A Distributed Android Security Framework

Benjamin E. Andow and Haodong Wang
Computer and Information Science Department
Cleveland State University
Cleveland, OH 44115, USA

***Abstract*—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 enables the system-wide privilege restrictions to be enforced on untrustworthy applications and sensitive data. Ultimately, DASF allows 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 as 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 other 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 machine, yet a last generation wireless communication technology. Second, even though as many as 81% healthcare professionals 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) systems.

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.
- *Sensitive data flow confinement:* The service administrator should be able to control the propagation of the sensitive data and, if necessary, revoke the access to protected data that is sent to the mobile devices. For example, the healthcare organization should prevent PHI from being saved to the devices (e.g., saving a medical record as a file) or being forwarded to another device.
- *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 are utilized. For example, the service administrator may want to dynamically enforce system-wide access control policies on the mobile devices to allow the organizations to dictate their own security policies.

This paper uses the healthcare communication application as a platform to demonstrate the design and development of a Distributed Android Security Framework (DASF), a customized 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. The coupling between the Android and the server is achieved by using secure heart-beat messages through a system security channel established during the installation time. In addition, 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 needs when dealing with sensitive information. In mobile devices that install DASF, the organization ultimately controls the security policies that the system enforces on the users device and on the sensitive data that applications receive.

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 variety of security attacks while only imposing a reasonable system overhead.

II. Attack Model and Security Assumptions

The mobile devices are not trusted. Once the mobile device is lost or stolen, the adversaries can capture stored data 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. Security Framework 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 $S_A \cap T_A = \{\}$.

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:

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

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

B. 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:

$T(d_j) \geq T(d_k) \rightarrow T(d_j \wedge d_k) = T(d_j)$,
$T(d_j) < T(d_k) \rightarrow T(d_j \wedge d_k) = T(d_k)$.

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 either be forwarded from the device or 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_0, denotes that $T(d_j) = t_0 \rightarrow \neg FW\,(d_j)$. The privacy tag, t_1, denotes $T(d_j) = t_1 \neg W\,(d_j)$. Finally, the privacy tag, t_2, denotes $T(d_j) = t_2 \rightarrow \neg F\,(d_j)$. Furthermore, we reserve the privacy tag, t_m, to denote sensitive information that have no security policies enforced by the system. The value of m is a system parameter and can be adjusted depending on the requirement of the security levels.

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_1 is combined with data with the privacy tag t_2, the combined data should have the privacy tag t_0 rather than t_1 or t_2. Thus, the propagation of privacy tags are adjusted as follows when data is combined:

$T\,(d_j) = t_1 \wedge T\,(d_k) = t_2 \rightarrow T\,(d_j + d_k) = t_0$.

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.

C. 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. 1.

Fig. 1. 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. For example, $s \in \{t_m, ..., t_2, t_1, t_0\}$.

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 above assurance is achieved by a server-side periodic encrypted heartbeat message that confirms the application's aliveness. Note that under our security framework, the connection between the server and the *medical application* is supposed to be persistent; the tear-down of the connection will lead to the termination of the application due to the security concern. The system secret key update and the privilege control message (as described below) can be arranged as a piggyback in the heartbeat messages.

The dynamic security provisioning is enforced by the *privilege control message*, which 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. The one-time key can be provided by a server generated one-way key chain, similar to S/Key mechanism [15]. Our security framework fetches the initial master secret key from the server when the device is first booted and updates the master key, if necessary, through the server heartbeat message as described above. 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. 2. 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.

IV. Prototype Implementation

We implemented the prototype of DASF on Android 4.1.1 (*Jelly Bean*) and TaintDroid. We have also implemented a

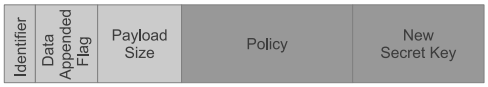


Fig. 2. Privilege Control Message

```
<?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. 3. An example of a policy restriction file.

server program that utilizes our message protocol to transfer data and dynamic privilege restrictions and unrestrictions to DASF and applications running on our prototype.

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

A. Restriction Policy Manifest File

For applications to utilize DASF, 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. 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. 3.

B. Dynamic Privilege Restrictions Implementation

We implement a system service, named as Privilege Restriction Service (PRS), which is responsible for enforcing and imposing the dynamic system-wide privilege restrictions on applications. PRS 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 decide to store the package information in RAM rather than on flash memory to prevent unauthorized modification of our framework data (e.g., pulling a file via a USB connection, or 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 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 PRS communicates with the trusted server during the boot process. PRS is started by the *Service Manager* in Android during the booting. When PRS starts, it communicates with the server to fetch the secret key needed to decrypt security policies received from the server.

We place a hook in Android's *Package Parser* to parse the *restriction_policy.xml* manifest file. When 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. Once the policies are approved, a hook in the *Package Manager Service* invokes PRS 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 PRS to populate the *medical applications hash map* and the *restriction policy hash map* if the application included the *restriction_policy.xml* manifest file. We also restrict 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 provide a *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 PRS to check whether the application that calls the method has requested to the restricted 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 PRS to ensure that the application attempting to unrestrict the permission is the one that imposes 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 place 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 PRS is called to enforce the security policies for starting applications as defined in our security model.

C. Sensitive Data Security Policies Implementation

DASF 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 corresponding privacy tag to the data.

To prevent data from being written to flash memory, we place 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 the operation if the privacy tag matches the tags that prevent disk writes. To prevent data from being forwarded from the device, we place 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.

D. 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 its corresponding sensitivity level to a medical application. Second, the server may prepend a *permission control message* to a *data message* to impose the system-level permission restrictions, or unrestrictions, on the device.

Whenever a medical application requests data from the server, the server replies the application with the payload of a *data message*. The *data message* contains a 4-byte header and a payload. The header 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. Messages larger than 2^{29} bytes can be fragmented and transmitted in 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, which 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 place a hook in the *Posix* class to invoke PRS to determine whether the application is a *medical application* by checking whether or not the its UID is in the *medical application hash map*. If it is true, we force the connection to the server through a tunnel that builds on either SSL or IPsec.

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*. PRS 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. 4.

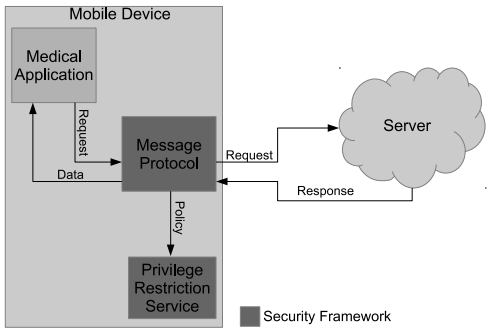


Fig. 4. Message Protocol Flow

V. Evaluation

We test 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 validate DASF that provides adequate protection against the previously discussed security threats and provide a performance evaluation for the system overhead.

A. Security Evaluation

As previously discussed, we consider the four different types of security threats that DASF must provide protections against. Due to the space constrain, we present our study of the security evaluation by using the following carefully selected attacking examples which emulate the most adverse environment.

- **Legitimate user misbehavior.** We create 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.
- **Malicious users.** Pretending to be a malicious user, we launch a device stealing attack to obtain the stored PHI.
- **Application misbehavior.** We create 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.
- **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.

In the first test, a scenario was created that a user was viewing an image with the write-to-flash permission disabled, transmitted from the server. Then the user tried to save the image (e.g., by long pressing on the image to have the option to save it to the SD card) to the local flash storage. This attempt was defeated because the image's privacy tag checking raised a red flag and then the flash memory writing is blocked.

The second test emulated a device stealing scenario where the lost is reported. Given the security features deployed in DASF, it is the attacker's best interest to immediately disconnect the device from Internet (to evade the launching of the system counter-attack mechanism, e.g., remote wipe) with an attempt to compromise the PHI stored in a previous session. This attacking attempt, however, was defeated since the system forced the termination of the *medical application* after a specific number of consecutive server heartbeats (e.g., 10) were missed, which in turn released the memory containing the PHI.

In the third test, we developed a misbehaving application that silently checks the received data sensitivities and stores the highly sensitive data to a file (without the proper authorization). Again, this attempt is defeated by the privacy tag verification place in DASF.

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. We ran the last 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. Actually, an easy extension can be designed to use RFID tags, instead of pressing a button, to restrict the permission. So doctors can simply wave their devices when entering areas where conversations may include PHI data (e.g., outside of patient rooms).

B. Performance Evaluation

Since DASF is invoked whenever an application, or application component is started, we test 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 services. We ran the tests for 10 times and record the

average value. The same procedures were repeated on the stock Android *Jelly Bean* platform and on TaintDroid.

Fig. 5 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 takes 118 milliseconds on average to start an activity and 17 milliseconds on average to start a service. The stock Android *Jelly Bean* platform takes 94 milliseconds on average to start an activity and 8 milliseconds on average to start a service. The TaintDroid platform takes 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 privacy tags in the Java stack. Compared to the overhead of TaintDroid, our prototype results in a minimal overhead on starting activities and services.

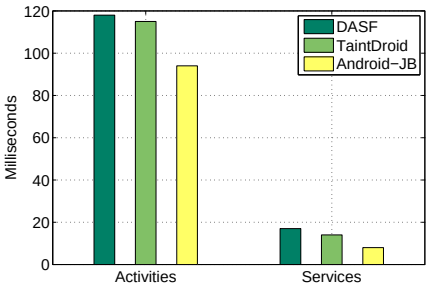


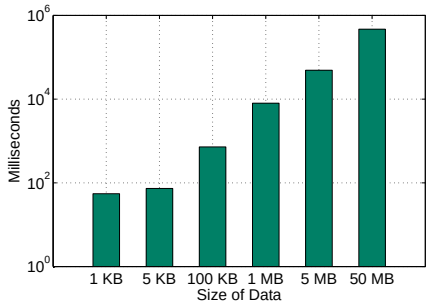
Fig. 5. Performance overhead when starting activities and services

Message Protocol Overhead. To test the overhead of our message protocol, we create an application that requests data of varying lengths from the server. The data lengths are 1-KB, 5-KB, 100-KB, 1-MB, 5-MB, and 50-MB. We record the time that it takes the *MessageInputStream* to receive the message and pass the data back to the application. We disregard the time that it takes 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.

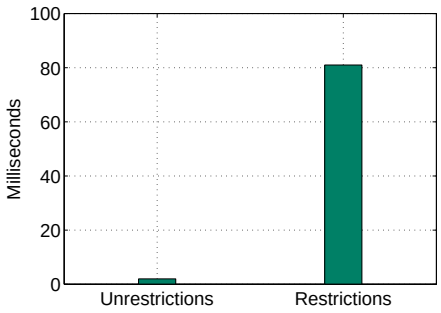
Furthermore, we also record the time that it takes the system to enforce policy *restrictions* and *unrestrictions*. As the Fig. 6 suggests, our message protocol spends 54.9 microseconds on average to parse and returns 1-KB of data back to the application, 73.2 microseconds on average for 5-KB, 720.2 microseconds 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 takes 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 collected by the applications can be cleared since TaintDroid's propagation logic does not address implicit flows. As the result, the



(a) Data Message Overhead



(b) Privilege Control Message Overhead

Fig. 6. Message Protocol Performance Overhead

applications may break the security policies imposed on that data. An immediate solution would introduce the application authentication procedure to certify the authenticity of the applications. A permanent fix is to get rid of TaintDroid and address the problem in the native Android. Second, DASF does not allow applications to use third-party native libraries (due to TaintDroid's restrictions) since they can be used to clear privacy tags. Our future work will address the above two limitations by directly building our DASF on the native Android. Third, DASF does not currently handle propagating the sensitivity level of data displayed on the screen. Therefore, a user can take a picture, from a separate camera, on the device screen and, as the result, acquires the picture that is unauthorized. At this moment, we have no solution to address this type of privacy breaches but argue that the amount of the information leakage is limited because the quality of the image captured in the above way (e.g., the number pixels per inch) has no comparison to the original quality.

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], [22] and other potential attacks [11], [18]. To improve the smartphone security defense, a number of Android system extensions [19], [7], [23], [16], [20], [10], [17], [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 [23] protects the private information leakage by an extra permission layer. AppFence [16] 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 [17] 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 and Future Work

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 concerns 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 implement a prototype of DASF by creating a system service to enforce privilege restrictions on applications, and our experiments show that DASF addresses the limitations of Android's current security model while imposing a reasonable performance overhead. Our future work includes the system enhancement in the following aspects. First, we need to develop a mechanism to differentiate the network brokeage and security attacks. Ideally, our DASF will allow a short disconnect operation without sacrificing the security countermeasure capabilities. Second, we will start our development of DASF on a native Android system instead of the TaintDroid to further improve the system latency performance and enhance the security performance as discussed previously.

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 *SOSP*, 2011.

[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*, 2012.

[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. *7th International Conference on Malicious and Unwanted Software*, 0, 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 IEEE Symposium on Security and Privacy*, 2011.

[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*, 2011.

[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*, 2011.

[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 *OSDI*, 2010.

[10] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *ACM CCS*, 2009.

[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*, 2012.

[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*, 2011.

[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*, 2011.

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

[15] Neil M. Haller. The S/KEY One-time Password System. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 1994.

[16] 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 *ACM CCS*, 2011.

[17] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android: a generic operating system framework for secure smartphones. In *Proceedings of the ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.

[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*, 2012.

[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*, 2010.

[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*, 2009.

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

[23] 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*, 2011.