StoreDroid: Sensor-Based Data Protection Framework for Android

Miriam Allalouf
Azrieli College of Engineering (JCE)
Jerusalem, Israel
miriam.allalouf@gmail.com

Radel Ben-Av
Azrieli College of Engineering (JCE)
Jerusalem, Israel
rbenav@gmail.com

Alex Gerdov
Azrieli College of Engineering (JCE)
Jerusalem, Israel
alexgerdov@gmail.com

Abstract—Android has become the most prevalent smartphone operating system. Despite its popularity, Android has a lot of flaws in security. In this research study we target a wide range of smartphone applications that share secret and local data with the service provider so that this data will not be leaked or accessed by other entities. The StoreDroid framework, described in this paper, addresses possible data violations that can occur in the current Android system by adding protection mechanisms in several layers as follows: (1) at the Linux level we use the security-enhanced Linux and security-enhanced Android plug-ins that prevent today's privileged escalation data access; (2) StoreDroidApp is a generic sensor-based access control mechanism where the sensors (such as biometric sensors and GPS) and the rules to access the data are defined by the service provider for a better protection — we took advantage of the fact that Android systems are usually integrated with various hardware sensors in order to protect the user as well as the service provider; and (3) secured message passing protocol to ensure that sensitive data will not be compromised by unwanted applications. The StoreDroid framework makes the following contributions: (1) the generic StoreDroidApp stub that is installed when the ROM is built narrows possible illegal data access for assigned application by the set of semantic and limiting sensor-based access rules, and (2) on top of the regular Linux used in Android, the customized security-enhanced Linux ensures that the sensor-based application will keep the data isolated and secured.


I. INTRODUCTION

A smartphone is a widely used personal and portable device built on a mobile computing platform. Typically its hardware includes a high-speed (multicore) CPU, high-resolution touch screens, high-speed data access via Wi-Fi and mobile broadband, a GPS receiver, and additional sensors. Its software includes modern OS (iOS/Android/Win8) with integrated support for all the hardware, with many standard built-in components—for example, web browsers, social network applications, and much more. In addition one can acquire additional applications and easily install them on the device. As smartphone adoption rises, these devices have increasingly become subject to attacks by malicious software (malware) [1], [2]. Frequently this malware is distributed through application stores that have minimal or no review process for their content [3]. In some cases the ability to acquire software directly from links on the web, where users are directed to click on links, such as ads that look legitimate, which then open in the device’s web browser, cause malware to be downloaded and installed automatically. In other cases, rootkit malware enables continued privileged access on the device to violate privacy or cause other damage.

Android has become the most successful smartphone operating system. As its popularity rose, so did the number of threats that target it and try to monetize by taking advantage of its security flaws. Android’s access control system is based on application sand boxing combined with Linux's application security. In addition it provides another level of security based on user-controlled permission authorization. As such it provides a decent initial level of security. Yet there are flaws in that design, mainly causing a false authentication of the owner, one of which stands out above all others: privilege escalation. Applications that gain access to the root user are able to easily bypass all of Android's defense mechanisms and easily gain access to protected information. Thus, while smartphones are personal and portable devices, they are not secure enough to protect data and allow reliable access to remote sites.

Our work, as described in this paper, demonstrates that it is possible to keep private and sensitive information on a modern Android device, despite the many privilege escalation problems that plague the Android platform. To accomplish this, we have created StoreDroid, a generic framework achieving a better data isolation by combining several layers of protection. We consider a case of sensitive and local data on the smartphone that is shared between a remote server and a local client. The framework provides a new sensor-based access control where a file access is based on a variety of sensors that are built into the Android such as biometric sensors or GPS, on a customized version of Linux kernel level security SELinux (which stands for Security Enhanced Linux), and on a secure network protocol between a remote server and a local client, ensuring that it will not be compromised by unwanted applications and will be used only in a controlled manner by eligible owners. The framework contains generic StoreDroidApp stubs that are installed when the ROM is built and can be used by highly sensitive applications for a variety of use cases. We demonstrated the framework with a sample of a StoreDroidApp stub for the case of an employee-employer time-reporting application.
II. BACKGROUND AND RELATED WORK

Android is an application execution environment for mobile devices that includes an operating system, application framework, and libraries. The Linux kernel is the base of the architecture of Android. The core libraries layer (on top of the kernel) includes over ninety open-source systems implemented by Android, such as OpenSSL, WebKit (a web browser), SQLite (database engine), OpenBinder (a typesafe IPC mechanism), and more. An important core library is the Dalvik Virtual Machine, which is an optimized version of the Java VM. It runs .dex (Dalvik Executable) files, which are more compact and memory efficient than Java class files. Its purpose is to efficiently execute Android applications. The application framework layer consists of Java-based interfaces used by Android applications, with which the user interacts.

Android’s application security relies on a secure IPC implementation called Binder and traditional Linux security concepts. Android applications state their required permissions at install time, and those permissions are checked when an application wants to perform sensitive actions such as send SMS messages or make phone calls. Android applications are sandboxed one from another; each application is assigned its own Linux user id at install time. Consequently, the code of two different applications runs in two different processes. In a way this creates a sandbox that provides an isolated environment for applications [4]. File permissions in Android are based on traditional Linux permissions (three sets of “rwx” for owner, group, and other). Generally, system files in Android are owned by either the “system” or the “root” user, and application files are owned by an application-specific user.

As we have seen with Windows malware in the past, malware creators are finding new ways to monetize malware [5] and use it to generate revenue. In parallel, there has been a rise in malware creation kits that simplifies the process of inserting malicious code into legitimate Android applications [6], making the creation of android malware accessible for less savvy people. Recently, more malware uses privilege escalation to escape the application sandbox altogether by gaining access to the root user. Of the over 1260 samples collected by Zhou et al. [7] 463 (36.7%) embed at least one root exploit. It is not uncommon for malware to have two or more root exploits that maximize its chances of successful exploitation on multiple platform versions. A further investigation of how these exploits are actually used shows that earlier malware in many cases simply copied verbatim the publicly available root exploits without any modification, even without removing the original debug output strings or changing the file names of associated root exploits. Recent samples encrypt the root exploit code, making it much harder to detect [8].

Another rising concern is the use of the MasterKey vulnerability that allows attackers to make changes to the application’s code without affecting the cryptographic signature used to check the legitimacy of an application. In Q3 of 2013, 101 unique samples of malware were sighted that use the MasterKey exploit [9]. Google was reportedly notified of this issue earlier in the year, and appropriate fixes were applied to AOSP and sent to the partners. This brings us to the main problem: because of inherent shortcomings in the Android’s security design, keeping private, secure, and sensitive data on modern Android phones is a challenge.

Many studies and methods have been purposed to straighten Android’s security. The Kirin system described in [10] is an extension of Android’s application installer. It checks the permissions requested by applications at the time of installation, denying an application’s installation if the permissions requested by the application encompass a set of permissions that breach a given system-centric policy. Kirin fails to address the issue of privilege escalation, since once the malicious application gains control of the root user, it doesn’t need to specify additional permissions. Saint service in [11] offers a fine-grained access control model that allows application developers to attach security policies to their applications, in particular to the application’s interfaces. It enforces security decisions based on signatures, and configurations, while security decisions themselves are enforced both at install time and at runtime. The problem with Saint’s approach is that application developers are not security experts, and most of them do not understand application security well enough to create a strong security policy. Quire [12] is a security extension that provides a system that prevents confused deputy attacks via Binder IPC. To determine the originator of a security-critical operation, Quire tracks and records the IPC call chain and denies the request if the originating application has not been assigned the corresponding permission. It does not solve the privilege escalation flaw. Bugiel et al. [13] combined TOMOYO Linux (a customized version of SELinux) and a decision making machine to deal with privilege escalation at both the IPC level and Linux levels. Their solution lacks the use of sensors found on Android devices. StoreDroid, on the other hand, will limit access to secure data to one specific user by leveraging the sensors to add another level of security. The TaintDroid information-flow tracking system in [14], [15] analyze 30 Android applications that required access to the Internet and revealed that some Android applications do exploit user data for purposes that may not be expected or desired by users.

III. PROBLEM DESCRIPTION AND MODEL

StoreDroid targets a wide range of applications executed on a smartphone and communicates with remote service providers as depicted in Fig. 1. One example is when the smartphone owner accesses his own bank account remotely from the smartphone. Such applications share with the service provider secret data that might be used for authenticating the owner of the smartphone in case of remote access or other business logic between them. Any remote information will be stored in the local storage, and the application may communicate with the remote service for the purpose of transmitting this information to the service provider. Both the owner of the smartphone and the provider wish to protect the data and keep it private.
This section models the above scenarios in terms of trustiness and lists the possible threats that may compromise the secret data. Consequently, Section IV will describe how our framework addresses these threats and protects the secret data.

A. Application Example

A simplified example of such scenarios is WorkTimeTracker, an application that helps the employer to track the working hours of his employees. A remote server collects the reports the employees send and stores them. The employer provides his employees with this specific application to run on their Android smartphones. The employee uses this application to report daily his or her time of arrival at and departure from work. This application holds the employee’s unique ID internally and secretly.

The WorkTimeTracker application ensures and authenticates the employee who reports the time. For this purpose, WorkTimeTracker uses the locally stored unique employee ID for the employee’s authentication by the employer.

The following is required in order to protect the WorkTimeTracker secret data from unwanted access: (1) the application does not allow the employee to fool the employer and prevents him from extracting the secret ID and report hours from another program; (2) malicious applications cannot hijack the ID; (3) the service provider—that is, the employer—imposes sensor-based restrictions to semantically control the data access. The semantic rules may include a rule saying the employee must report the work hours within a fifty meter radius of the company office.

This set of requirements cannot be satisfied with the current Android phones (version below 4.3). Once a program gains access to the Linux root user account, all the data stored in the phone is under its control, including the operating system itself. Furthermore, once the attacker has control over the operating system, data coming from the sensors can be easily manipulated and tampered with, and network/memory injections can be easily performed to hijack data. Hence, no semantic rules can be involved with current access control. Thus, our main contribution is the addition of the sensor-based access control.

B. Component and Threat Modeling

We identify a set of threats that are common to such applications in which an application has secret data in its possession, and communicates with a remote service provider. Our threat model assumes a trusted application (that possess the secret data and authenticates itself) and the trusted service provider application. These scenarios have the following adversaries:

- Malicious applications: Unprivileged applications might try to access the sensitive information without permission (e.g., privilege escalation).
- Data leakage: In some scenarios the trusted Android application might need to communicate with other, untrusted Android applications to accomplish its goal. Thus, we need to deal with information leaking out by mistake.

- Impersonating hosts: A malicious host on the network can try to impersonate the back-end server (e.g., by spoofing the back-end server’s IP) and divert the data sent from the smartphone to it.
- Mimicking applications: Malicious applications might start communication with a server and try to mimic the trusted application.
- Network sniffing: Another host might try to use a network sniffer and read the data sent on the network.
- Malicious user: The user might want to circumvent the rules imposed on him by the service provider.

IV. StoreDroid Framework

StoreDroid is an integrated framework that provides a stronger protection to the applications under the scenarios that were described in Section III. This section describes the details of StoreDroid architecture detailed in Fig. 2 and is based on the architecture of Android that was described in section II. Our solution consists of the following layers:

1) StoreDroidApp is a new trusted stub application located in Android' application Layer. We describe it in subsection IV-B. This application provides stronger file access control by incorporating data coming from a variety of sensors provided by Android to create rules that are based on biometric data, GPS, and more. In addition its security relies on SELinux policy enforcement capabilities that were set when the ROM installed.

2) At the operating system layer, we replaced the Linux kernel with a customized SELinux-enabled kernel. Our ROM consists of a set of strict policies that define fine-grained access control over the secret data. This layer is described in subsection IV-A.

3) The secure message passing layer is described in subsection IV-C - We secure the messages transferred between
the service provider and the smartphone by authenticating and encrypting the communication between them.

To thoroughly explain the strength of our StoreDroidApp sensor-based access control, we will first present the capabilities and benefits of the SELinux infrastructure.

A. SE Android and SELinux

For the operating system layer, we use an LSM (Linux Security Module). The LSM is a lightweight, general purpose, policy-based access control for the mainstream Linux kernel. The LSM mechanism allows a variety of access control models to be implemented as loadable kernel modules [16]. There are plenty of such LSM implementations, but we chose SELinux which was developed by the NSA. SELinux provides a fine-grained control over sensitive data and protects from applications that hijack the root user or manage to get by Android’s data security. SELinux allows the creation of a custom security policy that specifies the exact applications that have access to the data. With this policy one can achieve a higher level of control over the data and even protect from the root user. Shabtai et al. [17] showed that Android can run an SELinux-enabled kernel with minimal performance impact.

However, adding SELinux to the Linux kernel of Android is not straightforward. SELinux cannot be loaded into Android’s Linux kernel without additional work and customization on the Android stack. SELinux needs to be aware that Android applications run inside the Dalvik VM, where each application is forked from a single process (the Zygote process), hence having the same process id. To distinguish among the different applications, each application has to be labeled with an appropriate id, and the access control should consider it. Complementarily, each file has to be annotated with these labels to enforce the extended SELinux access control. Alas, Android’s file system yaffs2 [18] does not support extended file system attributes that are required for these labels.

To address these issues and others, the NSA has developed the Security Enhanced Android, termed SEAndroid, a package that customizes SELinux so it can be integrated with Android. It extends AOSP (Android Open Source Project) [19] and can be easily incorporated to a custom Android build using Android’s version control tools [20] and flashed to an Android phone. SEAndroid provides us with important, Android-specific features [21] such as per file security labeling support for yaffs2, file system labeling at build time, kernel permission checks controlling Binder IPC, and more.

Specifically, once SELinux is configured and running in the enforcing state, each access to system resources (such as files and sockets) is checked against a policy. The SELinux policy is a set of rules that guide the SELinux decision-making engine. It defines types for file, and objects and domains for processes. Then it uses roles to limit the domains that can be entered and has user identities to specify the roles that can be attained. More details on SELinux appear in [16].

The steps to create the ROM

- For having an SELinux-enabled Android running on an emulator, we used AOSP (Android Open Source Project) version 4.1.1 together with Linux-enabled Linux kernel and the SELinux-specific policy files that had been tailored specifically for Android’s needs. Note that the newest Android, version 4.4, already has SELinux built-in for all its users [22], though there is no way to customize it with a generic trusted stub as we did.
- We made the modifications in the SELinux policy so that only our trusted application, StoreDroidApp, will be able to access the secret data. Thus, we leveraged the currently used Linux access control, as will be described further on.

B. StoreDroidApp: Sensor-Based Access Control

The second part of StoreDroid is StoreDroidApp, a trusted applicative stub that is used by any application that has secret information. StoreDroidApp will address the threat model we defined in section III by providing two separate mechanisms: first, limiting secret file access to the owning application only by using SELinux policy enforcement, and second, given a set of rules, StoreDroidApp consults a variety of sensors. Based on the data provided from these sensors, it allows or disallows access to the secret data. StoreDroidApp is designed and built in a modular fashion to embed any application of this type. A developer who wishes to use StoreDroid’s components can use StoreDroidApp’s APIs to contain and protect secret information and set the sensor-based rules as required.

![StoreDroid Architecture](image-url)

The WorkTimeTracker application is again used as an illustration. It lets employees track their work hours through an Android application. Also, the employer wishes to make sure that the employee is actually at work when performing his work-hour tracking. The secret data in this example is the ID of the employee and the keys that are used for authenticating the employee to his employer. The WorkTimeTracker application

- For having an SELinux-enabled Android running on an emulator, we used AOSP (Android Open Source Project) version 4.1.1 together with Linux-enabled Linux kernel and the SELinux-specific policy files that had been tailored specifically for Android’s needs. Note that the newest Android, version 4.4, already has SELinux built-in for all its users [22], though there is no way to customize it with a generic trusted stub as we did.
- We made the modifications in the SELinux policy so that only our trusted application, StoreDroidApp, will be able to access the secret data. Thus, we leveraged the currently used Linux access control, as will be described further on.

B. StoreDroidApp: Sensor-Based Access Control

The second part of StoreDroid is StoreDroidApp, a trusted applicative stub that is used by any application that has secret information. StoreDroidApp will address the threat model we defined in section III by providing two separate mechanisms: first, limiting secret file access to the owning application only by using SELinux policy enforcement, and second, given a set of rules, StoreDroidApp consults a variety of sensors. Based on the data provided from these sensors, it allows or disallows access to the secret data. StoreDroidApp is designed and built in a modular fashion to embed any application of this type. A developer who wishes to use StoreDroid’s components can use StoreDroidApp’s APIs to contain and protect secret information and set the sensor-based rules as required.

The WorkTimeTracker application is again used as an illustration. It lets employees track their work hours through an Android application. Also, the employer wishes to make sure that the employee is actually at work when performing his work-hour tracking. The secret data in this example is the ID of the employee and the keys that are used for authenticating the employee to his employer. The WorkTimeTracker application...
is embedded in the StoreDroidApp stub. The employer can use the StoreDroidApp to apply restricting rules on the WorkTimeTracker application. One rule, for example, validates that the employee is in his workplace by using the GPS information. Thus, upon trusted registry the server’s application sets the rules for the WorkTimeTracker application (specifically, the location rule) via the StoreDroidApp service. Upon any request to report the hour, StoreDroidApp’s access control fetches the location rule and queries the sensor service, in this case the GPS. The employee ID and hour report is accessed only if all rule conditions are met.

Fig. 2 presents the following software module of the trusted StoreDroidApp stub:

1) A GUI - a set of Android activities that the user interacts with. The GUI is specific for each use case and is the user-facing part of the system. In the WorkTimeTracker example, an activity shows the user what will be reported- namely his ID and the time to be reported. The business logic of the application is embedded in this activity. Thus, the GUI will be the only part of the framework that will be changed from application to application. Any such change requires a new build of the StoreDroidApp stub.

2) Rules manager module - pushes new rules from the server to the database that is located on the smartphone. We define a new protocol between the service provider and StoreDroidApp. The messages sent by this protocol are used by the service provider to dynamically update the rules. Rule updates can be initiated by the service provider or prior to an access request and before a sensor check. More examples of the transferred messages are described in section IV-C.

3) Sensor Service - a background Android service that is in charge of querying all the sensors (through generic sensor logic such as GPSLogic, BluetoothLogic, and so on) and returning the appropriate answer back to the activity. For example, in the WorkTimeTracker, once the user clicks the “Report” button, an activity binds to this service and asks for confirmation that the user is complying with the rules set by the service provider. It prevents the usage of faked sensors by defining the sensor service as a private service that cannot be accessed from outside the scope of the application, thus reducing the attack surface. The message passing between the private service and the activity is akin to communication between threads, and thus it is considered secure. On the other hand, in a scenario when there will be several applications that implement the StoreDroid framework on one device, all will create their copy of this service. Thus, code redundancy could be saved by having one global service to which all StoreDroidApp stubs can bind. In such a case, the message passing between the activity and the global service becomes untrusted, requiring additional security measures such as two-way authentication. Also, having a centralized service will improve manageability because with only one update to the service one can modify the rules for all the applications using StoreDroidApp. We have decided to implement it using a private service in favor of isolation and security.

4) Database manager - holds a table for each sensor type. Each table holds the conditions that have to be asserted each time the user wishes to perform a specific action. For example, the database holds a table of locations and radiuses that are used to create a boundary in which a specific action can be performed. To increase the code modularity, sensor service asserts all the rules through their respective handlers. The SELinux mechanism guarantees that no application can tamper with the database.

5) Sensor logic - is a class that is responsible for polling the various sensors and returning the data back to the background service. This module is divided into smaller sub-modules so that each sub-module will be in charge of querying a single sensor. For GPS queries we use the GPSLogic.

6) Network manager - handles all of the network operations. It performs the communications with the service provider’s server and is described in detail in section IV-C.

StoreDroid’s modularity as a framework is expressed by the fact that it is not written for one given use case. The only implementation-specific part of StoreDroid is the user-facing part. If it is desired to use StoreDroid for another use case, all that is necessary is to replace the module that interacts with the user (in our case, MainActivity) with another implementation, and all of the security elements will still continue to serve as planned.

C. Secure Messaging Layer

We deal with applications that communicate remotely with another server sharing a mutual interest to keep the transferred data and communication safe. There are several types of messages that can be transferred between the server provider and the Android application. For example, the conversation between the WorkTimeTracker and the employer goes as follows:

1) Update Request - in which the client identifies himself to the server by sending the secret ID and the current version of the rule set that is applied.

2) Rule Response - after the server has verified that the client ID is valid, it checks if the rule version on the client’s device is up to date; if it is not, the server will send the set of the current rules to the client. If the rules on the client are up to date, the server will reply with the same number that was sent to it.

3) Report Request - after the client has executed all the necessary sensor validity checks with the most recent rule set, it will create a report request and send it to the server in a string format.

The transferred messages are serialized and encoded with the JSON standard that allows the serializing and de-serializing
of any object that need to be transferred/received in the network.

To make sure that no third-party host can eavesdrop on the conversation, we secure the transferred messages at the transport layer by using the SSL protocol. SSL provides data encryption as well as authentication protocol, so the client can trust the server and know that no hostile host took over the trusted server’s address. The service provider keys are installed and exchanged in the smartphone during the installation of the phone’s operation system. Additionally, we modified the SELinux policy and explicitly stated that only the StoreDroidApp has access to the key. Thus we trust that the key that is stored with the application is secured.

V. System Setup and Evaluation

A. Configuration

This section presents the required steps to install and configure the StoreDroid framework. As described, we protect an application running on the smartphone by embedding it in the StoreDroidApp stub. We set the maximum number of stubs to embed applications, and prepare their infrastructure when the ROM is installed. At a later stage during the runtime, each application that requires higher security will be assigned to a stub. To prevent a malicious attack when assigning the application to the stub, we require the application to be added in a trusted location such as in the site of the service provider.

1) ROM Preparation: We have defined ten stubs (stub0-stub9). The following steps, described for stub0, will be operated for all other stubs as well. The sensitive data of stub0 was set under a directory named stub0_data. Since we want only stub0 to access stub0_data, we create a type (domain) for stub0, a type (object) for stub0_data data and a policy that sets access rights accordingly.

The following files were modified to allow the stubs to run under SELinux. Note that they are written in a special SELinux language: M4 macro language.

| file.te | Defines the types that are used in the system. |
| file_contexts | Maximum Concurrent Multi Commodity Flow |
| app.te | Holds the policy for known applications |
| seapp_context | Configuration is used to label app processes and app package directories |
| port_contexts | Configures TCP/IP port labels |

First, we declared a type for the data files we want to protect; the type will be called top_secret_file. We added the following line to file.te file:

```
type top_secret_file, file_type, data_file_type
```

Afterward we set the default context for our data files. It states that our files will be labeled with our context at the initial files-system labeling. We edited the file_contexts policy file as follows:

```
/data/secret, */?u:object_r:top_secret_file:s0
```

Then we created a domain for our stub application by adding the following line in the policy file_app.te:

```
type secret_app, domain:app_domain(secret_app)
```

Our stub application will then be labeled with the domain in seapp_context file:

```
user=app seinfo_release name=il.ac.jce.StoreDroid
domain=secret_app type=platform_app_data_file
```

Finally, we create a policy stating that our domain can access our object. We write it in the file secret_police.te, place it in /external/sepolicy so it will be compiled with all the other policies and loaded into the kernel at the next boot:

```
> module sec 1.0:
require {
type secret_app;
type top_secret_file;
class file read write getattr ;
allow secret_app top_secret_file:file read write getattr ;
}
```

The above commands ensure that only our stub application can access the data.

In order to handle the network connections, the port towards the service provider will be mapped with a security context. The following line will be added to the port_contexts file:

```
portcon tcp 1234 u:object_r:secret_port:s0
```

Then a policy allowing our stub to use this port, and save it will be added to /external/sepolicy:

```
> module sec 1.0:
require {
type secret_app;
type top_secret_port;
type secret_port;
class port bind connect shutdown recvfrom sendto recv_msg send_msg ; allow secret_app secret_port:portbind
connect shutdown recvfrom sendto recv_msg send_msg ;
}
```

2) Runtime: Suppose we wish to protect the data of the WorkTimeTracker application; it will be downloaded in a trusted position at the employer and be integrated with stub0. With those policies set in place, we can guarantee that WorkTimeTracker can have only stub0’s permissions and access to the files containing the user id. No other application can interfere with the communication with the server, and no other application can access system files and corrupt the input coming from the

B. Addressing the Threat Model

1) Malicious applications: We address privilege escalation by using SELinux to set file-grained permissions over our data and specify that only the trusted application can access the secret data.

2) Data leakage: In this scenario we consider the server to be trusted, so this has no meaning here.

3) Impersonating hosts: A malicious host on the network can try to impersonate the back-end server. We address this issue by considering the service provider application to be trusted. Specifically, we use two-sided SSL connection so that the stubbed application can verify the back-end server’s application and vice versa.
4) Mimicking applications: Malicious applications might start communication with the server and try to mimic the trusted application. We handle this by verifying each application communicates with the server only after an SSL authentication.

5) Network sniffing: Another host might try to use a network sniffer to read the data sent on the network. The SSL channel between the smart phone and the server provides us with encryption so that no other application can eavesdrop on the communication.

6) Malicious user: The user might not always wish to comply with the limitations imposed on him by the system. The stub enforces a rule check before performing user’s request. SELinux prevents unwanted access to system files that are responsible for talking to the sensors. Thus, the validity of the data coming from the sensors is guaranteed.

C. Testing and Validation

We have implemented the above process using as a baseline a NEXUS-4 phone that was running the Android 4.1 version. The SEnAndroid was downloaded from the NSA site. The setup was performed using Eclipse with an ADT development environment on a standard PC.

Functional Testing The crucial components have been tested to verify that everything is working properly and that the components function as intended:

- Refuting threat number 3 - We have tried to change the certificate on the server side (to simulate an IP spoofing attack on the client). The result was that the client threw an SSL exception.
- Refuting threat number 4 - We have tried to change the certificate on the client side (to simulate a mimicking android application). The result was that an SSL exception was thrown on the server side.
- Refuting threat number 1 - We have tried to access the secret data using a ‘root’ user. The result was an “access denied error.”
- Refuting threat number 5 - We have used WireShark to listen to data that was being transmitted. The result was that the data captured was not readable.
- Refuting threat number 6 - We have used the application not according to the rule set. The result was a message saying that “the action is not permitted.”

VI. SUMMARY

We have described why the standard Android environment is not safe enough for data-sensitive applications, especially because of malware penetration from either malicious or careless users. We have shown that there are already tools that can be used to prevent such threats. We have built a sample application using these principles. We have also packaged this framework - the storeDroid. This framework can either be used directly or as a building block to build safer applications simply by implementing a GUI for the already included stubs.

REFERENCES