ABSTRACT

With the introduction of the smartphone, mobile phones have gone from closed systems with limited capability to mobile software platforms with rich software ecosystems. The same operating systems are now present in other mobile devices as well, such as tablets. The security mechanism of such software platforms have, apart from end-user requirements, heavily been influenced by business and legislative requirements.

The enormous growth of the smartphone business has spurred interest in research on mobile device security. This paper explores the security architecture and access control mechanisms present in Android, a well established mobile device platform and compares the approach to Tizen, a up-and-comping mobile operating systems developed by Intel and Samsung. In addition some research directions in mobile device security are briefly presented.

1. INTRODUCTION

In 1996 Nokia introduced the Nokia 9000 Communicator, one of the first devices reminiscent of modern smartphones. In addition to phone calls it allowed uses to send and receive faxes, e-mail and SMS messages as well as access the Internet. It also featured functions such as an electronic address book, calendar, notepad and perhaps most importantly an open development environment enabling third party application development [5]. Since then, smartphones have become a multi-billion dollar business. According to estimates by Strategy Analytics the number of smartphones in use exceeded 1 billion units in 2012 [7].

Modern smartphones are characterized by many of the same properties as the original Nokia Communicator. They support installation of third-party applications, typically have wireless Internet access and also contain private or sensitive information such as personal messages, contacts etc. [18]. Recently, the same operating systems present in smartphones have been adopted to a multitude of different devices, such as tablets, netbooks, smart TVs and perhaps, in the near future, In-Vehicle Infotainment (IVI) systems as well [24].

Mobile phones started out as closed systems with limited functionality. As mobile platforms have been opened up to third-party developers, it’s become increasingly important to protect end users from malicious or malfunctioning software. Apart from privacy issues, which can become a nuisance, such software can cause direct harm, such as incurring considerable financial losses with phone plans allowing access with little user intervention to premium numbers and services [18].

Even though the number of mobile malware samples collected by security companies such as F-secure and McAfee has increased dramatically in 2012 compared to previous years [12, 19], the reasons behind platform security measures in place in mobile devices are largely related to business and regulatory requirements. Apart from malware, the threat model of mobile operating systems also includes malicious users attempting to circumvent content protection schemes, subsidy locks or otherwise tamper with the device. Regulators want to ensure secure storage of radio frequency parameters and device identifiers usable as theft deterrent, such as the International Mobile Equipment Identifier (IMEI). Mobile networks operators want to enforce vendor lock-in, such as in preventing customer who bought subsidized mobile phones from moving to other operators for the duration of their contract. Preventing tampering with the IMEI and the International Mobile Subscriber Identifier (IMSI) is essential in enforcing such subsidy locks [18].

In order to meet the business, regulatory and end-user requirements mobile platform vendors have developed various hardware and platform security measures. Many platforms have been designed with some kind of permissions based security schemes of varying granularity allowing limited access to device capabilities by third-party applications according to the principle of least privilege. A trend in such systems seems to be movement towards finer granularity in permissions and better extendability by third-party developers. The developments of mobile platforms and the increase in popularity of smartphones has also led to an increased interest in research on mobile device security [18].

This paper explores the security architecture and access control models of two Linux-based operating systems for mobile devices. A well established system, namely Android, is compared to Tizen, a up-and-comming system targeting various device types. The architecture and access control models of the two systems are described in Section 2. Commonalities and differences between the two systems are discussed in more detail in Section 3. Section 4 gives a brief overview of the current state of research concerning mobile device security. Section 5 concludes.
2. MOBILE SOFTWARE PLATFORMS

What follows is an overview of the security architecture and access control models of Android and Tizen. Both operating systems are Linux-based, which presents a favorable basis for comparison. Furthermore, information on the internals for both systems is publicly available. Other prominent mobile platforms, such as iOS, Windows Phone 8 and Blackberry, for which information is not as readily available, are outside the scope of this paper as are recently introduced operating systems such as Firefox OS and Ubuntu Phone.

2.1 Android

Android is an operating system for mobile devices developed by a consortium of technology companies and mobile operators known as the Open Handset Alliance, led by Google [23]. According to estimates made by research company Gartner, Android had a market share of close to 70% of worldwide smartphone sales to end users in the fourth quarter of 2012 [14]. Android is also the most targeted platform by malware. According to a report by F-secure, close to 80% mobile threats target the Android platform [12].

2.1.1 Architecture Overview

The foundation of the Android platform architecture, shown in Figure 1, is the Linux kernel and a set of device drivers depending on the devices hardware configuration. At the core of the architecture lies the Android Runtime, consisting of the Core Libraries and the Dalvik virtual machine, a register based bytecode interpreter optimized for mobile systems. Android also includes C/C++ libraries which provide various functionality to different system components, such as a BSD-derived standard C system library (libc), SQLite database and the Webkit rendering engine. The functionality of these libraries is exposed to developers through the Android Application Framework, which provides an Application Programming Interface (API) for accessing middleware system services underlying each application [1].

Application Components

Android applications are typically written using the Java programming language, where the Java standard library has been replaced with the Android Application Framework. However, applications can also include or consist completely of native code [3].

Android applications consist of four different kinds of components [8]:

- **Activities** are visible portions of application user interfaces [8].
- **Services** are software components which run in the background without interacting directly with the user. Services provide functionality to other application components through a client-server interface. In order to invoke methods provided by service’s interface, the client component must first bind to the service [8].
- **Broadcast Receivers** facilitate messaging between multiple applications [8]. These are discussed in more detail in Section 2.1.2.

Content Providers provide persistent local data storage and also facilitate sharing data between applications. Content Providers are accessed through a Uniform Resource Identifier (URI) defined by the application owning the content provider [8].

Application access to system resources is controlled via installation permissions. Permissions may be required to invoke methods defined in the Android Application Framework API, access Content Providers and for Inter Process Communication (IPC). Applications can also define permissions of their own in order to protect access to application components [13].

Android applications are distributed as **Application Package Files** (APKs). Applications must declare in advance what permissions they require in a application manifest file, contained within the APK. When an user installs a application the permissions required by that applications are displayed to the user, who must confirm the permissions in order to continue with the installation. The user does not have the option to selectively refuse certain permissions from a particular application but can abort the installation [25].

APKs are cryptographically signed with a developer certificate. Applications updates are required to be signed with the same certificate as the original application [25]. Applications will not receive automatic updates if the the new version of the applications declares more permissions than the old one. In this case the application update must the confirmed by the user. This has been suggested as an incentive to break the principle of least privilege, and declare unnecessary permissions in order to automatically update in the future when the extra permissions have become necessary [13].

Application Sandbox

The kernel enforces process and file level separation between applications and the Android system through Linux user identifiers (UID) and group identifiers (GUID). Each application is assigned a unique low-privilege UID, which functions as the effective UID of the application process and owner of the applications files. Thus, by default, applications cannot interact with one another or access each others files. As the application sandbox is enforced by the kernel, all software above the kernel, including the C/C++ libraries, application framework and runtime and the application code itself, including native code, is subject to to the process isolation [3].

Permission Enforcement

The Android Application Framework consists of a library contained within each applications runtime environment and an API implementation running as part of the system process(es). The API implementation in the system process is privileged and not subject to any permissions checks, whereas the library is run with the same permission as the application it is attached to. When an application invokes a function, part of the public library API, the library invokes a Remote Procedure Call (RPC) stub part of the libraries private interface, which in turn initiates an RPC request.
to the system process where the appropriate system service handles the requested operation [13].

To enforce permissions the API implementation in the system process calls a permissions validation mechanism when necessary to make sure the calling application has the required permissions to invoke the operation in question. The framework library in the application process may also redundantly check the calling applications permissions. However, this must not be relied upon, as the calling application may utilize Java's reflection mechanism to invoke the libraries private RPC stub directly, thus bypassing any permissions checks in the API method of the library [13].

Some permissions are enforced by the kernel by using Linux groups. Namely applications which are granted the INTERNET, WRITE_EXTERNAL_STORAGE or BLUETOOTH permissions are assigned to groups which have access to the relevant sockets and files. Thus the framework library which runs within the application process and therefore has the same access rights as the application itself can access the relevant sockets and files directly, without going through the system process [13]. It is worth noting that the external storage is shared between applications and any files that need to be protected from being accessed by other applications should be stored on internal storage instead [2].

### 2.1.2 IPC Mechanism

The Android message passing system is based on self-contained messages called **Intent**s. Intent is used both for inter-application and intra-application communication. Additionally the operating system can notify applications of certain events by sending Intents. Such notifications can be broadcast system-wide [8].

#### Intents

Applications can specify an Intent to be delivered to a particular application component, specified in the Intent itself. These are called *explicit* Intents. Apart from the recipient component name Intents can include *action*, *category* and *data* fields. The action field specifies an operation to be performed upon the receipt of the Intent. The category fields provides additional information about the operation and the data field specifies the type of data to operate on. If an Intent does not include a recipient component name it is considered *implicit*. This means that the Intent leaves determining the recipient up to the platform based on the remaining Intent fields. In other words the intent can be delivered to any application which supports a particular operation. Implicit Intents are important in that they enable late runtime binding between applications [8].

Intents can be sent between Activity, Service and Broadcast Receiver application components introduced in Section 2.1.1. Intents are used to start and stop Activities, bind to Services and notify Broadcast Receivers. Both explicit and implicit Intents can be used for these purposes, but by default any component can only receive Intents originating from within the same application. In order to receive external Intents, components must be *exported* in the application **manifest**, a configuration file included in the application installation package. A component is considered exported if it has least one **Intent filter** or the **exported** flag in the manifest. Broadcast Receivers can exceptionally also be exported at runtime [8].

#### Intent Filters

Intent filters are used to constrain the type of implicit Intents that will be delivered to a component. An Intent filter...
Broadcast Intents can be broadcasted or received by the operating system and applications depending on the type of broadcast [8]. When the system needs to deliver an Intent to a particular component, it must first identify which component is capable of receiving the Intent. This is done through the Intent Filters, which are specified within the AndroidManifest.xml file of each application.

When multiple applications are registered to receive the same Intent, the system must determine which component the Intent should be delivered to. This is done through the Intent Filters, which specify the conditions under which an Intent should be delivered to a particular component.

Broadcasts

Broadcast Intents can be normal, sticky or ordered. Normal broadcasts are sent to all Broadcast Receivers eligible to receive the Intent at once. Sticky broadcast are similar to normal broadcasts, but are stored by the system and re-broadcast to new eligible Broadcast Receivers when such become available. Ordered broadcasts are delivered each eligible Broadcast Receiver one at a time, and the propagation of the Broadcast Receiver stops when one of the receivers in the chain decides to handle the Intent. Broadcast Receivers have the option of specifying a priority level as part of its Intent filter, which is used to determine the order of in which ordered broadcast are delivered to Broadcast Receivers [8].

Intent Security

The Intent Filter mechanism by itself does not provide any security, as a sender can specify any action, category or data in an Intent or bypass the Intent Filters entirely by sending an explicit Intent. Furthermore there is no restriction on the kind of Intent Filters that can be defined for a component, allowing any component to become an eligible receiver for implicit Intents regardless of if the component is capable of performing an action requested by an Intent. In order to protect exported application components from receiving malicious Intents developers can add permission requirement to the manifest for the entire application of individual components. Broadcast senders can limit to the propagation of broadcasts Intents to Broadcast Receivers having a particular permission [8].

2.1.3 Permission Granularity

Android permissions are divided into four levels [8]:

- **Normal** permissions control access to API calls which are not considered harmful, such as changing the home screen background image or cause the device to vibrate [13]. These permissions are automatically granted to all application [8].

- **Dangerous** permissions control access to API calls which can be harmful if abused, such as incurring monetary costs for the user or expose private information [13]. Dangerous permissions are granted by the device user at applications install time [8].

- **Signature** permissions protect functionality which can cause significant data loss or damage such as resetting the system to factory defaults or disable the device. To obtain signature permissions an application must be cryptographically signed using the device manufacturer’s certificate. Signature permissions can also be used by developers to restrict access to an application component to applications by the same developer. In this case the application requesting the permission must be signed with certificate of the developer that defined the permission [8].

- **SignatureOrSystem** permissions can be obtained in the same way as signature permissions or installing the application to a special system directory. Typically such applications are pre-installed by the Original Equipment Manufacturer (OEM), manufacturer [13].

The purpose of division of permissions into different groups is to raise the difficulty of obtaining the most destructive permissions. Obtaining Normal permissions is trivial. Obtaining Dangerous permissions requires user confirmation during install. Signature of SignatureOrSystem are the hardest to obtain as they require access to the OEM signing certificate or root access to the device normally available only to the OEM [13].

In order to access the Internet an application needs the Dangerous INTERNET permission and thus Internet access needs to be confirmed by the user. The INTERNET permission is coarse grained in that once obtained, the application is granted full access to the Internet. It is not possible to restrict Internet access to a particular host or domain [17].

Read and Write access to system Content Providers, such as the Contacts Provider are protected by separate permissions (READ_CONTACTS and WRITE_CONTACTS) in case of the Contacts Provider. It is however not possible to restrict read or write access to only particular contact fields [17].
Some sensors, such as the Global Positioning System (GPS) subsystem provide both coarse and fine grained access to an approximate or more exact location through the ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION permissions [17].

Some Android permissions provide read or update access to several unrelated properties, such as the READ_PHONE_STATE permission which, amongst other things, allows an authorized application to determine if a phone call is currently occurring, but also allows the application access to the IMEI number of the device. Another such example is the WRITE_SETTINGS permission which allows an application to change several distinct settings on the device, harmless ones such as changing the default notification sounds, but also modify the network settings [17].

2.2 Tizen

Tizen is an Linux-based open source software platform aimed towards many different kinds of devices such as smartphones, tablets, netbooks, IVI systems and smart TVs. One of the aims with the platform is to provide consistent user experience across these different types of devices. Tizen development is governed within The Linux Foundation by a Technical Steering Group consisting of Intel and Samsung [24].

2.2.1 Architecture Overview

The Linux kernel along with device drivers and a Hardware Abstraction Layer (HAL) make up the foundation of the Tizen platform architecture, shown in Figure 3. The middleware layer of the architecture consists of a set of core services such Multimedia, Telephony and Messaging subsystems. Tizen provides a Web API which consists of HTML5 APIs and the Tizen Device API, which provides access to the core services. Web applications (widgets) are the primary type of application on the Tizen platform, although native applications are also supported through separate APIs as well [22]. The Web RunTime (WRT) which handles web application installation and execution is based on the WebKit rendering engine [6].

Access Control Model

Access control enforcement in Tizen is done on two levels. Process level containment is based on a kernel level application sandbox. On top of this the WRT Access Control Engine (ACE) provides fine grained access control of JavaScript APIs [6].

The Tizen access control model differs from the one in Android in that all applications run under the same predefined non-root UID as do most of the middleware daemons of the core services layer. Privilege elevation from non-root user to root is not allowed. Access to privileged features is provided by service daemons and subject to user space access control [16].

Application Sandbox

Application sandboxing on Tizen is based on kernel enforced access control through the Simple Mandatory Access Control Kernel (SMACK). SMACK is a Linux Security Module (LSM) which provides a simple filesystem label based access control enforcement [16].

A subject in SMACK terminology refers to an active entity attempting to access resources i.e. a Linux task. SMACK recognizes read, write, execute and append accesses. An object is a passive entity that is accessed, such as a files, sockets or tasks. A label identifies the security characteristics of a subject or object in the SMACK context. SMACK defines two types of labels that are stored in extended file attributes. Object labels define a SMACK context for filesystem objects. Process labels can be defined for executables and set the SMACK context of a subject when the executable is executed [6].

The access control policy enforced by SMACK is defined as a set of rules of the form: subject_label object_label access_mode
For example the rule:
Album Camera r
would grant read access to any object with the label Camera for any subjects with the label Album [6].
In order to support SMACK Tizen, adds a Security Manifest Plugin to the RPM Package Manager which sets SMACK labels for files being installed based on a manifest file included in the RPM package. The manifest must at least declare a security domain for an application, which will determine the default SMACK labels for the application and its files. Additional labels can be defined and assigned to application files in manifest as well [16].

Each widget runs in a separate security domain. A widget cannot access the files of other applications, systems files nor communicate other processes such as system service daemons without proper SMACK rules. SMACK rules are configured during install, uninstall or update by the package manager, at runtime by a trusted Security Server daemon and as a result of user confirming feature permissions for a widget [6].

Tizen also adds SMACK support to Xorg through the X Access Control Extension (XACE) which allows for isolation between processing requiring Xorg access. Similarly the D-Bus IPC system has been modified to enforce SMACK access rules in order to restrict access between application accessing the message bus. Finally the udev device manager has been modified to create device nodes with SMACK labels, allowing for instance the device node /dev/camera to be created with a Camera label [16].

2.2.2 IPC Mechanism

Application launch is aided by Application Utility Library (AUL). The AUL consists of a library part of the application which will send and receive requests for application launch and termination. A AUL daemon, better known as the Launch Pad, will handle the requests made by the AUL library. The AUL enables applications to invoke other applications to perform operations on their behalf. Applications can explicitly invoke a particular application or implicitly request a service and let a high level application launch service, the App Service, determine the actual application to launch [21].

The App Service exposes general operations, such as view, create, call etc. Each service is described through an operation, URI scheme and a MIME type. The App Service will inquire the Application Information Library (AIL) for services matching a particular description. The AIL in turn will query the Application Information Database, which houses information on installed applications and the services they provide. The database is maintained by the package manager which will invoke the AIL to add application information at install time. Once the application to be invoked has been determined, either via the App Service or an explicit launch request, the AUL library will request the Launch Pad to launch the application, passing along possible launch arguments. The Application Data Exchange (ADE) occurs through a dictionary of storing abstract key-value pairs called a bundle. If the calling application does not specify any launch arguments the Launch Pad can query the Application Information Database through the AIL for the default configuration of the application to be launched. The AUL will then proceed to launch the application passing along the bundle of launch arguments. Any results generated invoked application is passed back as a bundle to the invoking application [21].

User Space Access Control

When unprivileged applications are accessing system resources through the system services such as the Launch Pad or App Service, the actual privileged operation is done by the service itself, thus kernel enforced access control is insufficient and the service must enforce access restrictions regarding the calling application. Tizen has two mechanisms for this, Direct IPC and Indirect IPC, illustrated in Figure 4 [16].

With Direct IPC the application will make a service request to the service daemon directly through native sockets. The
Service in turn will check the SMACK rules for the client process label through SMACKFS access, a mechanism allowing userspace application check SMACK permissions [16].

With Indirect IPC the unprivileged application will first contact the Security Server daemon through a native socket interface. The Security Server will identify the client and issue it a random security cookie. The client can then use the issued cookie to contact the appropriate service daemon through an intermediate process (namely D-Bus). The service daemon can verify the permissions of the requesting application by consulting the Security Server, which can identify the client application using the security cookie. The Security Server will verify the SMACK rules of the client label on behalf of the service daemon [16]. Without the security cookies and assistance from the Security Server the service daemon would not be able to identify application permissions on individual application level and access control would rely on the more coarse grained access to a particular D-Bus interface. This would open up the service daemon to privilege escalation through a confused deputy attack.

2.2.3 Permission Granularity
A subset of the JavaScript APIs on Tizen are considered restricted. Restricted functions provide access to private data on the device, such as location, contacts calendar etc. In order to access restricted parts of the API widget developers need to declare which features the widget wants access to in the widget manifest file. Features are divided into API Groups, such as Time, Calendar, Contact etc. Features pertaining to a API group typically allow full, read or write access. A feature will be granted to a widget by the WRT based on a Original Equipment Manufacturer (OEM) configurable WRT ACE policy and user confirmation to a prompt, the type of which is defined by the policy. Example prompts displayed to the user are shown in Figure 5. Users can affect the policy only in a restricted way through preference configuration. The possible prompt types are [6]:

Blanket Prompt which prompts the user to confirm access to feature once, the first time the feature is accessed by the widget. Once confirmed user is not prompted on subsequent accesses.

Session Prompt which prompts the user to confirm access to feature once per session.

One-Shot Prompt which prompts the users to confirm access to feature each time it is accessed.

Permit which always permits access to the feature, without prompting the user.

Deny which always denies to the feature.

By default widgets are denied all network access. In order to access network resources the widget manifest must declare which protocols, domains and sub-domains the widget wishes to access. Widgets can also request full network access [6].

3. DISCUSSION
Even though Android and Tizen have significant architectural differences, they both provide similar sets of features. Both have permission based access control models, allow for late runtime binding of application invocations, and similar sets of systems services. In regards to the application sandbox Android takes a rather unconventional approach in using Linux user separation mechanism to enhance process separation. Application sandboxing in Tizen is based on the SMACK LSM.

In regards to permission granularity Tizen has some advantages such as allowing restricting Internet access to certain domains, compared to the single permission in Android enabling full Internet access. A study done on popular Android applications by Jeon et.al. suggests that many applications access a fairly small number of domains, suggesting that enumerating per-domain permissions is plausible for many applications [17]. Of course it does not make sense for general purpose applications such as web browsers or feed readers, but for many applications, such as news media or social network apps, which basically offer alternatives to regular web pages per-domain permissions can be appropriate.

Another advantage with Tizen is the flexibility the WRT ACE policy mechanism allows, supporting end-user intervention while still allowing set more authoritative policies. Similar application hardening schemes have been been suggested for Android in several research projects [17, 25].

There clearly is no one-size-fits all solution to defining permission granularity. Coarse permissions might lead to overprivilege in applications [17]. Very fine grained permissions again might increase developer burden, unless the development environment provides some sort of automated mechanism for determining the sufficient application permissions.

4. RELATED WORK
Several research projects have extended the Android security model. Research has focused on four major directions: static and dynamic application analysis, security enhancements based on modifying parts of the Android system, virtualization and application hardening [25]. In this section some approaches are briefly presented.

Static and Dynamic Analysis
Static and dynamic analysis has been used to detect malware or software vulnerabilities in Android applications. A drawback of static analysis is limitations regarding Java reflection, whereas dynamic analysis can add runtime overhead and adversely affect application performance [25].

Enck et al. have developed ded, a decompiler for Dalvik bytecode and performed a study of Android application security by performing static analysis of popular applications [11].

Felt et al. have used dynamic analysis to build a permission map that correlates Android permissions requirement to Android Framework API calls. Based on the map they have also developed StowAway, a static analysis tool which detects overprivilege in Android applications. [13].
Chin et al. have developed ComDroid, a tool that performs static analysis of Android Applications in order to detect vulnerabilities related to the Android IPC model. Such may arise when implicit Intents are used for intra-application communication or application components are inadvertently exported [8].

**Operating System Modification**

Numerous research projects have modified the Android operating system to provide enhanced security and privacy controls. Unless adopted by upstream developers such modifications face considerable challenges hindering widespread adoption. A major challenge is that Android plagued by platform fragmentation due to several different operating systems versions with differing API versions being in use a multitude of different hardware configurations [25].

Enck et al. have developed TaintDroid, a system capable of tracking the flow of privacy sensitive data on Android and identified privacy leaks in popular third-party Android applications [10].

Hornyack et al. have developed AppFence, system which protects user privacy by substitutes shadow data in place of private information and hinders private data from being transmitted from the device [15].

Dietz et al. have developed Quire, a system intended to guard against confused deputy attacks by adding mechanism to Android which allows applications to operate with reduced permissions. Quire also adds a signature scheme which allows application to create signatures verifiable by other applications on the device [9].

Nauman et al. have developed Apex, a policy enforcement framework which modifies Android to allow users to selectively grant permissions to applications [20].

**Application Hardening**

In a fairly recent development systems which modify and repackage android applications, adding monitoring and policy enforcement code which enables more fine-grained access control policies compared to the default Android permissions. An advantage with these types of approaches is that they don’t require any modification to the operating system itself, allowing in principle for more widespread deployment compared to approaches discussed in Section 4. A drawback of application repackaging is that it interferes with the package signatures. This can be mitigated by re-signing packages by the same author using the same self-signed certificate [25].

Xu et al. have developed Aurasium, an application hardening tool which supports fine-grained access control features such as IP blacklisting and intercepting service access attempts allowing users to selectively confirm or deny access to individual resources [25].

Jeon et al. have developed Mr. Hide, a fine-grained permission enforcement service and Dr. Android, a tool for modifying existing Android applications to support Mr. Hide. The fine-grained permissions provided by Mr. Hide include restricting Internet access to specific URLs and restricting access to individual columns of contacts Content Provider [17].

5. CONCLUSION

This paper has given an overview of the security architecture and access control mechanism of the Android and Tizen mobile platforms. Both systems provide similar functionality, such as application sandboxing, late runtime binding of application invocations and permission based access restriction of system APIs. The more recent Tizen provides improvements in access policy configurability for device manufactures, finer-grained permissions such as per-domain network access restrictions and application feature control for end users. Similar enhancements have been developed for android by several research projects. In addition some research directions mobile device security have been briefly discussed.
6. REFERENCES


