bit to identify the type of message, 2-bits to identify the sensitivity level that the server imposes on the payload, and 29-bits for the length of the payload. Thus, the server can transmit 2^{29} bytes of data in one *data message*. However, if the server needs to transmit more than 2^{29} bytes of data, it can break the response up into multiple *data messages*.

Furthermore, the server may also prepend a *permission control message* to the *data message*. The *permission control message* contains a 2-byte header. The header of the *permission control message* contains 1-bit to identify the type of the message, 1-bit to denote that a *data message* is appended, and 14-bits for the length of the payload. Therefore, the server can transmit 2^{14} bytes of permission restrictions and permission unrestrictions in one *permission control message*.

To force *medical applications* to communicate with the trusted server, we placed a hook in the *Posix* class to invoke our *Privilege Restriction Service* to determine whether the application is a *medical application* by checking whether or not the application's UID is in the *medical application hash map*. If the application attempting to open a socket is a *medical application*, we force the connection to the server by changing the IP address to the server's IP address. Again, we assume that the server will reject the connection if communication is not attempted over an SSL connection.

To ensure that *medical applications* use our protocol, which is transparent to the actual application, we have implemented the protocol over an *InputStream* called the *MessageInputStream*. The *MessageInputStream* is returned when an application calls the *getInputStream()* method instead of the default input stream.

Whenever an application receives a message from the server, the *MessageInputStream* first checks whether the message contains a *permission control message*. If the message has

a *permission control message* prepended to it, the *MedicalInputStream* parses the header and continues reading the length of the *permission control message's* payload. After it has received the entire payload, it passes the encrypted payload to the *Privilege Restriction Service*. The *Privilege Restriction Service* decrypts the payload, enforces the requested permission restrictions as defined in our security model, enforces the requested permission unrestrictions, and then updates the secret key. If the *data appended flag* was set in the *permission control message*, the *MessageInputStream* reads the header of the *data message*, extracts the size of the payload, and the sensitivity level of the payload. Once the *MessageInputStream* obtains this information, it reads in the requested number of bytes, adds the corresponding sensitivity level as the data's privacy tag, and returns the data back to the application that requested the data. This process is shown in Fig. 5.

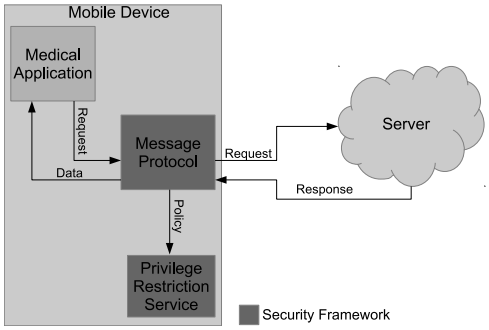


Fig. 5. Message Protocol Flow

V. Evaluation

We tested our prototype of DASF on a Galaxy Nexus (*GT-I9250*), which includes a 1.2 GHz dual-core ARM Cortex-A9 processor, 1 GB of RAM, and 16 GB of flash memory. Our goal is to illustrate that DASF provides adequate protection against the previously discussed security threats and provides a reasonable performance overhead.

A. Security Threat Evaluation

As previously discussed, we consider four different types of security threats that DASF must provide protections against *(1) legitimate user misbehavior*, *(2) malicious users*, *(3) application misbehavior*, and *(4) malicious applications*. Since CleanOS addresses the 2nd security threat by providing a mechanism for revoking sensitive data and we assumed that our model was built on top of CleanOS, we do not evaluate that security threat. However, we perform the following three tests to show that DASF adequately addresses the remaining security threats.

- *Legitimate user misbehavior.* We created an application that receives data from the server that has restrictions imposed on the data. Next, we attempt to manually save and forward the data by passing it to other applications on the device that attempt to perform these operations.
- *Application misbehavior.* We created an application that requests data from the server (e.g., an MRI image) that is restricted to be saved to the device. When the application is receiving the data, it attempts to break the policy by writing the data directly to a file when receiving it.
- *Malicious applications.* We created a malicious voice memo recorder application that attempts to steal information from the environment. Our voice memo recorder requests the *record audio* permission and the *internet* permission to allow users to sync memos to their other devices. However, the voice memo recorder maliciously launches a background service to secretly record audio in the background and transmits the recorded audio to an external server.

We ran the first test and confirmed that DASF successfully blocked the data from being forwarded from the device and saved to the device. Thus, DASF enforced the policy imposed on the data from the server.

When we ran the second test, we confirmed that DASF successfully blocked the data from being saved to a file. Moreover, we modified the application to check the sensitivity level on the data received and determine whether to store the data in a file or keep it in main memory.

Finally, we ran the third test by starting the malicious voice memo recorder and confirmed that it successfully recorded data in the background. Next, we created another application that restricts the *record audio* permission when the user presses a button. DASF successfully closed the malicious voice memo recorder, including its background service, and prevented it from being opened again until the *record audio* permission was unrestricted. Instead of restricting the permission by pressing a button, an application can be designed to restrict permissions when certain RFID tags are scanned, which would allow doctors to simply wave their device by an RFID tag when entering areas where conversations may include PHI data (e.g., outside of patient rooms). To prevent malicious applications from attempting to steal sensitive information stored on the device (e.g., PHI information authorized to be saved to the device), we rely on Android's file system privileges (e.g., UIDs) and our restriction that medical applications are assigned a unique UID.

B. Performance Evaluation

Since DASF is invoked whenever an application, or application component is started, we tested the performance overhead on the starting activities and services on the Galaxy Nexus running our prototype. To get a baseline for comparison, we repeated the tests on the stock Android *Jelly Bean* platform and on TaintDroid. Furthermore, we also tested the overhead of *medical applications* using our message protocol.

Application Component Overhead. Whenever an activity or service is started, our*Privilege Restriction Service* is invoked to determine whether the activity or service may be started. Therefore, we tested the time that it takes to launch activities and service. We repeated the same tests on the stock Android *Jelly Bean* platform and on TaintDroid.

Fig. 6 shows the performance overhead of starting activities and services on our prototype, the stock Android *Jelly Bean*

platform, and on TaintDroid. As the figure shows, our prototype took 118 milliseconds on average to start an activity and 17 milliseconds on average to start a service. The stock Android *Jelly Bean* platform took 94 milliseconds on average to start an activity and 8 milliseconds on average to start a service. The TaintDroid platform took 115 milliseconds on average to start an activity and 14 milliseconds on average to start a service. We assume that the overhead when comparing TaintDroid and the stock Android *Jelly Bean* platform is due to the extra memory that needs to be allocated for the taint tags in the Java stack. When compared to the overhead of TaintDroid, using our prototype results in a minimal overhead on starting activities and services.

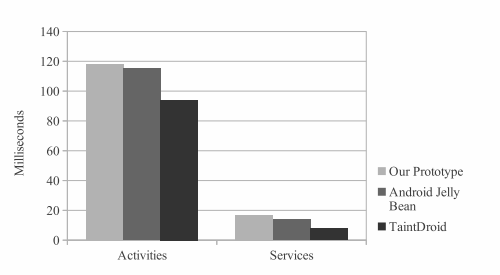


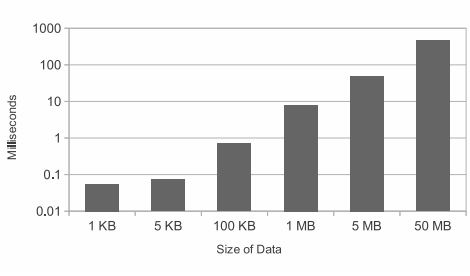
Fig. 6. Performance overhead when starting activities and services

Message Protocol Overhead. To test the overhead of our message protocol, we created an application that requested data of varying length from the server. The data lengths that we tested are 1-KB, 5-KB, 100-KB, 1-MB, 5-MB, and 50-MB. We have recorded the time that it took the *MessageInputStream* to receive the message and pass the data back to the application. We disregarded the time that it took to transmit the data over the network since that is related to the overhead of the network speed rather than the overhead caused by our message protocol. Furthermore, we also recorded the time that it took the system to enforce policy restrictions and unrestrictions.

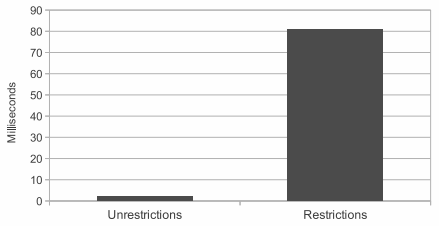
As the Fig. 7 suggests, our message protocol took 54931 nanoseconds on average to parse and return 1-KB of data back to the application, 73242 nanoseconds on average for 5-KB, 720215 nanoseconds on average for 100-KB, 8 milliseconds on average for 1-MB, 49 milliseconds on average for 5-MB, and 468 milliseconds on average for 50-MB. Furthermore, it took 81 milliseconds on average to impose permission restrictions and around 2 milliseconds on average for permission unrestrictions.

C. Limitations

There are three current limitations of the DASF prototype. First, the sensitivity level of the data can be cleared since TaintDroid's taint propagation logic does not address implicit flows. This is a major concern because applications can clear the sensitivity level from data and break the security policies imposed on that data. Second, TaintDroid also does not allow applications to use third-party native libraries since they can be used to clear privacy tags. Third, DASF does not currently handle propagating the sensitivity level of data displayed on the screen. Therefore, a user can break the security policy on sensitive data by taking a screenshot of an image.



(a) Data Message Overhead



(b) Privilege Control Message Overhead

Fig. 7. Message Protocol Performance Overhead

VI. Related Work

It is believed that 1.4 billion smartphones will be in active by the end of 2013 [21], and 57% of them will run the Android operating system. The popularity of the Android devices has made it the top target of the mobile malware [13], [6], [23] and other potential attacks [11], [17]. To improve the smartphone security defense, a number of Android system extensions [19], [7], [24], [15], [20], [10], [16], [3] have been proposed. Apex [19] proposes an Android enforcement frame that enables flexible permission granting and resource restrictions. MockDroid [7] is closely related to this work in that the Android system is modified to protect the privacy. The main difference, however, is that MockDroid protects the privacy by adding perturbations to achieve ambiguity. Our framework offers a system-wide, dynamic and strict access restrictions to protect the sensitive data. In addition, our framework is 100% transparent to general Android applications. TISSA [24] protects the private information leakage by an extra permission layer. AppFence [15] provides a similar fine-grained permission control. Different from their schemes, our security framework is more versatile, protecting both outgoing and incoming data through an on-demand dynamic security provisioning. Saint [20] uses the design-time security policies to manage permissions. Kirin [10] proposes a set of security rules to mitigate malware. L4Android [16] and Cells [3] provide the improved OS isolation for security.

TaintDroid [9] provides a information flow monitoring system to detect the potential privacy leakage. This idea is adopted in our system framework to enforce the data access privilege control and propagation restrictions. Static

analysis has been used to detect privacy leak in Android applications [5] and potential security vulnerabilities [18]. Crowdroid [8] analyzes the system calls dynamically to detect the malware. Android permission specification has been thoroughly studied in Stowaway [12] and PScout [4]. As discussed previously, the permission based Android security system is insufficient to address the security challenges in the application compliant with the HIPAA rule.

VII. Conclusion

This paper describes a distributed security framework (DASF) for Android-based mobile operating systems designed to provide dynamic privilege restrictions on applications and security policies on sensitive data sent to the device. Unlike Android's current security model, DASF allows an application's permissions to be restricted dynamically to address the security concern of mobile devices being used in sensitive environments. Further, DASF also allows the organization to remain in control of the sensitive data that it sends to the mobile device by imposing security policies on that data. We implemented a prototype of DASF by creating a system service to enforce privilege restrictions on applications, a custom *InputStream* to handle our message protocol, and by placing hooks in the Android system to enforce the security policies. Further, we built a server module to send data and security policies to the device to demonstrate that the security policies can be dynamically imposed on both an application's privileges and on sensitive data. Our experiments shows that DASF addresses the limitations of Android's current security model while imposing an reasonable performance overhead.

References

[1] Hipaa privacy rule. *U.S. Department of Health and Human Services*, 2002.

[2] Smartphones, tablets and mobile marketing. *Manhattan Research*, 2012.

[3] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 173–187, New York, NY, USA, 2011. ACM.

[4] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[5] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. *2012 7th International Conference on Malicious and Unwanted Software*, 0:66–72, 2011.

[6] Michael Becher, Felix C. Freiling, Johannes Hoffmann, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 96–111, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 49–54, New York, NY, USA, 2011. ACM.

[8] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[9] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol 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*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[10] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[11] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: an analysis of android ssl (in)security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[12] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[13] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David 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*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[14] Jessica Grogan. Smartphones partly to blame for hipaa compliance issue. *MDNews.com*, Jan 2012.

[15] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[16] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 39–50, New York, NY, USA, 2011. ACM.

[17] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[18] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[19] Mohammad Nauman, Sohail Khan, and Xinwen 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*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[20] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[21] ABI Research. http://venturebeat.com/2013/02/06/800-million-android-smartphones-300-million-iphones-in-active-use-by-december-2013-study-says/. 2013.

[22] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 77–91, Berkeley, CA, USA, 2012. USENIX Association.

[23] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

[24] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on android). In

Proceedings of the 4th international conference on Trust and trustworthy computing, TRUST'11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag.