H-Binder: A Hardened Binder Framework on Android Systems

Dong Shen[1], Zhangkai Zhang[1], Xuhua Ding[2], Zhoujun Li[1(✉)], and Robert Deng[2]

[1] School of Computer Science and Engineering, Beihang University, Beijing 100191, China
{dongshen,lizj}@buaa.edu.cn, zhangzhangkai315@gmail.com
[2] School of Information Systems, Singapore Management University, Singapore, Singapore
{xhding,robertdeng}@smu.edu.sg

Abstract. The Binder framework is at the core of Android systems due to its fundamental role for interprocess communications. Applications use the Binder to perform high level tasks such as accessing location information. The importance of the Binder makes it an attractive target for attackers. Rootkits on Android platforms can arbitrarily access any Binder transaction data and therefore have system-wide security impact. In this paper, we propose *H-Binder* to secure the Binder IPC channel between two applications. It runs transparently with Android and COTS applications without making changes on their binaries. In this work, we design a bare-metal ARM hypervisor with a tiny code base at runtime. The hypervisor interposes on the main steps of a Binder transaction by leveraging ARM hardware virtualization techniques. It protects secrecy and integrity of the Binder transaction data. We have implemented a prototype of the H-Binder hypervisor and tested its performance. The experiment results show that H-Binder incurs an insignificant overhead to the applications.

Keywords: Android · Binder · Virtualization · ARM · System security · Hypervisor

1 Introduction

Android is designed with an object-oriented philosophy where a variety of built-in system applications (named as *managers* by Android) are abstracted as objects and tasked to manage system-wide resources, such as display and network I/O. User applications such as games and m-banking apps usually do not directly access system resources like their counterparts on a PC. To access system resources or to distribute data, applications heavily utilize interprocess communication (IPC) to remotely call other objects' methods. The Linux kernel in the Android system offers the Binder mechanism [28] as the main avenue for IPC transactions. Functionality-wise, the centerpiece of Android's Binder framework

R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 24–43, 2017.
DOI: 10.1007/978-3-319-59608-2_2

is its Binder driver residing in the kernel, although a large portion of code is in user space for marshaling the data.

Of the similar consequence of a rogue router attacking networking applications, a malware with kernel privilege can attack the Binder-based interprocess communications. Recent attacks [4,24,34] have demonstrated the feasibility and easiness of reading and manipulating the Binder transaction data, including keyboard inputs, SMS messages. Note that the Binder IPC is also sometimes used for an application's internal data exchanges. For instance, an m-banking app's user-interface thread may use the Binder to forward the transaction amount to its processing thread. The usage of the Binder could be transparent to the app developer. As shown in [4], an app using HTTPS for its Internet communications does not send out ciphertext directly. Instead, its plaintext data is firstly forwarded to Android's Network Manager through the Binder channel. In short, the corrupted Binder framework is a single point of failure of system security because the rootkit can easily read/write *all* applications' transaction data by accessing their memory buffers, without applying any sophisticated tricks. Most existing schemes of secure Binder transactions [5,30,35] focus on application-level protection, which cannot deal with rootkit attacks.

In this paper, we design a tiny trustworthy hypervisor called *H-Binder* with a small trusted computing base (TCB) to protect sensitive Binder transaction data against the rootkit on the ARM platform. H-Binder interposes on the Binder transactions to ensure the secrecy and integrity of the transaction data against the rootkit's malicious accesses.

H-Binder functions transparently to the Linux kernel, Android middleware and COTS applications without any modification on their binary codes. It can smoothly work in tandem with other virtualization based schemes [9,10,20,36] to harden the platform's security such as data protection in the kernel. To the best of our knowledge, H-Binder is the first work on Binder security against the rootkit. We have built a proof of concept of H-Binder and evaluated its performance. The results show that it is practical to use H-Binder on mobile phones to protect critical Binder transactions.

Organization. In the next section, we explain the background of Android Binder framework and recent virtualization techniques introduced to ARM processors. In Sect. 3, we present an overview of H-Binder including the security problem, the threat model and the challenges. We present two building blocks of H-Binder in Sect. 4 and the details of H-Binder workflow in Sect. 5. A report on H-Binder implementation and performance evaluation is in Sect. 6. We then present related work in Sect. 7 and a conclusion in Sect. 8.

2 Background

We explain below the background information of Android's Binder framework and the virtualization techniques on ARM platforms.

2.1 The Binder Framework

The Android platform is designed with an object oriented style with a wide range of system manager applications managing various resources and providing capabilities for user applications. The Binder IPC is the primary channel for user applications to interact and collaborate with system services or among themselves to carry out their intended tasks. For instance, a user application needs to interact with Android's LocationManager to access the mobile phone's location data.

A Binder transaction follows the traditional client-server model. In a typical scenario, it involves three parties: a thread of a resource manager app acting as a server, a thread of a user app as a client, and the Binder driver in the kernel. To facilitate applications to engage in a Binder transaction, Android's ServiceManager works as a registry service for user apps to look up a registered service provider. In a high level view, the client and server thread interact in a Binder transaction with the following steps. Note that the server has a pool of worker threads in sleeping mode waiting for processing the requests.

(1) To request the service from a service thread, the client thread issues a blocking *ioctl* system call through which it issues a command to the Binder driver.
(2) The Binder driver saves the client thread information, locates the intended server's sleeping worker thread, and wakes it up to handle the request.
(3) The wakened worker thread immediately processes the request and issues an *ioctl* system call to return the reply to the Binder driver.
(4) The Binder driver uses the information saved in step 2 to locate the client thread, wakes it up and passes the data to it.

One of the most critical data structures in the Binder framework is the `binder_transaction_data` (as depicted in Fig. 1) which is passed by the user-space threads to the Binder driver as one of the parameters of *ioctl*. The shadowed boxes are those bytes which are not changed by an honest kernel. In essence, `code` specifies the method for the receiving app to execute, while the buffer pointed to by `data.ptr.buffer` stores the parameters and objects needed by that remote method with length `data_size`. For ease of reference, we collectively call the bytes in the shadowed boxes as *transaction raw data* throughout the paper. As shown later, we are concerned with the integrity of the transaction raw data and secrecy of bytes pointed to by `data.ptr.buffer`.

It is necessary to highlight how a client application looks up and identifies the service application it intends to engage, because it is relevant to authentication issues of a Binder transaction. The lookup procedure is also a Binder transaction.

The `target` field in Fig. 1 is a *local* handler passed to the Binder driver to specify the intended destination. To look up a service application, the client sets `target` as 0 in a Binder request containing a text string. The Binder driver forwards this lookup request to Android's ServiceManager which then returns a handler to the client. Therefore, to engage with the service application, the client sets its `target` with the handler in its Binder request.

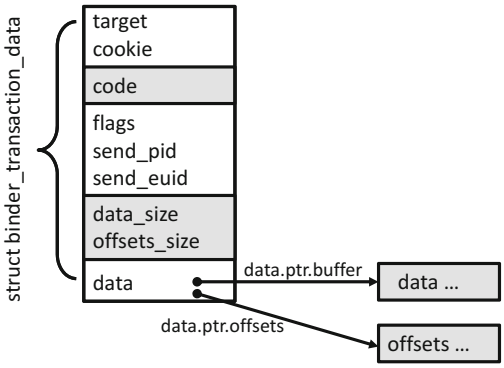


Fig. 1. Binder_transaction_data. The shadowed regions refer to the Binder transaction raw data which are the actual payload of a Binder communication.

2.2 Hardware Virtualization on ARM Processor

The recent ARMv7-A [1] architecture introduces hardware-assisted virtualization on ARM processors as an architectural extension. Different from x86 hardware virtualization where the CPU runs in the root mode (for the hypervisor) or the non-root mode (for the guest), ARM's hardware virtualization introduces a new privilege mode called the hyp mode for the hypervisor, which has a higher privilege level, i.e., Privilege Level 2 (or PL2 for short), than the svc mode used by the kernel.

When the CPU runs in the hyp mode, it accesses not only to those general registers and banked registers, but also to a set of new mode-specific registers including the Hyp Configuration Register (`HCR`) and the Hyp Syndrome Register (`HSR`). The former is used to configure the types of exceptions to be trapped into the hypervisor. For example, when `HCR`'s Trap General Exception (TGE) bit is set to 0×1, the supervisor call exception will be trapped to the hypervisor, which allows the hypervisor to intercept system calls from user space. `HSR` records the information about the exceptions trapping to the hypervisor. The Exception Class (EC) bits `HSR`[31:26] indicate the cause of the trap, e.g., 0×12 for a hypervisor call.

The `HVC` instruction can be used to enter into the hyp mode from the svc mode by raising a hypervisor call exception. After handling the call, the hypervisor uses the `ERET` instruction to switch the mode and returns to the next instruction following the `HVC` instruction. All exceptions trapping to the hyp mode use the exception vector at offset 0×14 of the hypervisor vector table.

Similar to memory virtualization on x86 platforms, ARM virtualization also supports two-stage address mapping for the virtual machine (VM). A virtual address (VA) in both usr and svc modes in the non-secure world is mapped to an intermediate physical address (IPA) by the Stage-1 page table managed by the kernel. Then, the IPA is mapped to the physical address (PA) by the Stage-2 page table managed by the hypervisor and is beyond the kernel's control. Therefore,

the hypervisor can control the attribute bits in the Stage-2 page table entries (PTEs) to regulate memory accesses from the VM for interception and isolation purposes.

3 Overview

This section presents an overview of our work. We begin with the explanation of the security problems.

3.1 The Problem Scope

Our aim is to protect the secrecy and integrity of sensitive transaction data transmitted between two Android applications through the Binder IPC. We consider the following adversary model and design restrictions.

Adversary Model. We consider rootkits whose attacks are targeted at the Linux kernel, e.g., reading and writing arbitrary kernel objects and manipulating the kernel's control flow. For instance, a rootkit can start its attack on Binder transactions by locating the global kernel object called `binder_context_mgr_node` which contains a pointer pointing to an array of buffers used for each Binder transaction.

Caveat. Out of several reasons, we do *not* consider rootkits that directly reads/writes an application's user space data and tampers with its control flow. Firstly, most rootkits do not target a specific application because it is less cost-effective to attack user space. It requires non-significant semantic knowledge of the victim application (e.g., the source code) while the damage is limited to the victim. Secondly, attacking on the kernel objects is much more catastrophic as it impacts all applications. Lastly, user-space protection techniques have been proposed on x86 platforms. Systems like Overshadow [9], InkTag [20], TrustPath [36], AppShield [11] can be exported to the ARM platform to cope with the user space security problem. The systems can run in tandem with H-Binder for the full protections. We do not attempt to re-invent the wheel.

Design Restriction. We restrict our design from modifying the existing Linux kernel, Android middleware or the applications. It is also refrained from changing the existing Binder framework, including the protocol and the syntax of relevant data objects. This is mainly due to compatibility concern.

Caveat. Under the design restriction above, the Binder data integrity protection only prevents rootkits from modifying the Binder data sent by applications. It does not deal with forgery. A rootkit can always inject its own Binder data to an application. Any countermeasure requires changes on either the application code or the Binder framework.

3.2 Challenges

The security problems described above present several challenges. Firstly, H-Binder should not incur significant overhead to the mobile phone. Since mobile

phones are power constrained, this requirement is especially more critical than a secure system on desktop computers. Therefore, the hypervisor should only interpose on system call for Binder transactions, instead of all system calls. Unfortunately, the current ARM virtualization technology does not have the ability to filter out system calls.

Secondly, the interposition on Binder transactions should be at the thread level rather than in the process level, because Android apps are multi-threaded. A process level interposition may stall all running threads no matter whether they are relevant to the security, and therefore downgrades the performance of the application.

Another challenge is the transparency and compatibility to the COTS Android system and applications. It precludes any changes to the present Binder framework, including the IPC protocol and the data structures. A tentative way to protect Binder IPC is to follow the SSL style on communication protection. Namely, the Binder client and server run a key exchange protocol (possibly mediated by a trusted party) and then exchange their encrypted Binder requests and replies. We do not opt for this method because it requires non-negligible changes not only on the Android runtime, but also the applications' code.

3.3 Our Contributions

The rest of the paper presents our proposed solution to the aforementioned problems. In a nutshell, our work makes the following contributions.

- We propose two novel techniques which can be used in hypervisors, i.e., selective system call issuance interception and thread-level system call return interception. These techniques can be used in H-Binder and in other hypervisors as well.
- We propose H-Binder, a security system running in the hyp mode that protects the Binder transactions to ensure the transaction raw data's integrity and secrecy. H-Binder is fully compatible and transparent to Android and its applications without requiring any changes on their codes.
- We build a prototype of H-Binder and evaluate the performance and compatibility with off-the-shelf applications.

4 H-Binder Building Blocks

In the following, we first introduce two novel techniques used as building blocks for H-Binder, i.e., selective interception for system call issuance and thread-level interception for system call return. We then present the details of H-Binder workflow.

4.1 Selective Interception for System Call Issuance

With ARM virtualization extensions, the system calls can be easily trapped to the hypervisor by setting `HCR.TGE` bit to 0×1. Nonetheless, it traps *all* system

calls from user space, which takes a significant performance toll on the whole system. To avoid unnecessary traps, H-Binder does not set HCR.TGE bit. Instead, it securely places a hook in the kernel which notifies the hypervisor on selected system calls according to the system call number and the issuing process. Unrelated calls are passed to the kernel directly.

Normal System Call Trap to Svc Mode. The system call trap from the usr mode to the svc mode is triggered by the SVC instruction. The exception vector addresses are stored in a vector page shown in Fig. 2 where the base address __vectors_start is set according to the 13th bit of the System Control Register (SCTLR), i.e. SCTLR.V bit.

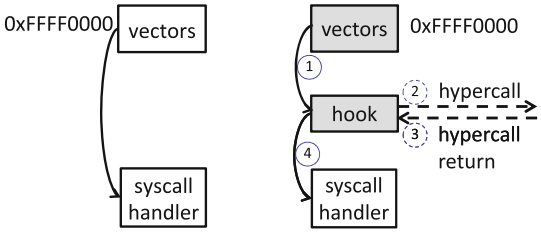
```
1  __vectors_start:
2      b vector_rst
3      b vector_und
4      ldr pc, __vectors_start+0x1000
5      b vector_pabt
6      ...
```

Fig. 2. Exception vectors stored in the vector page at either 0×00000000 or 0×FFFF0000 based on SCTLR.V bit.

An SVC instruction causes the Program Counter (PC) to jump to Line 4 in Fig. 2. As a result, the hardware loads PC with the content stored at __vectors_start+0×1000, which is exactly the address of the kernel's SVC handler. Thus, the control flow jumps to the SVC handler which then responds to the system call.

System Call Hook. We use a hook to filter out unrelated system calls as depicted in Fig. 3. When H-Binder protection starts (at secure boot up or triggered by a hypervisor call at runtime), the hypervisor writes the hook code into a reserved memory page in kernel space. It then places the entry address of the hook into __vectors_start+0×1000. As a result, whenever a system call is invoked in the user space, the hardware passes the control to the hook code instead of the kernel's handler. The hook code examines the system call number in R7 and the value of the Translation Table Base Register 0 (TTBR0)[1] which allows the hypervisor to check the identity of the issuer process. For instance, if it is *ioctl* issued by a concerned process, the hook issues a hypervisor call to the hypervisor. When the control returns from the hypervisor, the hook passes the control back to the original handler.

[1] In ARM architecture, TTBR0 points to the translation tables used by the current running user process and the Translation Table Base Register 1 (TTBR1) points to the translation tables used by the kernel.



(a) Normal system call control flow

(b) Hooked system call control flow

Fig. 3. Illustration of hooking the system call control flow where the shadowed boxes refer to pages that are read-only to the kernel and whose addresses cannot be changed. Step 2 and 3 are executed when the intercepted system call needs to be trapped.

Hook Protection. A rootkit may tamper with the physical addresses of the vector page or the hook page to bypass the interception. For this purpose, the hypervisor freezes the control flow path from the vector page to the hook code, in the sense that (1) the physical addresses of the vector page and the hook code page cannot be remapped by the kernel; (2) the code and data in both pages cannot be altered by the kernel. For this purpose, the hypervisor takes the following steps before placing the hook code page.

1. Set the Trap Virtual Memory (TVM) bit of `HCR` to 0×1 in order to intercept the kernel's write access to `SCTLR` and the Translation Table Base Register 1 (`TTBR1`) so that the hypervisor blocks all changes to `SCTLR.V` bit and `TTBR1`.
2. It traverses from the root of the Stage-1 page table pointed by `TTBR1` to the page table page pointing to the vector page which resides at 0×00000000 or $0\times$FFFF0000 depending on the `SCTLR.V` bit. Set all pages on this path as *read-only* by configuring the Stage-2 page table. In this way, any attempt to remap the physical address of the vector page is then trapped and blocked by the hypervisor.
3. It traverses from the root of the Stage-1 page table pointed by `TTBR1` to the page table page pointing to the hook code page. In the same fashion as the previous step, an update on any page on this path is not allowed if it affects the mapping of the hook code.
4. The hypervisor sets both the vector page and the hook code page as read-only so that the kernel cannot tamper with their contents.

Therefore, when a system call is invoked, the hardware always locates the vector page and the hook page at their predefined addresses. Moreover, since both pages are read-only, the correct hook code is executed as expected.

4.2 Thread-Level Interception for System Call Return

A thread expecting the Binder transaction data sleeps after issuing an *ioctl* system call. When the data arrives, the kernel completes the system call invocation by waking up the thread.

Unlike system call issuance, system call return does not throw out any exception. Hence we need to inject an exception in order to intercept the event. The challenge is how to generate a thread-specific event. It is a common practice for Android applications to use a dedicated worker thread for handling Binder transactions. Process-level interception affects all threads of the application and is not a good choice. For instance, setting a code page as non-executable introduces a page fault for all threads attempting to fetch instructions from this page, because the application's code (data) sections are shared among threads.

Our proposed method is based on the fact that threads do not share their stacks. The underlying idea is to manipulate the relevant thread's user space stack so that a stack operation after system call return is trapped to the hypervisor.

The first step is to map an empty physical memory page into the target application's heap. This page, named as *vault page* is to introduce the needed exception. The application is in fact not aware of its vault page and never uses it. The hypervisor sets the vault page *inaccessible* by configuring the Stage-2 page table, in order to block any access and to introduce the page fault for system call return. Note that the vault page must be mapped to the application's virtual address space. Otherwise, the exception it incurs is trapped to the kernel instead of the hypervisor.

Next, when the system call for receiving data is issued, the hypervisor intercepts it using the technique described previously. The hypervisor saves the thread's `SP_usr` into the hypervisor space, and then sets `SP_usr` to point to the application's vault page. The stack manipulation does not affect the kernel's execution because both the system call parameters and the return address are passed to the kernel through registers.

Lastly, when the system call returns upon data arrival, the thread returns from the svc mode to the usr mode. The user space stack is then used to resume user space execution. Since `SP_usr` points to the inaccessible page, a stack popup operation triggers a page fault exception and is trapped to the hypervisor.

5 The H-Binder Workflow

To facilitate the description of H-Binder, we present the basic idea and details of H-Binder.

5.1 The Approach

While it is straightforward to encipher the Binder raw data for a sending application, it is challenging to perform decryption securely. Without a rigorous checking of the recipient's identity, the improper decryption may reveal the plaintext to an imposter application. Therefore, the issues of data confidentiality and entity authentication are mingled together. It is difficult to authenticate the recipient thread because of the semantic gap faced by the hypervisor. The actual destination of the Binder transaction data is determined by the Binder driver at

runtime, instead of by the user threads. A rootkit can tamper with the data used by the Binder driver, and as a result, the driver delivers the transaction data to an imposter.

In a nutshell, the H-Binder scheme uses the building blocks introduced in Sect. 4 to interpose on each of the four Binder steps. After a step is intercepted, the hypervisor either saves or restores the data, depending whether it is to send or to receive the data. Nonetheless, the interception is transaction-agnostic in the sense that the intercepted data does not exhibit its relation to other events or any specific Binder transaction. Hence, the hypervisor has to trace the Binder transaction data flows in order to restore the data properly, including the lookup transaction. In specific, when a client issues a Binder request, the hypervisor saves the data and replaces it with a random number as an ID which is different from the existing entries. When the request arrives at the server end, the corresponding client's request is restored by checking the received request's ID. Therefore, when the server's worker thread replies, the hypervisor knows exactly its intended destination. When the reply arrives at the client end, the hypervisor checks whether the present application is the intended thread.

5.2 Details

We elaborate the details of H-Binder by explaining its protection over a Binder transaction between a user app and a resource manager app.

Initialization. When a user app is launched, an untrusted kernel module allocates a vault page whose virtual address is passed to the hypervisor. The hypervisor configures the Stage-2 page table to set the vault page *inaccessible*. It saves into the hypervisor space a pair $\langle ttbr, addr \rangle$ representing `TTBR0` data and the vault page's physical address, respectively. This page is used to intercept system call return as described in Sect. 4.2 and save the Binder data.

The hypervisor also maintains a Service Table whose entries pair a service description with the `TTBR0` value of the corresponding system service application, e.g., LocationManager. For each user application, the hypervisor also maintains a Handler Table whose entries pair a handler with the corresponding service's `TTBR0`. A user application's Handler Table is initialized with an entry $\langle 0, ttbr^* \rangle$ where $ttbr^*$ is the `TTBR0` used by ServiceManager.

The hypervisor creates the *Transaction Table* shown in Table 1 to save data related to every Binder transactions such that each intercepted event can be linked to a Binder transaction. In this table, *ClientID* is set as the client application's `TTBR0` which points to the root of its page table. *SApp* identifies the server app by using its `TTBR0` value while *SThread* identifies the server's worker thread by using the virtual address of its stack base. *ReqID* and *AckID* save the ID of the request and reply as their respective identifiers. *State* records the present transaction states.

Table 1. The format of the transaction table

ClientID	SApp	SThread	ReqID	AckID	State
0×96206f40	0×960b4280	0×76ef2000	0×47aa6d75	0×b0aacdf4	2
...	...	...	...	...	...

Runtime. The H-Binder hypervisor interposes on all four steps of a Binder transaction by using the building blocks priorly described. The workflow of H-Binder proceeds in four phases as depicted in Fig. 4 wherein a user app requests data from a manager app through a Binder IPC channel.

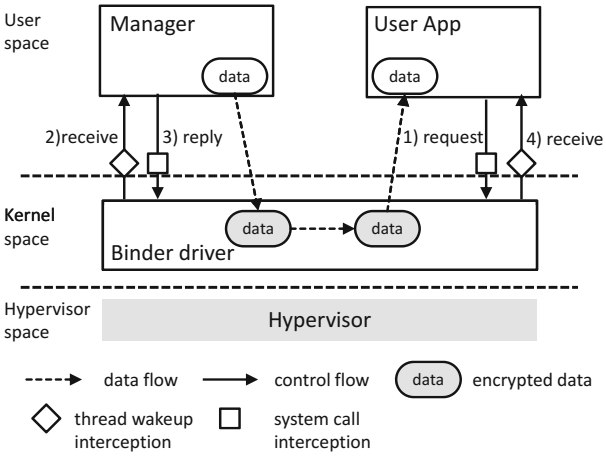


Fig. 4. Overview of H-Binder work flow.

Phase 1: User App Sending Request. Using techniques in Sect. 4.1, the hypervisor intercepts the user app's *ioctl* call right after it traps to the kernel. If the second parameter of *ioctl* is `BINDER_WRITE_READ`, the hypervisor locates the `binder_transaction_data` structure via the third parameter. Then it executes the following steps:

(1) It saves the request data in the client's vault page and replaces it with a random number which is different from the ReqID entries in the Transaction Table.
(2) It inserts to the Transaction Table a new record $\mathcal{T}$, where $\mathcal{T}$.ClientID is the current value of `TTBR0`; $\mathcal{T}$.ReqID is the generated random number in the first step; $\mathcal{T}$.State is set to 0 to indicate that a request is sent out. Based on the `target` of the intercepted Binder structure, the hypervisor looks up the client app's Handler Table to retrieve the corresponding `TTBR0` and assigns it to $\mathcal{T}$.SApp. (An error is returned if no matching record is found in the Handler Table.) All other fields of the new entry are set as `NULL`.

Phase 2: Manager Receiving Request. When the request is delivered by the Binder driver to the manager app, the manager's worker thread is wakened up to handle it. Using techniques in Sect. 4.2, the control is trapped to the hypervisor before the request is processed further by the thread. The hypervisor first checks data integrity and verifies whether the intercepted app is an imposter. It executes the following steps:

(1) It looks up the Transaction Table for a record with a matching record $\mathcal{T}$ such that $\mathcal{T}$.ReqID equals to the request data.
(2) If no matching record is found or $\mathcal{T}$.State is not 0, it drops this request and returns an error to the manager because the incoming request's integrity is compromised.
(3) To check whether the intercepted app is legitimate for receiving the request, it compares $\mathcal{T}$.SApp with the present `TTBR0`. If they do not match, an exception is thrown out. Otherwise, it loads the data from the client's vault page to recover its original Binder request, saves $\mathcal{T}$.SThread with `SP_usr`&0×FFFFE000 to record the worker thread's identity, and lastly set $\mathcal{T}$.State to 1 to indicate that the request is received by the server.

Phase 3: Manager Sending Reply. After handling the user app's request, the manager's worker thread returns a reply to the user app. Using the hook in Sect. 4.1, the thread's *ioctl* is trapped to the hypervisor which then performs the following steps:

(1) It looks up the Transaction Table for a matching record $\mathcal{T}$ such that $\mathcal{T}$.SThread equals to the present worker thread's stack base address. If no matching record is found, it drops the reply and returns an error indicating that the reply is not associated with any previously checked Binder request. Otherwise, it goes to the next step.
(2) It checks whether $\mathcal{T}$.State is 1. If not, it drops the reply and returns an error indicating inconsistent states. Otherwise, it goes to the next step.
(3) It saves the data pointed to by `data.ptr.buffer` in `Binder_transaction_data` structure in the vault page, and replaces it with a random number which is different from the AckID entries in the Transaction Table. It then updates $\mathcal{T}$ by assigning $\mathcal{T}$.AckID with the generated random number and setting $\mathcal{T}$.State to 2.

Phase 4: User App Receiving Reply. When the Binder driver delivers the manager's reply to the user app, it wakes up the user's blocked thread described in Phase 1. Using the techniques in Sect. 4.2, the control is trapped to the hypervisor before the thread processes the reply. Similar to Phase 2, the hypervisor checks both data integrity and the recipient app's authenticity before restoring the data. It runs the following steps:

(1) It looks up the Transaction Table to find a matching record $\mathcal{T}$ such that $\mathcal{T}$.AckID equals to the reply data. If no matching record is found, it discards the reply as its integrity is compromised and returns an error. Otherwise, it proceeds to the next step.

(2) It checks whether the present TTBR0 is the same as $\mathcal{T}$.ClientID and whether $\mathcal{T}$.State is 2. If either one fails, it returns an error because the present application is not the intended destination of the reply.
(3) It loads the data from the server's vault page, deletes $\mathcal{T}$ from the Transaction Table, and passes the control back to the user app. If $\mathcal{T}$.SApp refers to ServiceManager, the hypervisor obtains the handler from the data and updates the Handler Table of the client app. Note that if a suitable permission model is in place, the hypervisor can also enforce the access control policies before restoring data.

5.3 Security Analysis

We provide an informal analysis to explain how the Binder transaction is protected. The analysis begins with recipient authenticity which is the premise of proper Binder data protection.

Recipient Authenticity. Recipient authenticity is about whether a Binder transaction request/reply is delivered to the intended destination. For the flow from the client to the server, the hypervisor extracts the intended recipient's identity when the request is sent out and verifies the recipient's identity by checking its TTBR0 value when the request is delivered. Note that the rootkit's attack on an app's handler only leads to denial-of-service and cannot be used for impersonation.

For the return trip from the server to the client, the hypervisor verifies the recipient's identity by tracking the transactions flows using the Transaction Table. Specifically, for the matching record $\mathcal{T}$, $\mathcal{T}$.ReqID links Phase 1 and Phase 2, and $\mathcal{T}$.SThread links Phase 2 and 3, while $\mathcal{T}$.AckID links Phase 3 and 4. In this way, the hypervisor has sufficient knowledge to decide the intended recipient for a Binder reply from the server app.

Application Data Integrity and Secrecy. The rootkit's attack on the Binder data is neutralized by the data replacement used by the hypervisor. The sensitive data in the `Binder_transaction_data` structure is replaced before it is passed to kernel space in a system call issuance. As shown in Phase 2 and 4, a restoring is only performed after a successful authentication of the recipient app. Therefore, only the intended applications can access those data.

Binder data integrity is ensured by $\mathcal{T}$.ReqID and $\mathcal{T}$.AckID. A fraudulent Binder request is detected in Phase 2 and 4 before the recipient app processes it. The transaction's state stored in $\mathcal{T}$.State is used to detect replay attacks which show inconsistence.

Caveat. The security of H-Binder hypervisor can be protected by the hardware. The hyp mode is transparent to the system so that the rootkits don't know the existence of the hypervisor. Furthermore, the small TCB can reduce the probability of vulnerability.

6 Implementation and Performance Evaluation

We have implemented a prototype of H-Binder running in the hyp mode. The runtime TCB of H-Binder only consists of 1,813 SLOC (1,144 lines of C code and 669 lines of asm code).

The experimental environment is Linux Ubuntu 14.04 on a PC with an Intel(R) Core(TM) i7-4790 CPU @3.6 GHz processor and 16 GB main memory. In this platform, we run ARM FastModels [3] with FVP which emulates a mobile phone with a Cortex-A15 $\times$ 1 processor. The H-Binder hypervisor runs in the emulated phone as a bare-metal hypervisor. On top of the hypervisor, it runs Android 4.1 with a Linux kernel 3.9.0-rc3+. Due to the emulation, we do not measure the absolute time in our experiments. Instead, we use the CPU cycles to evaluate H-Binder performance.

6.1 Component Cost of H-Binder

The overall time overhead incurred by H-Binder is the sum of the CPU time for context switches due to the hypervisor interceptions or hypervisor calls and the CPU time spent by the hypervisor's execution. To evaluate the former cost, we measure the turnaround time of an empty hypcall which causes the CPU to enter to the hyp mode and return immediately. Our experiments show that the average cost for a round-trip mode switch cycle in a hypervisor call is about 96 cycles in our environment.

We also measure the CPU time spent in each of the four phases described in Sect. 5. The average CPU cycles spent in each of the phases are listed in Table 2 where the transaction involves 100 bytes returned by the server application. In general, the hypervisor spends 854 CPU cycles for involving in sending the Binder data, and spends 630 cycles for involving in receiving the Binder data.

Table 2. The number of CPU cycles spent in four phases of a Binder transaction, where the Binder request has 48 bytes and the Binder reply has 100 bytes

Phase 1	Phase 2	Phase 3	Phase 4
712	607	996	654

As shown in Sect. 5, a Binder IPC upon H-Binder involves 4 traps into the hypervisor. Therefore, the overall H-Binder cost for protecting a Binder based IPC is the sum of mode switch costs and the hypervisor's processing time, which amounts to 3,353 CPU cycles. For a mobile phone with 1 GHz CPU frequency, the time latency for one Binder transaction is about 3.4 µs, which is very tiny.

Note that the system call hook has negligible performance overhead as it only adds few instructions in the existing system call handler.

6.2 Application Level Performance Evaluation

To measure the performance impact of H-Binder on Android applications using the Binder, we measure the time spent for completing a task, e.g., to acquire the current location. We use the open-source application *RMaps*[2] as the client requesting for the mobile phone's location data. The program is instrumented to count the CPU cycles for invoking the LocationManager's `getLastKnownLocation()` function which runs Binder transactions with Android's LocationManager. We conduct the experiment in three different environments: the native Android, the Android running inside the host domain of KVM, and the Android running on the H-Binder hypervisor. Note that all three environments are hosted by ARM FastModels emulation. The results are presented in Table 3 below.

Table 3. Turnaround time (in CPU cycles) needed to obtain the location in different settings

	Android	KVM	H-Binder
Read location	68,577	69,929	77,344
Overhead	–	1,352	8,767

It shows that H-Binder incurs about 9,000 CPU cycles to get the location more than in Android. This relative overhead does not affect the whole application's performance because the absolute time delay is insignificant. For a mobile phone with 1 GHz CPU frequency, the time latency incurred by H-Binder is less than 9 µs. Note that the physical location is normally obtained in every one second or every three meters the device has moved. Therefore, supposing that the phone is on a running car moving with the speed of 15 m/s, the shortest time interval of location update is 67 ms. The latency of 9 µs is only around 0.01% compared to the time interval of location update. Hence the delay caused by H-Binder does not affect the location software's performance. The delay is also imperceptible for human users as the shortest time interval a human perceives is roughly between 50 ms to 150 ms [29].

Caveat. Our selective system call interception technique in Sect. 4.1 allows for performance isolation since those unrelated system calls are not intercepted and their performance is not affected by H-Binder. It can be further extended to select the critical applications and service to protect.

6.3 Time Cost for Different Sizes of Transferred Data

We then analyze how the size of the transferred data affects the overhead. We implement two Android applications using Binder IPC to transfer data

[2] https://github.com/ramnathv/rMaps.

between them. One app registers itself to Android's ServiceManager as the service providers while the other acts as a client. We vary the size of the data the server application returns and evaluate the turnaround time of getting the data, including the time spent for the Binder channel setup. Table 4 reports the experiment results in three different platforms.

Table 4. A whole binder transaction time in CPU cycles with different sizes of transferred data

# of Bytes	Android	With KVM	H-Binder
4	94,848	94,932	95,170 (0.3%)
8	95,070	95,178	95,781 (0.7%)
12	95,670	95,900	96,812 (1.2%)
20	96,070	96,318	96,960 (0.9%)
40	97,196	97,579	102,871 (5.8%)
80	100,349	100,743	107,118 (6.7%)
200	109,219	109,631	118,508 (8.5%)
400	120,875	121,353	130,496 (8.0%)

It shows that the time cost grows with the size growing. The main overhead is incurred by H-Binder's protection. As the size of data is not very large when the data is transferred in the Binder directly, the overhead of H-Binder in a whole Binder transaction will be less than 9%.

7 Related Work

Xen [6] is one of the earliest open source hypervisors initially developed for x86 platforms. Based on the Xen hypervisor, Hwang et al. proposed and implemented Xen-on-ARM [22] for the ARM architecture. Xen-on-ARM is a paravirtualization hypervisor and requires modifications to the kernel, as it is built on ARMv4/v5 which does not offer virtualization extension. From ARMv7 onwards, virtualization extension was introduced to support hardware virtualization on ARM architecture [1]. The first hypervisor using ARM virtualization extension was proposed in [31]. EmbeddedXen presents a new virtualization framework tailored to various ARM-based embedded systems [27]. ARMvisor [15] provides system virtualization for ARM, and KVM/ARM [13] is the first full system ARM virtualization solution that can run unmodified operating system on ARM multicore hardware. KVM/ARM has been integrated to the Linux kernel as a Linux ARM hypervisor.

H-Binder addresses the security of the Binder framework. A brief study on the technical details of Binder mechanism and its security weaknesses was described in [26]. More recent attacks [4] presented in the Black Hat conference further

demonstrated the cruciality of Binder security. It is shown in [4] that a malware which controls the Binder framework by attacking the *ioctl* system call can access and manipulate a variety of sensitive data, including keystrokes, in-app data, and SMS messages. ComDroid [12] proposes a tool to detect the vulnerabilities in Binder transaction, but it can't provide a runtime protection. AppFence [21] is built based on TaintDroid [16], using dynamic taint analysis to track the spread of the taint data. H-Binder can combine with this method to get stronger ability. However, TaintDroid can only protect the sensitive data while H-Binder can also protect the RPC with the Binder transactions.

The protection of Binder transactions has a direct impact on Android's access control mechanism. Some fine-grained access control mechanism [8,25,32] are proposed for diverse security and privacy policies. Android has a systematic permission model to control how applications access sensitive devices and data stores [33]. Most sensitive resource accesses are through the Binder framework where the data is returned or through call-backs by the resource manager app. The manager app typically checks the permission of the requesting apps before offering the service. Nonetheless, malicious apps without proper permissions may bypass the permission check by launching the permission re-delegation attacks [19]. In [17], Felt et al. analyzed different kinds of permission re-delegation attacks and proposed some possible ways to address this problem. Their method is to reduce the privileges of callee. In [7], Bugiel et al. also proposed a solution for a system-centric and policy-driven runtime monitoring of communication channels between applications at multiple layers in the inspiration of QUIRE [14] and a tool called Woodpecker [18] is developed to employ inter-procedural data flow analysis. The ways above are faced to the usr mode of Android. If the binder driver is hijacked by attackers, two data buffers will be changed which will lead to the leakage of some sensitive data. While H-Binder is faced to untrusted kernel, the encryption will keep the security of the sensitive information.

In a broader sense, H-Binder is related to Android's malware defense. CopperDroid [30] and VetDroid [35] leverage system call analysis and Binder transaction analysis to detect the application behavior. While Scippa [5] uses a call chain to get provenance information to implement the defense of the attack in Binder transaction. Cells [2] provides a virtualization architecture for enabling multiple virtual smartphones to run in an isolated secure manner. Nonetheless, neither of these systems can deal with kernel space attacks. To the attacks towards kernel space, many of them are against kernel interfaces like system call interface [23]. Attackers will hijack the system call handlers to let the kernel execute the attackers' instructions. H-Binder may not block all these attacks, but it can protect the sensitive data from being leaked as the data will be encrypted before entering the svc mode.

8 Conclusion

We have proposed H-Binder which leverages the recent ARM hardware virtualization techniques to secure Binder transactions in Android platforms. H-Binder

ensures secrecy and integrity of the sensitive data transported between two application threads interacting via Binder IPC. The H-Binder hypervisor intercepts the critical system calls from target applications and protects their data by using replacement techniques against attacks from rootkit. We have implemented a prototype of H-Binder on ARM FastModels. Our experiments show that the overhead incurred by H-Binder is not significant. Our future work is to prevent malicious code residing in the Android framework from attacking Binder transactions.

Acknowledgment. This research work is supported in part by the Singapore National Research Foundation under the NCR Award Number NRF2014NCR-NCR001-012, the National Natural Science Foundation of China under grants (Nos. 61170189, 61370126, 61672081), the National High Technology Research and Development Program of China under grant No. 2015AA016004, and Beijing Advanced Innovation Center for Imaging Technology (No. BAICIT-2016001).

References

1. Architecture Reference Manual (ARMv7-A and ARMv7-R edition). ARM DDI C (2008)
2. Andrus, J., Dall, C., Hof, A.V., Laadan, O., Nieh, J.: Cells: a virtual mobile smartphone architecture. In: 23rd ACM Symposium on Operating Systems Principles, pp. 173–187. ACM (2011)
3. Fast Models - ARM. http://www.arm.com/products/tools/models/fast-models/
4. Artenstein, N., Revivo, I.: Man in the Binder: He Who Controls IPC, Controls the Droid. Black Hat (2014)
5. Backes, M., Bugiel, S., Gerling, S.: Scippa: system-centric IPC provenance on android. In: 30th Annual Computer Security Applications Conference, pp. 36–45. ACM (2014)
6. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., et al.: Xen and the art of virtualization. ACM SIGOPS Oper. Syst. Rev. **37**(5), 164–177 (2003)
7. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on android. In: 19th Annual Network and Distributed System Security Symposium, pp. 346–360 (2012)
8. Bugiel, S., Heuser, S., Sadeghi, A.R.: Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In: 22nd USENIX Security Symposium, pp. 131–146 (2013)
9. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., et al.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. ACM SIGPLAN Not. **36**(1), 2–13 (2008)
10. Cheng, Y., Ding, X., Deng, R.H.: DriverGuard: a fine-grained protection on I/O flows. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 227–244. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23822-2_13
11. Cheng, Y., Ding, X., Deng, R.H.: Efficient virtualization-based application protection against untrusted operating system. In: 10th ACM Symposium on Information, Computer and Communications Security, pp. 345–356. ACM (2015)
12. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: 9th International Conference on Mobile Systems, Applications, and Services, pp. 239–252. ACM (2011)

13. Dall, C., Nieh, J.: KVM/ARM: the design and implementation of the linux ARM hypervisor. ACM SIGPLAN Not. **49**(4), 333–348. ACM (2014)
14. Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: QUIRE: Lightweight Provenance for Smart Phone Operating Systems. USENIX Security Symposium (2011)
15. Ding, J.H., Lin, C.J., Chang, P.H., Tsang, C.H., Hsu, W.C., Chung, Y.C.: ARMvisor: system virtualization for ARM. In: Proceedings of the Ottawa Linux Symposium, pp. 93–107 (2012)
16. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., et al.: Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. ACM Trans. Comput. Syst. **32**(2), 99–106 (2014)
17. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission redelegation: attacks and defenses. USENIX Secur. Symp. **6**, 12–16 (2011)
18. Grace, M.C., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock android smartphones. In: 19th Annual Network and Distributed System Security Symposium (2012)
19. Hardy, N.: The confused deputy: (or Why capabilities might have been invented). ACM SIGOPS Oper. Syst. Rev. **22**(4), 36–38 (1988)
20. Hofmann, O.S., Kim, S., Dunn, A.M., Lee, M.Z., Witchel, E.: Inktag: secure applications on an untrusted operating system. ACM SIGARCH Comput. Archit. News **41**(1), 265–278. ACM (2013)
21. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: 18th ACM Conference on Computer and Communications Security, pp. 639–652. ACM (2011)
22. Hwang, J.Y., Suh, S.B., Heo, S.K., Park, C.J., Ryu, J.M., Park, S.Y., Kim, C.R.: Xen on ARM: system virtualization using Xen hypervisor for ARM-based secure mobile phones. In: 5th IEEE Consumer Communications and Networking Conference, pp. 257–261. IEEE (2008)
23. Lee, H.C., Kim, C.H., Yi, J.H.: Experimenting with system and Libc call interception attacks on ARM-based linux kernel. In: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 631–632. ACM (2011)
24. Li, W.X., Wang, J.B., Mu, D.J., Yuan, Y.: Survey on Android Rootkit. Microprocessors (2011)
25. Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H.J., Cowan, C.: User-driven access control: rethinking permission granting in modern operating systems. In: 33rd IEEE Security and Privacy, pp. 224–238. IEEE (2012)
26. Rosa, T.: Android binder security note: on passing binder through another binder (2011)
27. Rossier, D.: EmbeddedXEN: A Revisited Architecture of the Xen Hypervisor to Support ARM-Based Embedded Virtualization. White Paper, Switzerland (2012)
28. Schreiber, T.: Android binder-android interprocess communication. Seminar thesis, Ruhr-Universität Bochum (2011)
29. Shneiderman, B.: Designing the User Interface: Strategies for Effective Human-Computer Interaction. Pearson Education, India (2010)
30. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: CopperDroid: automatic reconstruction of android malware behaviors. In: 22nd Annual Network and Distributed System Security Symposium (2015)
31. Varanasi, P., Heiser, G.: Hardware-supported virtualization on ARM. In: 2nd Asia-Pacific Workshop on Systems (2011)

32. Wang, Y., Hariharan, S., Zhao, C., Liu, J., Du, W.: Compac: enforce component-level access control in android. In: 4th ACM Conference on Data and Application Security and Privacy, pp. 25–36. ACM (2014)
33. Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Permission evolution in the android ecosystem. In: 28th Annual Computer Security Applications Conference, pp. 31–40. ACM (2012)
34. You, D.H., Noh, B.N.: Android platform based linux kernel rootkit. In: 6th International Conference on Malicious and Unwanted Software, pp. 79–87. IEEE (2011)
35. Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., et al.: Vetting undesirable behaviors in android apps with permission use analysis. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 611–622. ACM (2013)
36. Zhou, Z., Gligor, V.D., Newsome, J., McCune, J.M.: Building verifiable trusted path on commodity x86 computers. In: 33rd IEEE Symposium on Security and Privacy, pp. 616–630. IEEE (2012)