

A HYPERVISOR-BASED IPC PROTECTION SYSTEM AND METHOD FOR ANDROID PLATFORM

Field of the Invention

This invention relates to a system and method for preventing unauthorised access to sensitive information stored on an Android operating system. Particularly, this invention relates to a system and method that includes a layer of verification to ensure that the underlying Android middleware and the Linux kernel are prevented access from unauthorised application to sensitive information through the normal Inter-Process Communication (IPC) channels. More particularly, this invention relates to a system and method for ensuring that a check on the privilege granted on an application is performed prior to allowing access to the requested data.

Summary of the Prior Art

The Android operating system has the highest number of users. Inevitably, the number of Android applications also grows at a fast pace. It is convenient for users to download applications from the Android market such as Play Store. However, as the applications are mostly written by third parties, the users have to assume that the applications written by the third parties are trustworthy. Typically, some of these applications are likely to be malicious and may inject rootkits to compromise the underlying Android middleware and even the Linux kernel. These malicious applications can bypass the permission check mechanism and access the unauthorised data belonging to the users, which are considered sensitive information such as contacts, location traces, and online banking information. This sensitive information is of high value to spammers and identity thieves.

In the Android operating system, critical data is typically managed by a dedicated Android system application and is shared with authorised user application via Android's IPC mechanism. Therefore it is crucial to protect the sensitive information from being leaked through normal IPC channels.

An existing technology is TaintDroid which is an information tracking system for real-time privacy monitoring on smartphones. It uses dynamic taint analysis to monitor privacy sensitive information on smartphones. TaintDroid is able to monitor the data flow of the tainted data. However, TaintDroid cannot prevent the data from being leaked. Specifically, it can only inform the user after the data has been sent. Furthermore, it has the shortcoming of the dynamic taint analysis which suffers from taint explosion. Hence, it may not be advisable to be used on a smartphone which has limited resources as it may slow down the speed of the smartphone tremendously. In addition, it requires significant changes on the existing software.

Another existing technology is TrustZone. ARM TrustZone is a hardware security extension incorporated into recent ARM processors. With TrustZone, the processor can execute instructions in one of two possible security modes, the secure world and the normal world. The applications can execute the instructions related to the sensitive data in the secure world to protect the sensitive information. However, TrustZone offers coarse granularity isolation. Particularly, the instructions running in the Normal World cannot access memory and other resources attached to the Secure World. Such isolation is not suitable for IPC protection, where sensitive information flows between applications.

For the above reasons, those skilled in the art are constantly striving to come up with a system and method that prevents attack on the IPC channel.

Summary of the Invention

The above and other problems are solved and an advance in the art is made by systems and methods provided by embodiments in accordance with the invention. A first advantage of embodiments of systems and methods in accordance with the invention is that it prevents malicious applications from accessing unauthorised data through IPCs. A second advantage of embodiments of systems and methods in accordance with the invention is that it prevents both kernel and user spaces malware from accessing unauthorised sensitive information without rewriting the Linux kernel, middleware or applications in Android operating system. In particular, the combination of IPC protection and generic user space protection prevents sensitive information from leakage in Android operating system. A third advantage of embodiments of systems and methods in accordance with the invention is that the trust base is in the hypervisor running the hypervisor mode can be much smaller than the kernel. Therefore, it is less vulnerable to be compromised.

The above advantages are provided by embodiments of a system and a method in accordance with the invention operating in the following manner.

In accordance with embodiments of this invention, a method for monitoring an inter-process communication (IPC) privilege control between a first application to a second application in an operating system is provided. The operating system has an application framework layer having a user space and a kernel space. The operating system provided with a hypervisor configured to perform the method. The method is provided in the following manner. The method begins by monitoring a request received in a kernel space of the operating system wherein the request is being transmitted from the first application to the second application via the kernel space. An event based on the request is being determined. The method intercepts the request according to the determined event and encrypts the data prepared by the second application according to the determined event. The method then monitors a page fault from the first application. In response to receiving a page fault from the

first application, the method retrieves an access control policy and determines if the first application has privilege to access data from the second application based on the access control policy. If the method determines that the first application has privilege to access data from said second application, the method decrypts the data prepared by the second application and allow the first application to access the data.

In accordance with embodiments of this invention, the method further comprises reporting an error and terminating the request in response to the first application not having privilege to access data from the second application.

In accordance with embodiments of this invention, the step of determining an event from the request may be provided in the following manner. The method reads the request and determines whether the request involves use of shared memory. If the request does not involve use of shared memory, the method obtains data from the second application and encrypts the data according to a first process. If the request involves use of shared memory, the method obtains data from the second application and encrypts the data according to a second process.

In accordance with embodiments of this invention, the first process is provided in the following manner. The method sets the code segment for the first application to non-executable and the data segment other than the send and receive buffer to non-accessible. The method then monitors for an ioctl system call from the second application. In response to receiving an ioctl system call from said second application, the method intercepts the ioctl system call from the second application and locates the data based on parameters in the ioctl system call. The method then encrypts the data prepared by the second application.

In accordance with embodiments of this invention, the step of intercepting the ioctl system call from the second application in response to receiving an ioctl system call from the

second application further comprises determining a binder protocol in said ioctl system call equals to BC_REPLY.

In accordance with embodiments of this invention, the step of encrypting the data prepared by the second application further comprises storing a hash value derived from said encrypting said data in the hypervisor space.

In accordance with embodiments of this invention, the step of determining the first application has privilege to access data from the second application based on the access control policy further comprises verifying the first application is receiving the correct data from the second application in the following manner. The method computes a hash value of the data and compares the hash value with the hash value saved on the hypervisor memory space. If the hash value matches the hash value saved on the hypervisor memory space, the method decrypts the data prepared by the second application.

In accordance with embodiments of this invention, the second process is provided in the following manner. The method monitors for an mmap system call from the second application and intercepts the mmap system call from the second application in response to receiving an mmap system call from the second application. The method then monitors for a page fault at the second application. In response to receiving a page fault at the second application, the method obtains and saves address and size of a mapped region. The method then monitors for an ioctl system call from the second application. In response to receiving the ioctl system call from the second application, locates the data. The method then encrypts the data and subsequently monitors for an mmap system call from the first application. In response to receiving an mmap system call from the first application, sets code segment for the first application to non-executable.

In accordance with embodiments of this invention, the step of intercepting the mmap system call from the second application in response to receiving an mmap system call from the second application further comprises setting a code segment of the second application non-executable, saving a page table in a hypervisor memory space, and passing the mmap system call back to the kernel space.

In accordance with embodiments of this invention, the step of obtaining and saving address and size of a mapped region may be provided in the following manner. The method compares a page table with the page table stored on the hypervisor memory. The method then obtains the address and size of the mapped region based on comparing the page table with the page table stored on the hypervisor memory. The address and size of the mapped region are saved on the hypervisor memory space. The method then unlocks the second application.

In accordance with embodiments of this invention, the step of locating the data in response to receiving the ioctl system call from the second application comprises determining whether a binder protocol in the ioctl system call equals to BC_REPLY, and locating the data based on the address and size saved on the hypervisor memory space.

In accordance with embodiments of this invention, the step of encrypting the data further comprises storing a hash value derived from encrypting the data in the hypervisor space.

In accordance with embodiments of this invention, the method further comprises saving another page table in the hypervisor memory space and returning the mmap system call to the kernel space. In accordance with embodiments of this embodiment, the step of retrieving an access control policy in response to receiving a page fault from the first application may be provided in the following manner. The method compares a new page

table with the page table stored on the hypervisor memory space to obtain a new address and size of said mapped region. The method then compares the new address and size with the address and size stored on the hypervisor memory space. In response to the new address and size being the same as the address and size stored on the hypervisor memory space, the method retrieves the access control policy.

In accordance with embodiments of this invention, an operating system comprising an application framework layer having a user space and a kernel space; a hypervisor installed on said application framework; and a set of instructions executable by the hypervisor is provided. The set of instructions is similar to the method set forth above.

Brief Description of the Drawings

The above advantages and features in accordance with this invention are described in the following detailed description and are shown in the following drawings:

Figure 1 illustrating a block diagram of the user space and the kernel space of an Android operating system;

Figure 2 illustrating a block diagram of having a hypervisor in the hypervisor space intercepting requests from the user application and the server application in accordance with embodiments of the invention;

Figure 3 illustrating a flow diagram of the process performed by the hypervisor in accordance with an embodiment of the invention;

Figure 4 illustrating a flow diagram to further elaborate the processes of determining an event from a request in accordance with an embodiment of the invention;

Figure 5 illustrating a flow diagram to further elaborate the first process for obtaining data from the server application and encrypting the data in accordance with an embodiment of the invention;

Figure 6 illustrating a flow diagram to further elaborate the second process for obtaining data from the server application and encrypting the data in accordance with an embodiment of the invention;

Figure 7 illustrating a timing diagram of a normal binder IPC procedure;

Figure 8 illustrating a timing diagram of a binder IPC procedure incorporating the processes executable by a hypervisor in accordance with embodiments of the invention;

Figure 9 illustrating a timing diagram of an Ashmem IPC procedure; and

Figures 10a and 10b illustrating a timing diagram of an Ashmem IPC procedure incorporating the processes executable by a hypervisor in accordance with embodiments of the invention.

Detailed Description

This invention relates to a system and method for preventing unauthorised access to sensitive information stored on an Android operating system. Particularly, this invention relates to a system and method that includes a layer of verification to ensure that the underlying Android middleware and the Linux kernel are prevented access from unauthorised application to sensitive information through the normal Inter-Process Communication (IPC) channels. More particularly, this invention relates to a system and method for ensuring that a check on the privilege granted on an application is performed prior to allowing access to the requested data.

Figure 1 illustrates normal data flow 100 between user applications and system applications in the user space 105. As shown, IPC is the primary channel for a user application to access system data such as location and contacts through the kernel space 110. A malicious user application may access sensitive information by exploiting vulnerabilities of the normal IPC channels or by compromising the underlying Android middleware and the Linux kernel. Android offers two methods for IPC transferring of data, namely, Binder mechanism and Android shared memory (Ashmem). One skilled in the art will recognise that figure 1 is only meant to show a portion of an Android framework that is relevant to the invention and other portions of the Android framework are omitted for brevity.

It is proposed that a virtualization-based approach which leverages on the virtualization extension on ARM be provided to control the information flow through normal IPC mechanism in Android. Particularly, a bare metal hypervisor running in HYP mode of the ARM platform is provided. Figure 2 illustrates the adding of a hypervisor 210 residing in the hypervisor space 120. The use of the bare metal hypervisor is as follows:

1. The hypervisor protects critical application data against kernel access by setting those space buffers inaccessible when the kernel is running.
2. The hypervisor intercepts exceptions that traps into the kernel and sets the application's code segment as non-executable as shown by arrows 220 and 230.
3. The hypervisor encrypts the server's IPC data before the kernel copies it to kernel space.
4. The client obtains the encrypted data and the hypervisor checks the client's access rights and decrypts the data if the client is authorised.

Two key techniques related to ARM's virtualization are used. The first technique is system call interception. In an ARM architecture, the application uses Supervisor Call (SVC) instruction as a request to the operating system (OS) to provide a service. With the virtualisation extension on ARM, the hypervisor sets the register HCR.TGE to 1 so that SVC instructions will trigger SVC exceptions which are caught by the hypervisor. This mechanism is used to intercept system calls in Android. Once getting this exception, the hypervisor can obtain the system call number in general register R7, and look it up in the system call table.

The second technique is to intercept mode switch from kernel user. This type of switch normally does not generate any exception and therefore the hypervisor may not catch this. To overcome this, a page fault is introduced for this switch. When the control is trapped to the kernel, the hypervisor sets the application code segment to non-executable by properly setting the corresponding page table entries (PTEs) in the Stage Two page tables. As a result, once the control returns to user mode, there will be a page fault because of the non-executable code segment. The hypervisor can intercept this page fault as a stage 2 memory management unit (MMU) fault.

One skilled in the art will recognise that the bare metal hypervisor may be replaced with a hypervisor running within an operating system environment without departing from the invention. Further, the hypervisor may be a software, firmware or hardware configuration.

Figure 3 illustrates a flow diagram of process 300 performed by the hypervisor 210 in accordance with an embodiment of this invention. Process 300 begins in step 310 to monitor for a request for IPC over the kernel space from a client application to a server application. Particularly, process 300 monitors for a system call in the request for IPC such as input/output control (ioctl), map Binder memory (mmap), or etc.

In step 320, upon detecting a request for IPC over the kernel space, process 300 determines an event from the request. The event may be read, write, etc. Further, the event may be a system call using shared memory or otherwise. Particularly, process 300 checks the "type" file in the data structure of the system call. If "type==Binder_type_FD" is present in the data structure of the system call, this means that the system call is for a request using shared memory. In step 330, process 300 intercepts the request according to the determined event. The event may be a request for data from the server application through the binder or shared memory. The data that is prepared by the server application is encrypted by the hypervisor before transmitting to the client application. Further details pertaining to step 330 will be described below with reference to figure 4.

In step 335, process 300 monitors for a page fault from the client application. Additionally, if the event involves the use shared memory, process 300 proceeds to step 337 prior to proceeding to step 340. In step 337, the hypervisor 210 compares the address and size of the mapped region. If the addresses and sizes are the same, the hypervisor 210 proceeds to step 340. Further details pertaining to step 337 will be described below with reference to figures 10a and 10b.

Process 300 then retrieves an access control policy associated to the client application in step 340. In step 350, process 300 determines whether the privilege accorded to the user application stored on the access control policy allows the client application access to data from the server application. If process 300 determines that the user application has access to data from the server application, process 300 proceeds to step 360 to decrypt the data. If process 300 determines that the client application does not have access to data from the server application, process 300 proceeds to step 370 to deny the user application access to the data from the server application.

Figure 4 illustrates a block diagram of a process 400. Process 400 further elaborates step 330 in process 300. Process 400 begins in step 405 to read the request.

In step 410 to determine an event from the request. If the event is a system call using shared memory, process 400 proceeds to step 430. Otherwise, process 400 proceeds to step 420.

In step 420, process 400 obtains data according to the request and subsequently encrypts the data based on a first process. Further details regarding the first process will be described below with reference to figure 5.

In step 430, process 400 obtains data according to the request and subsequently encrypts the data based on a second process. Further details regarding the second process will be described below with reference to figure 6.

Process 400 ends after step 420 or step 430.

The client application refers to an application requesting for a service from a target application while the server application is the target application that provides the service requested by the client application. One skilled in the art will recognise that other naming convention may be implemented and the use of client application and server application is not intended to be limited as such.

Figure 5 illustrates a block diagram of a process 500. Process 500 further elaborates the first process in step 420 of process 400. Process 500 begins in step 510 where hypervisor 210 sets the code segment of the client application non-executable and the data segment other than the send and receive buffer to non-accessible.

In step 520, hypervisor 210 monitors for an ioctl system call from the server application.

In step 530, hypervisor 210 intercepts the ioctl system call in response to receiving the ioctl system call from the server application. Hypervisor 210 further determines that the binder protocol equals to BC_REPLY and then locates the response data with the help of parameters in this ioctl system call.

In step 540, the hypervisor 210 encrypts the response data. Optionally in step 540, the hypervisor 210 may store a hash value derived from said encryption in the hypervisor space. The hash value can be used for verifying that the client application receives the correct data from the server application. The hypervisor 210 then passes the transfer to the binder driver and process 500 ends.

Figure 6 illustrates a block diagram of a process 600. Process 600 further elaborates the second process in step 430 of process 400. Process 600 begins in step 610 where hypervisor 210 monitors for an mmap system call from the server application.

In step 620, in response to an mmap system call from the server application, process 600 intercepts the mmap system call, sets the server application's code segment non-executable, saves a page table entries in the hypervisor memory space and subsequently passes the call back to the kernel.

In step 630, the hypervisor 210 monitors for a page fault at the server application as the code segment has been set to non-executable by the hypervisor in step 620.

In step 640, the hypervisor 210 compares the current page table with the page table stored on the hypervisor 210 in step 620 to obtain the address and size of the mapped region. The address and size of the mapped region are saved on the hypervisor space and the hypervisor 210 then unlocks the server application.

In step 650, the hypervisor 210 monitors for an ioctl system call from the server application.

In step 660, in response to receiving an ioctl system call from the server application, the hypervisor 210 determines whether the binder protocol equals to BC_REPLY and then locates the response data which is the mapped buffer with the help of address and size information save in step 640.

The hypervisor 210 then encrypts the buffer in the shared memory in step 670. Optionally in step 670, the hypervisor 210 may store a hash value in the hypervisor space. The hash value can be used for verifying that the client application receives the correct data from the server application. The hypervisor 210 then passes the transfer to the binder driver.

In step 680, the hypervisor 210 monitors for an mmap system call from the client application.

In step 690, in response to receiving a mmap system call from the client application, the hypervisor 210 sets the client's code segment non-executable and saves another page table entries in the hypervisor memory space. The hypervisor returns the mmap system call to the binder driver and process 600 ends.

In order to further illustrates process 300, a brief understanding of a normal Binder framework communication will be discussed with reference to figure 7 and a normal Ashmem framework communication will be discussed with reference to figure 9.

Figure 7 is a timing diagram of a process 700 illustrating the flow of the Binder framework communication. The Binder framework communication is a Client-Server model. The client application normally initiates a communication to the server application by sending out a request and waits for a response from server application.

Process 700 begins in step 710 where client application maps a memory region in kernel space to its user space for receiving data. The client application then transmits an input/output request (through the ioctl system call) to the binder driver for querying data and then sleeps.

In step 720, the binder driver receives the ioctl system call from the client application and locates the target server. Upon locating the target server, binder driver sends a request to the server application and wakes the server application if required.

In step 730, the server application receives the request from the binder driver and completes the task and issues an ioctl system call to the binder driver.

In step 740, the binder driver receives the ioctl system call from the server application and in response copies the server's data to kernel space. The binder driver then moves the data into the memory region mapped by the client application in step 710, modifies the pointer in client application pointing to it and wakes the client application.

In step 750, the client application fetches the data by using the returned pointer.

Figure 8 illustrates a timing diagram of a process 800 that incorporates process 300 performed by the hypervisor 210 into the process 700. Similar to process 700, process 800 begins with step 810 where client application maps a memory region in kernel space to its user space for receiving data. The client application then transmits an input/output request (through the ioctl system call) to the binder driver for querying data and then sleeps.

In step 820, the binder driver receives the ioctl system call from the client application and locates the target server. At this juncture, in step 830, hypervisor 210 determines an event based on the ioctl system call according to step 320 in process 300 and intercepts the ioctl system call according to step 330 in process 300. Since process 700 is a Binder framework communication, it does not use shared memory and hence encrypts data according to the first process in step 420 in process 400. Hypervisor 210 sets the code segment of the client application non-executable and the data segment other than the send and receive buffer to non-accessible according to step 510 in process 500. The binder driver continues with locating the target server. Upon determining the target server, the binder driver sends a request to the server application and wakes the server application if required in step 840.

In step 840, the server application receives the request from the binder driver and calls the required functions to prepare the response data. After completing the response, the server application issues an ioctl system call to the binder driver.

In step 850, the binder driver receives the `ioctl` system call from the server application and in response copies the responses data to kernel space. At this juncture, in step 860, hypervisor 210 intercepts the `ioctl` system call according to step 530 in process 500. In particular, the hypervisor 210 determines that the binder protocol equals to `BC_REPLY` and then locates the response data with the help of parameters in this request. The hypervisor 210 then encrypts the response data and saves the hash in the hypervisor space 120 according to step 540 in process 500. The binder driver then moves the encrypted data into the memory region mapped by the client application, modifies the pointer in client application pointing to it and wakes the client application.

In step 870, client application wakes up and due to segment code being set to non-executable, receives a page fault. At this juncture, in step 880, hypervisor 210 determines an event based on the page fault and intercepts the page fault according to step 335 in process 300. Hypervisor 210 retrieves the access control policy and checks the permissions of client application based on an access control list assigned during installation of the client application according to steps 340 and 350 in process 300. If hypervisor 210 determines that the client application has required permission to access the data, the hypervisor decrypts the data according to step 360 in process 360. Optionally, prior to decrypting the data, the hypervisor computes the hash value of the data prior and compares the hash value with the hash value saved in step 860. If both hash values match, the data is decrypted. This ensures that the client application is receiving the correct data from the server application. This is preferably required in cases where multiple requests are being transmitted by the client

application to various server applications, so that the system ensures that the data from the server application is meant for the client application prior to decrypting the data. After decryption, the hypervisor unlocks the memory space of the client application and returns. If hypervisor 210 determines that the client application does not have the required permission to access the data, the hypervisor 210 reports an error and terminates this IPC procedure according to step 370 in process 300.

In step 890, the client application accesses the response data.

Location data is widely regarded as sensitive since the access to a series of location data allows an adversary to track and trace the mobile device owner's geographic location information and other personal behaviour. In Android platforms, location data is transferred from the location manager service to applications through the binder driver. Hence, the process 500 can be applied to protect the location data from being shared unwillingly. When a user application is installed, the hypervisor gets the granted permissions by the Java Native Interface (JNI) call issued from the package manager. After the user application gets the response from the binder driver, the hypervisor checks if the user application has the required permission namely, `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. If the user application has these permissions, the hypervisor decrypts the location data for the user application to access. If not, the hypervisor reports an error and terminates the IPC procedure.

When the user application gets the location data, the user application may become a service provider to other applications. Hence, it is up to the system administrator to decide whether the binder protection mechanism to be applied to the user application. By protecting any application processing location data, the system administrator can prevent location data leakage.

Figure 9 is a timing diagram of a process 900 illustrating the flow of the Ashmem IPC. The Ashmem IPC is called Android anonymous shared memory subsystem based on the traditional shared memory mechanism in Linux kernel. The client application normally initiates a communication to the server application via the binder driver.

In step 910, client application sends a request (i.e. an ioctl system call) to the binder driver. In response, the binder driver locates the target server application and sends the request to the server application in step 920.

In step 930, server application receives the requests and calls an mmap system call to map a region in the kernel space buffer to its user space. After mapping the kernel space to its user space, the server application writes data to the kernel space from the user space. After the writing is completed, server application sends a file descriptor (fd) as a reference to the mapped region in the kernel space.

In step 940, the kernel maps a region in response to the call to map a region from the server application. The kernel then receives the data from user space of the server application.

In step 950, binder driver receives the file descriptor and in turn sends the file descriptor to the client application.

In step 960, the client application calls mmap system call to map the same kernel space buffer to its user space based on the file descriptor received from the binder driver. The client application then accesses the data from the shared memory in step 970.

Figures 10a and 10b illustrate a timing diagram of a process 1000 that incorporates process 300 into the process 900. Similar to process 900, process 1000 begins with step 1005 where client application sends a request (i.e. an ioctl system call) to the binder driver. In response, the binder driver locates the target server application and sends the request to the server application in step 1010. At this juncture prior to sending the request to the server application, in step 1012, hypervisor 210 determines an event based on the ioctl system call according to step 320 in process 300 and intercepts the ioctl system call according to step 330 in process 300. Since process 900 is an Ashmem IPC, it uses shared memory and hence encrypts data according to the first process in step 430 in process 400. The hypervisor 210 continues to monitor for an mmap system call from the server application.

In step 1015, server application receives the request and calls for an mmap system call to map a region in the kernel space buffer to its user space.

In step 1020, the kernel receives the mmap system call to map a region in the kernel space. At this juncture, in step 1025, hypervisor 210 intercepts the mmap system call according to step 620 in process 600. The hypervisor 210 sets the server's code segment

non-executable and saves a page table entries in the hypervisor memory space in step 1027.

After hypervisor passes the call back to the kernel, the kernel maps a region in response to the call to map a region from the server application and returns the mmap function to the server application in step 1027.

In step 1027, the server application activates a page fault as the code segment has been set to non-executable by the hypervisor in step 1025. At this juncture, in step 1030, the hypervisor 210 intercepts the page fault according to steps 630 and 640 in process 600. Particularly, the hypervisor 210 compares the current page table with the page table stored on the hypervisor 210 in step 1025 to obtain the address and size of the mapped region. The address and size of the mapped region are saved on the hypervisor space. The hypervisor 210 then unlocks the server application in step 1035.

In step 1035, the server application writes data to the kernel space from the user space in step 1040. After the writing is completed, server application issues an ioctl system call to send a file descriptor (fd) to the binder driver as a reference to the mapped region in the kernel space.

In step 1045, the binder driver receives ioctl system call from the server application. At this juncture, in step 1050, the hypervisor 210 intercepts the ioctl system call according to steps 650 and 600 in process 600. Particularly, the hypervisor 210 determines that the binder protocol equals to BC_REPLY and then locates the response data which is the mapped buffer with the help of address and size information saved in step 1025. The

hypervisor 210 then encrypts the buffer in Ashmem in step 1055 and store a hash value in the hypervisor space according to step 670 in process 600. The hypervisor 210 then passes the transfer to the binder driver in step 1060 which in turn sends the file descriptor to the client application in step 1065.

In step 1065, the client application calls mmap system call to map the same kernel space buffer to its user space based on the file descriptor received from the binder driver.

In step 1070, the kernel receives the mmap system call from the client application. At this juncture, in step 1075, hypervisor 210 intercepts the mmap system call according to steps 680 and 690 in process 600. The hypervisor 210 sets the client's code segment non-executable and saves another page table entries in the hypervisor memory space. After the hypervisor passes the control to the kernel, the latter returns from the mmap system call to the client application.

In step 1080, the client application returns from the mmap system call and activates a page fault as the code segment has been set to non-executable by the hypervisor in step 1075. At this juncture, in step 1085, the hypervisor 210 intercepts the page fault according to step 335 in process 300 and proceeds to step 337 in process 300. Particularly, the hypervisor 210 compares the current page table with the page table stored on the hypervisor 210 in step 1075 to obtain the address and size of the mapped region. The address and size of the mapped region are saved on the hypervisor space. The hypervisor 210 then compares current address and size with the address and size stored in step 1025. If both the

addresses and sizes are the same, the hypervisor 210 proceeds to retrieve the access control policy according to step 340 in process 300 check the permissions of client application based on an access control list assigned during installation of the client application according to step 350 in process 300. If the hypervisor 210 determines that the client application has the required permission to access the data, process 1000 decrypts the buffer according to step 360 in process 300. Optionally, prior to decrypting the data, the hypervisor computes the hash value of the data prior and compares the hash value with the hash value saved in step 1055. If both hash values match, the data is decrypted. This ensures that the client application is receiving the correct data from the server application. This is preferably required in cases where multiple requests are being transmitted by the client application to various server applications, so that the system ensures that the data from the server application is meant for the client application prior to decrypting the data. After decryption, the hypervisor unlocks the memory space of the client application and returns. If the hypervisor 210 determines that the client application does not have the required permission to access the data, it 210 reports an error and terminates this IPC procedure according to step 370 in process 300.

In step 1090, the client application reads the data.

Contacts stored in the mobile device reveal the mobile device user's social relations.

Unauthorized access to the contacts invades the user's privacy and may also be exploited by adversaries to launch social engineering attacks.

In Android platform, contact information is transferred from the contact service to applications through Ashmem. Hence process 1000 can be applied to protect the contact information from being shared unknowingly. When the application is being installed, the hypervisor gets the granted permission by the Java Native Interface (JNI) call issued from the package manager. After the user application maps the contact data from the shared memory to user application, the hypervisor checks if the user application has the required permission namely, READ_CONTACTS or WRITE_CONTACTS. If the user application has the permission, the hypervisor decrypts the location data for the user application to access. If not, the hypervisor reports an error and terminates the IPC procedure.

The above is a description of embodiments of a system and process in accordance with the present invention as set forth in the following claims. It is envisioned that others may and will design alternatives that fall within the scope of the following claims.

CLAIMS:

1. A method for monitoring an inter-process communication (IPC) privilege control between a first application to a second application in an operating system having an application framework layer having a user space and a kernel space, said operating system being provided with a hypervisor configured to perform said method, said method comprising:

monitoring a request received in a kernel space of said operating system, said request being transmitted from said first application to said second application via said kernel space;

determining an event from said request;

intercepting said request according to said determined event;

encrypting data prepared by said second application according to said determined event;

monitoring a page fault from said first application;

retrieving an access control policy in response to receiving a page fault from said first application;

determining said first application has privilege to access data from said second application based on said access control policy;

decrypting said data prepared by said second application in response to said first application having privilege to access data from said second application.

2. The method of claim 1 further comprising:

reporting an error and terminating request in response to said first application not having privilege to access data from said second application.

3. The method of claim 1 wherein the step of determining an event from said request comprises:

reading said request;

determining said request involves use of shared memory;

obtaining data from said second application and encrypting said data according to a first process in response to determining said request does not involve use of shared memory;

obtaining data from said second application and encrypting said data according to a second process in response to determining said request involves use of shared memory.

4. The method of claim 3 wherein said first process comprises:

setting code segment for said first application to non-executable and the data segment other than the send and receive buffer to non-accessible;

monitoring an ioctl system call from said second application;

intercepting said ioctl system call from said second application in response to receiving an ioctl system call from said second application;

locating said data based on parameters in said ioctl system call; and

encrypting said data prepared by said second application.

5. The method of claim 4 wherein said intercepting said ioctl system call from said second application in response to receiving an ioctl system call from said second application further comprises:

determining a binder protocol in said ioctl system call equals to BC_REPLY.

6. The method of claim 5 wherein said encrypting said data prepared by said second application further comprises:

storing a hash value derived from said encrypting said data in the hypervisor space.

7. The method of claim 6 wherein said determining said first application has privilege to access data from said second application based on said access control policy further comprises:

computing a hash value of said data;

comparing said hash value with said hash value saved on said hypervisor memory space;

decrypting said data prepared by said second application in further response to said hash value matches said hash value saved on said hypervisor memory space.

8. The method of claim 3 wherein said second process comprises:

monitoring a mmap system call from said second application;

intercepting said mmap system call from said second application in response to receiving a mmap system call from said second application;

monitoring a page fault at said second application;

obtaining and saving address and size of a mapped region in response to receiving a page fault at said second application;

monitoring an ioctl system call from said second application;

locating said data in response to receiving said ioctl system call from said second application;

encrypting said data;

monitoring a mmap system call from said first application; and

setting code segment for said first application to non-executable in response to receiving a mmap system call from said first application.

9. The method of claim 8 wherein said intercepting said mmap system call from said second application in response to receiving an mmap system call from said second application further comprises:

setting a code segment of said second application non-executable;
saving a page table in a hypervisor memory space; and
passing said mmap system call back to said kernel space.

10. The method of claim 9 wherein said obtaining and saving address and size of a mapped region comprises:

comparing a page table with said page table stored on said hypervisor memory;
obtaining said address and size of said mapped region based on said comparing said page table with said page table stored on said hypervisor memory;
saving said address and size of said mapped region on said hypervisor memory space; and
unlocking said second application.

11. The method of claim 10 wherein said locating said data in response to receiving said ioctl system call from said second application comprises:

determining whether a binder protocol in said ioctl system call equals to BC_REPLY; and
locating said data based on said address and size saved on said hypervisor memory space.

12. The method of claim 11 wherein said encrypting said data further comprises:

storing a hash value derived from said encrypting said data in the hypervisor space.

13. The method of claim 12 further comprising:

saving another page table in said hypervisor memory space; and
returning said mmap system call to the kernel space.

14. The method of claim 13 wherein said retrieving an access control policy in response to receiving a page fault from said first application further comprises:

comparing a new page table with the page table stored on said hypervisor memory space;

obtaining a new address and size of said mapped region.

comparing said new address and size with the address and size stored on said hypervisor memory space; and

retrieving said access control policy in response to said new address and size are same as the address and size stored on said hypervisor memory space.

15. The method of claim 14 wherein said determining said first application has privilege to access data from said second application based on said access control policy further comprises:

computing a hash value of said data;

comparing said hash value with said hash value saved on said hypervisor memory space;

decrypting said data prepared by said second application in further response to said hash value matches said hash value saved on said hypervisor memory space.

16. An operating system comprising an application framework layer having a user space, and a kernel space; and a hypervisor installed on said application framework having a set of instructions, said set of instructions executable by said hypervisor to:

monitoring a request received in said kernel space, said request being transmitted from a first application to a second application via said kernel space;

determining an event from said request;

intercepting said request according to said determined event;

encrypting data prepared by said second application according to said determined event;

monitoring a page fault from said first application;

retrieving an access control policy in response to receiving a page fault from said first application;

determining said first application has privilege to access data from said second application based on said access control policy;

decrypting said data prepared by said second application in response to said first application having privilege to access data from said second application.

17. The operating system of claim 16 wherein said set of instructions executable by said hypervisor further comprising instructions to:

reporting an error and terminating request in response to said first application not having privilege to access data from said second application.

18. The operating system of claim 16 wherein said instruction for determining an event from said request further comprises instructions to:

reading said request;

determining said request involves use of shared memory;

obtaining data from said second application and encrypting said data according to a first process in response to determining said request does not involve use of shared memory;

obtaining data from said second application and encrypting said data according to a second process in response to determining said request involves use of shared memory.

19. The operating system of claim 18 wherein said first process comprises instructions executable by said hypervisor to:

setting code segment for said first application to non-executable and the data segment other than the send and receive buffer to non-accessible;

monitoring an ioctl system call from said second application;

intercepting said ioctl system call from said second application in response to receiving an ioctl system call from said second application;

locating said data based on parameters in said ioctl system call; and

encrypting said data prepared by said second application.

20. The operating system of claim 19 wherein said instruction to intercepting said ioctl system call from said second application in response to receiving an ioctl system call from said second application further comprises instruction to:

determining a binder protocol in said ioctl system call equals to BC_REPLY.

21. The operating system of claim 20 wherein said instruction to encrypting said data prepared by said second application further comprises instruction to:

storing a hash value derived from said encrypting said data in the hypervisor space.

22. The operating system of claim 21 wherein said instruction to determining said first application has privilege to access data from said second application based on said access control policy further comprises instructions to:

computing a hash value of said data;

comparing said hash value with said hash value saved on said hypervisor memory space;

decrypting said data prepared by said second application in further response to said hash value matches said hash value saved on said hypervisor memory space.

23. The operating system of claim 18 wherein said second process comprises instructions executable by said hypervisor to:

monitoring a mmap system call from said second application;

intercepting said mmap system call from said second application in response to receiving a mmap system call from said second application;

monitoring a page fault at said second application;

obtaining and saving address and size of a mapped region in response to receiving a page fault at said second application;

monitoring an ioctl system call from said second application;

locating said data in response to receiving said ioctl system call from said second application;

encrypting said data;

monitoring a mmap system call from said first application; and
setting code segment for said first application to non-executable in response to receiving a mmap system call from said first application.

24. The operating system of claim 23 wherein said instruction to intercepting said mmap system call from said second application in response to receiving an mmap system call from said second application further comprises instructions to:

setting a code segment of said second application non-executable;
saving a page table in a hypervisor memory space; and
passing said mmap system call back to said kernel space.

25. The operating system of claim 24 wherein said instruction to obtaining and saving address and size of a mapped region comprises instructions to:

comparing a page table with said page table stored on said hypervisor memory;
obtaining said address and size of said mapped region based on said comparing said page table with said page table stored on said hypervisor memory;
saving said address and size of said mapped region on said hypervisor memory space; and
unlocking said second application.

26. The operating system of claim 25 wherein said instruction to locating said data in response to receiving said ioctl system call from said second application comprises instructions to:

determining whether a binder protocol in said ioctl system call equals to BC_REPLY; and

locating said data based on said address and size saved on said hypervisor memory space.

27. The operating system of claim 26 wherein said instruction to encrypting said data further comprises instructions to:

storing a hash value derived from said encrypting said data in the hypervisor space.

28. The operating system of claim 27 wherein said set of instructions executable by said hypervisor further comprising instructions to:

saving another page table in said hypervisor memory space; and
returning said mmap system call to the kernel space.

29. The operating system of claim 28 wherein said instruction to retrieving an access control policy in response to receiving a page fault from said first application further comprises instructions to:

comparing a new page table with the page table stored on said hypervisor memory space;

obtaining a new address and size of said mapped region.

comparing said new address and size with the address and size stored on said hypervisor memory space; and

retrieving said access control policy in response to said new address and size are same as the address and size stored on said hypervisor memory space.

30. The operating system of claim 29 wherein said instruction to determining said first application has privilege to access data from said second application based on said access control policy further comprises instructions to:

computing a hash value of said data;

comparing said hash value with said hash value saved on said hypervisor memory space;

decrypting said data prepared by said second application in further response to said hash value matches said hash value saved on said hypervisor memory space.