1-2018

# TinyVisor: An extensible secure framework on Android platforms

Dong SHEN
*Beijing University of Aeronautics and Astronautics (Beihang University)*

Zhoujun LI
*Beijing University of Aeronautics and Astronautics (Beihang University)*

Xiaojing SU
*Chinese Academy of Sciences*

Jinxin MA
*China Information Technology Security Evaluation Center*

DENG, Robert H.
*Singapore Management University*, robertdeng@smu.edu.sg

**DOI:** https://doi.org/10.1016/j.cose.2017.09.006

## Citation

SHEN, Dong; LI, Zhoujun; SU, Xiaojing; MA, Jinxin; and DENG, Robert H.. TinyVisor: An extensible secure framework on Android platforms. (2018). *Computers and Security*. 72, 145-162. Research Collection School Of Information Systems.
**Available at:** https://ink.library.smu.edu.sg/sis_research/3844

Published in Computers & Security, 2018
January, Volume 72, Pages 145-162
http://doi.org/10.1016/j.cose.2017.09.006

# TinyVisor: An extensible secure framework on android platforms

*Dong Shen* [a], *Zhoujun Li* [a,*], *Xiaojing Su* [b], *Jinxin Ma* [c], *Robert Deng* [d]

[a] *School of Computer Science and Engineering, Beihang University, Beijing 100191, PR China*
[b] *Key Laboratory of Microelectronics Devices & Integrated Technology, Institute of Microelectronics of Chinese Academy of Sciences, Beijing 100029, PR China*
[c] *China Information Technology Security Evaluation Center, Beijing 100085, PR China*
[d] *School of Information Systems, Singapore Management University, Republic of Singapore*

*Keywords:*
Android
Virtualization
ARM
Hypervisor
System Security

A B S T R A C T

As the utilization of mobile platform keeps growing, the security issue of mobile platform becomes a serious threat to user privacy. The current security measures mainly focus on the application level and the framework level, with little protection on the kernel. Virtualization technologies have been used in x86 platforms to protect the security of the kernel. With a higher privilege than the guest operating system, the hypervisor can effectively detect and defend against the malicious activity inside the guest kernel. In this paper, we build a hypervisor framework called TinyVisor leveraging the ARM virtualization extensions to protect the guest system security. The framework is transparent to the guest operating system and applications without any code modification. On top of the framework, we propose a secure module called H-Binder to protect the integrity and secrecy of the Binder transaction data in Android system. We implement the prototype of TinyVisor with the H-Binder module and evaluate the performance. The experiment results show non-significant performance loss.

## 1. Introduction

In recent years, more than 95 percent smartphones, numerous tablets and embedded devices are using ARM processors. ARM processors have occupied most of the market share on the strength of low energy consumption (Aroca and Gonçalves, 2012; Ou et al., 2012; Smith, 2008). At the same time, many malwares have focused on the operating systems of mobile platforms with the growing of the user group, such as Android (Arzt et al., 2014; Mulliner et al., 2014; Wen-Xin et al.; You and Noh, 2011) and iOS (Spaulding et al., 2012). However, the security protection measures on ARM platforms are less mature than that on x86 platforms. The security study on mobile platforms are mainly focused on the application and framework level at present. The attacks focusing on the system kernel which can gain root privilege have great harmfulness, nevertheless, there is no effective protection methods now. Therefore, how to enhance the security of the system kernel on mobile platforms has become an important problem.

Virtualization technology is a widely-used technology, which has become a main technology in the cloud environment on x86 platforms (Uhlig et al., 2005). Using the virtualization technology, multiple guest operating systems can run on the same physical hardware and keep isolation with each other through the interfaces offered by the hypervisor. Besides, virtualization can be used in the protection of system security. As the hypervisor has higher privilege level than the guest operating system (OS), the hypervisor can discover the malicious behaviours in the guest OS and protect the OS from being compromised. Systems like Overshadow (Chen et al., 2010), InkTag (Hofmann et al., 2013), TrustPath (Zhou et al., 2012), AppShield (Cheng et al., 2015) are all the important researches in which the virtualization technology is used to protect the system security on x86 platforms.

Inspired by the researches on x86 platforms, many researchers start to leverage virtualization technology to protect the system security on ARM platforms. In 2011, ARM released ARM-v7 CPU processor which added virtualization extensions and security extensions (Architecture Reference Manual (ARMv7-A and ARMv7-R edition), 2008). Thereafter, the hypervisors like Xen (XenProject, 2013), KVM/ARM (Dall and Nieh, 2014) and OKL4 hypervisor (Varanasi, 2010) were proposed using the virtualization extensions on ARM. However, these hypervisors focused on multiple systems, which added the trusted computing base (TCB). Furthermore, Xen and KVM/ARM need a host system to handle some important interrupts, which leads that the TCB contains the code base of the host system. The bigger the TCB is, the more vulnerable the system is. The TCB of OKL4 hypervisor is relatively small, but the structure is too simple to add any additional functions.

In this paper, we propose TinyVisor, an ARM-based hypervisor framework with a tiny TCB. TinyVisor offers a general hypervisor framework leveraging ARM virtualization extensions. Users can add virtualization functions as needed. TinyVisor focuses on the single system and does not have a host system, which can reduce the TCB efficiently. We have added a Binder transaction protection module called H-Binder and tested the performance of TinyVisor, which shows an insignificant overhead.

This paper is extended from our original work which has been presented at the *12th EAI International Conference on Security and Privacy in Communication Networks, SECURECOMM2016* (Shen et al., 2016). This paper structurally puts forward the hypervisor framework, while the conference paper only introduces the H-Binder module. Compared to the conference version, this paper systematically introduces the structure of TinyVisor, including booting, context switch and events handling. Meanwhile, we add the formalized representation and theoretical analysis in the H-Binder module of this paper. We use the set operation to present the workflow and analyze the security theoretically.

Broadly speaking, the main contributions of this paper are as follows.

- We design and implement a bare-metal ARM hypervisor called TinyVisor with a tiny code base (around 2000 SLOC) at runtime. TinyVisor is only targeted for one guest system and thus its TCB is considerably small. Meanwhile, TinyVisor is an extensible framework, to which other functional modules can be added.
- We propose two novel techniques which can be used in hypervisors, i.e., issuance interception of alternative supervisor call and thread level interception for supervisor call return. These methods can be applied to TinyVisor or any other hypervisors.
- We add a Binder transaction protection module called H-Binder into TinyVisor to protect the Binder transactions by ensuring the integrity and secrecy of data transmission. We also theoretically prove that the scheme can protect the integrity of Binder transaction data flow.
- We build a prototype of TinyVisor with H-Binder and evaluate the performance and compatibility with off-the-shelf applications. The result shows that the overhead is insignificant.

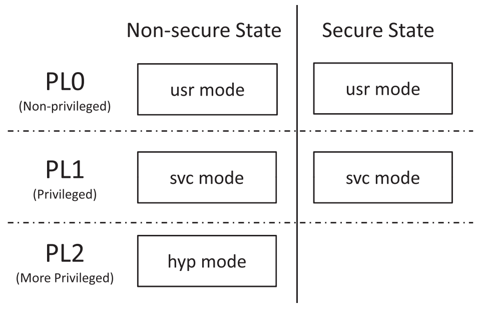


**Fig. 1 – Structure of ARM processor.**

In the next section, we explain the background of recent virtualization techniques introduced to ARM processors. In Section 3, we describe the design of TinyVisor. We present the novel techniques used in TinyVisor in Section 4 and a Binder protection module in Section 5. A report on TinyVisor's implementation and performance evaluation is in Section 6. We then discuss various issues in Section 7 and related work in Section 8 with a conclusion in Section 9.

## 2. Background

The relevant background of virtualization techniques on ARM platforms will be provided in the following content in order to make the proposed design of TinyVisor much easier to understand.

### 2.1. *Hardware virtualization on ARM processor*

The recent ARMv7-A architecture introduces virtualization extensions on ARM processors. The virtualization extension should be used in combination with security extensions and the large physical address extensions.

The ARM virtualization extensions are only applied in non-secure state. ARM introduces a new processor mode called *the hyp mode* which has a higher privilege level than the existing non-secure svc mode. Accordingly, the original guest OS without any modification and applications can still run in the existing svc mode and usr mode respectively. The structure of ARM processor with the virtualization extensions and the security extensions is depicted in Fig. 1.

There are two ways to enter the hyp mode. It is possible to use an `hvc` asm instruction to actively enter the hyp mode from the svc mode. Moreover, according to the configuration of relevant registers, some specific events which come from the usr mode or the svc mode can be trapped into the hyp mode. ARM virtualization extensions also have some banked

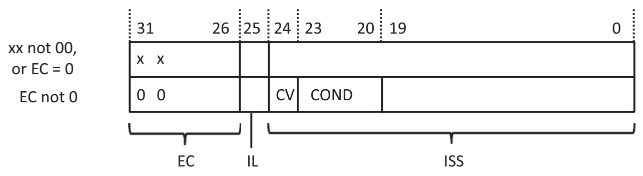


**Fig. 2 – The format of HSR.**

registers. Take the Stack Pointer (SP) as an example, it is `SP_usr` in the usr mode while it is `SP_hyp` in the hyp mode. Using banked registers has an edge on improving the system performance because it can avoid saving and loading the values of the registers at the time of mode switch.

### 2.1.1. *Configurable instruction trap*

The configuration provided by ARM virtualization extensions could help to decide whether an instruction is trapped into the hypervisor or not. The different bits in the Hyp Configuration Register (`HCR`) indicate different ways of the traps. For example, when the Trap General Exception (`TGE`) bit is set to 0x1, all the supervisor calls will be trapped into the hypervisor. Therefore, the hypervisor can intercept the chosen event by configuring the `HCR`, while other events will be handled as usual. After the hypervisor intercepts the event, the Hyp Syndrome Register (`HSR`) records the exception information. As shown in Fig. 2, `HSR[31:26]` are the Exception Class (`EC`) bits, which can directly illuminate the cause of the trap, e.g., 0x12 for a hypervisor call.

### 2.1.2. *Two-stage page table translation*

ARM virtualization extensions use two-stage page table translation to achieve better control over the guests' virtual memories. The first stage page table is maintained by the guest OS which translates the guest virtual address (VA) to the intermediate physical address (IPA). The second stage page table which translates the IPA to the physical address (PA) is maintained by the hypervisor and keeps transparent to the guest OS. Using this method, the memory access of the guest OS will be in the control of the hypervisor through the appropriate configuration of the property bits in the Stage-2 page table entries (PTEs).

### 2.1.3. *Virtual interrupt*

The virtual interrupt is introduced by ARM virtualization extensions for the avoidance of simulating interrupt controller because the complexity will increase significantly and it may be trapped into the hyp mode frequently. Virtual interrupt is supported by a new hardware component called virtual CPU (VCPU) interface, which can be mapped into the guest OS and used as the Generic Interrupt Controller (GIC) CPU interface. Therefore, this interface can be used by the guest OS in order to conform and clear the interrupt without being trapped into the hypervisor. The hypervisor still needs to simulate the interrupt distributor and trap all the guest accesses to the interrupt distributor. This action will not lead to performance degradation because the distributor is always accessed to register the driver for special interrupts and route them into special (virtual) CPU at booting time (or module-loading time).

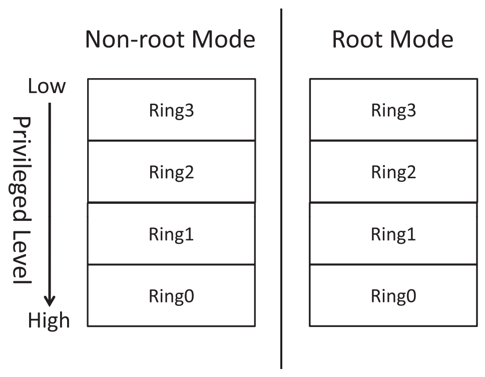


**Fig. 3 – Structure of x86 virtualization.**

## 2.2. *Comparison with x86 virtualization*

Before the release of ARM virtualization extensions, the x86 platform (Intel VT-x as an example) has already supported virtualization. The virtualization extensions on these two platforms are distinct due to the differences of the CPU architectures. The comparisons between these two virtualization technologies will be expanded from different perspectives below.

### 2.2.1. *Privileged mode*

The hypervisor always has a higher privilege than the operating system in both ARM and x86 virtualization techniques. In x86 virtualization, the hypervisor runs in the so-called *root* mode. On the other hand, the hypervisor runs in the hyp mode on ARM. However, the differences between them are fundamental. An x86 CPU in the root mode runs in a totally different fashion as in the non-root mode. It has a full set of new registers independent of the non-root mode and owns another set of privilege rings, from Ring 0 to Ring 3. In comparison, the hyp mode is just a privilege extension of the svc mode, in the same fashion as the svc mode extending the usr mode. The ARM hypervisor can directly access the registers used in the svc mode, while the x86 hypervisor can only access the registers used in the non-root mode via accessing a data structure. Comparing Fig. 3 with Fig. 1, the root mode in x86 is more similar with the security extensions in ARM.

### 2.2.2. *Virtual memory*

The virtualization extensions both in ARM and x86 introduce an additional page table translation for the purpose of memory virtualization. It is called the extended page tables (EPTs) in x86; and two-stage page table translation in ARM. They both translate the guest physical address (IPA in ARM) into the machine physical address (PA in ARM). The structural differences mainly show in the format of the page table. The EPT in x86 architecture uses existing page table format, while the two-stage page table in ARM uses a new page table format.

**Table 1 – Comparison between ARM virtualization and x86 virtualization.**

| Property | ARM | x86 |
|---|---|---|
| Privileged Mode | hyp mode | root mode |
| Virtual Memory | 2-stage PT | EPT |
| Guest Identification | TLB label | TLB label |
| VM Entry | directly | VMCS |
| Simulation Support | fast trap | - |
| I/O | - | secure DMA |
| Hyp Trap | 27 cycles | 600–800 cycles |

#### 2.2.3. *Virtual machine entry*

There is a memory mapped area named Virtual Machine Control State (VMCS) for one VM in Intel processors. To enter a VM, the hypervisor will set the registers and launch a VM-entry instruction, such that the CPU discontinues the current execution and switches into a different one. In ARM processors, entering into a VM is, to certain degree, a continuation of the hypervisor execution as the hypervisor only needs to set the Program Counter (PC) and perform a return.

#### 2.2.4. *Performance of hypervisor trap*

The hypervisor is often used to intercept events in the guest domain. Therefore, the cost of trapping to the hypervisor is crucial to the performance of the whole system. According to Dall and Nieh (2014), the context switch in ARM costs around 27 cycles, while the context switch in x86 costs between 600 to 800 cycles. However, the x86 CPU saves all registers used in the guest domain, while an ARM CPU only banks some of them. Therefore, the ARM hypervisor has to save those general registers if it needs to restore them.

#### 2.2.5. *Summary*

In summary, ARM virtualization extensions are similar to x86 in the aspect of design concepts. However, the differences in the implementation are mainly caused by the differences in the aspects of instruction sets and platforms. Table 1 summarizes the differences between ARM and x86 in virtualization extensions.

### 2.3. *TrustZone vs virtualization*

The security extensions (aka TrustZone) of ARM were released earlier than the virtualization extensions, which have been used in many security systems in practice and research. We make a comprehensive comparison between these two techniques.

#### 2.3.1. *Security*

To fairly compare the security strength between security extensions and virtualization extensions, we consider the scenario where the same code residing in the TrustZone and in the hyp mode, and examine how they can be attacked from a malicious kernel in the svc mode of the non-secure state.

For the code in the hyp mode, the access from the kernel is blocked by the Stage-2 page tables whose PTEs are under the control of the hypervisor and do not map any address used by the kernel to the physical memory occupied in the hypervisor space. Nonetheless, since the current ARM virtualization extensions do not support I/O virtualization, the malicious kernel may launch Directional Memory Access (DMA) attacks to invade the hypervisor space. For the code in the secure state, the access from the non-secure state is blocked by the memory controller regardless of the page tables in use.

#### 2.3.2. *Interception capability*

By their design rationales, virtualization offers a greater capability than TrustZone in terms of intercepting events occurring in the domain where the users run applications on top of the OS. HCR and Secure Configuration Register (SCR) shown in Fig. 4 are used by the hypervisor and the TrustZone to specify which events are to be intercepted, respectively.

Most guest events that can be intercepted by the TrustZone can also be intercepted by the hypervisor, though the TrustZone has the priority of intercepting the same event. Furthermore, the hypervisor can intercept some guest events which cannot be intercepted by the TrustZone, such as Stage-2 page fault. The limitation of TrustZone's interception capability makes it ill-suited for introspecting the domain and in-domain protection, albeit powerful in isolation. That is the reason why Hypervision (Azab et al., 2014) made remarkable changes on the OS.

#### 2.3.3. *Cost of switch*

The costs of switching to the secure state and the hypervisor affect the performance of respective systems. Paolino et al. (2015) made a comparison between them. The overhead of switching to the hyp mode is about 1400 CPU cycles, while the overhead of switching to the secure state is about 3700 CPU cycles. The overhead using hypervisor is about 62% less than using TrustZone.

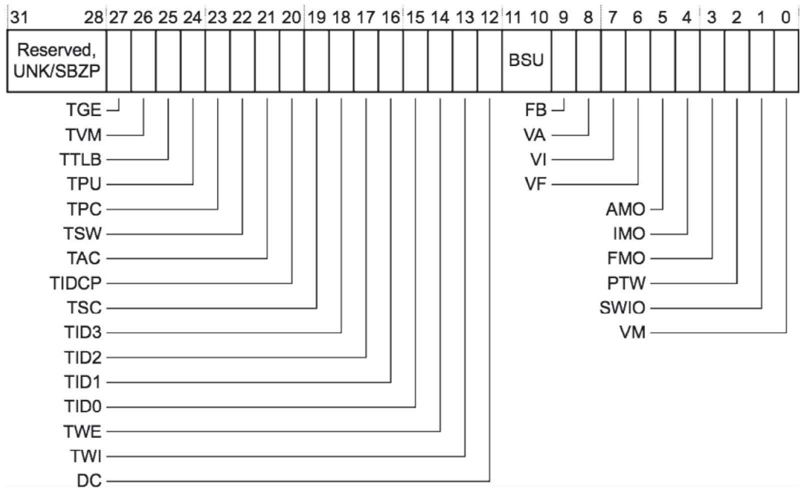
#### 2.3.4. *Summary*

To sum up, TrustZone can provide a more secure information storage environment while virtualization is more flexible and effective to handle and intercept the events. Therefore, combining advantages of the two techniques can offer more secure and effective protection for the system.

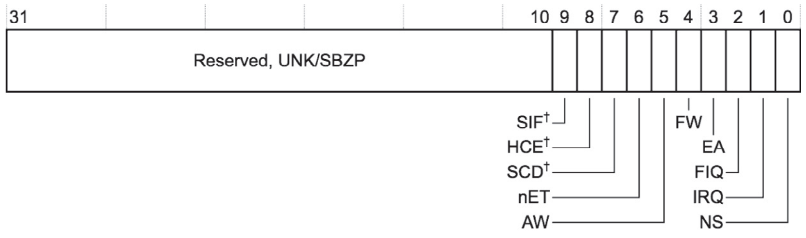
## 3. System design

### 3.1. *Overview*

Mainstream ARM hypervisors such as Xen and KVM/ARM are burdened with a myriad of complicated tasks in order to manage multiple virtual machines (VMs) on the platform. As a result, they have a huge code base and many hypervisor call interfaces, which seriously undermine their security and reliability. OKL4 hypervisor has a small TCB, but the functions are too simple to perform complex control or add extensible modules. In TinyVisor, we build a hypervisor framework with a tiny code size, simple logic, and few interfaces to the kernel, so that it exposes a minimal attack surface to upper layer software. The hypervisor does not support multiple guest domains. Instead, it is devised to act as a trust anchor when the OS



(a) HCR bit assignments [14]



(b) SCR bit assignments [14]

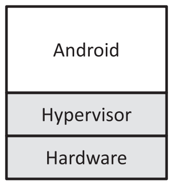
**Fig. 4 – Comparison between HCR and SCR.**

becomes untrustworthy. Fig. 5 depicts the software architectures of our platform and the one used by KVM/ARM.

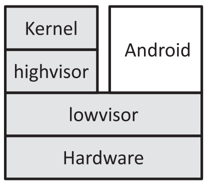
In our implementation, we assume that a hardware based trust chain is built by using loading time integrity check. Namely, the TrustZone verifies the bootloader's integrity while the bootloader verifies the kernel image's integrity. The data objects and resources needed by the hypervisor initialization are prepared by the static code as part of kernel initialization. The hypervisor completes its initialization *before* the kernel launches any process. In the end of hypervisor initialization, it isolates itself from the kernel by configuring the Stage-2 page table so that the kernel has no more control/access over the hypervisor thereafter. We elaborate the details below with Fig. 6 depicting the process.

## 3.2. *Bootstrap and hypervisor loading*

TinyVisor is assembled into the kernel image and is loaded by the bootloader. After leaving the secure state, the bootloader enters the hyp mode in the non-secure state (Step 1 in Fig. 6). The bootloader loads the compressed hypervisor image and kernel image into the main memory, and sets the HYP Vector Based Address Register (HVBAR) to point to an exception vector. It then switches to the svc mode and decompresses the kernel image (Step 2). The decompression allows the bootloader to locate the HYP stub. In Step 3, it then switches back to the hyp mode by issuing a hypervisor call, runs the decompressed HYP stub to configure HYP registers, and sets HVBAR for a new HVC handler for the upcoming hypervisor trap from the kernel.



(a) TinyVisor architecture



(b) KVM/ARM architecture

**Fig. 5 – Comparison between TinyVisor and KVM/ARM architectures. Shadowed boxes are trusted components.**

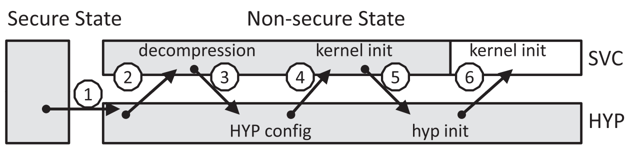


**Fig. 6 – Hypervisor and kernel bootstrap steps. Trusted steps are in the shadowed regions.**

In Step 4, it then returns to the svc mode to start kernel initialization. After the kernel completes its own address space setup, it prepares all resources needed for spinning off the hypervisor. In specific, the kernel allocates a continuous physical memory region as the hypervisor's memory space which, for example, stores the Stage-2 page table, the hypervisor's code and data sections. The kernel copies the uncompressed hypervisor image into the allocated region and configures the Stage-2 page table which defines an identity mapping from the IPAs to the PAs using the long-descriptor translation table format with three levels translation. The Stage-2 page table does not map the memory region priorly allocated by the kernel for the hypervisor because it is considered as the hypervisor space. To avoid the exceptions due to the kernel's benign accesses to this region, e.g., due to memory management, the region is declared as a DMA buffer such that the uncorrupted kernel does not attempt to deallocate or access it. In the end, it issues a hypervisor call to trap to the hyp mode (Step 5). The HVC handler registered in Step 3 then takes control. The handler configures several registers to isolate itself from the kernel and to prepare for runtime event interceptions. To enable the Stage-2 translation, it sets `HCR.VM` to 0x1, and sets Virtualization Translation Table Base Register (`VTTBR`) to the physical address of the root of the Stage-2 page table priorly set by the kernel. It also updates `HVBAR` with the real HVC handler used by the hypervisor for future hypervisor calls. Lastly, the handler returns to the svc mode (Step 6) and resumes the kernel initialization. Note that, from this moment onwards, all kernel and user space code execution use two-stage address translation. Hence, the kernel is excluded from the runtime TCB, which significantly reduces the TCB size. The hypervisor is capable of protecting itself at runtime attacks from the kernel by virtue of the Stage-2 page table.

Bootup security is ensured by the trust chain rooted at the TrustZone. Namely, the TrustZone code verifies the bootloader which in turn verifies the kernel image including the hypervisor image. In addition, the hypervisor is launched before any process starts.

## 3.3. *Runtime interception handling*

After the hypervisor finishes the configuration at boot time, the preparation for hypervisor traps is the upcoming task needed to complete during the runtime. The hardware does not bank all registers during mode switches. Therefore, the hypervisor has to save those unbanked registers by pushing them into the stack before handling the events.

In TinyVisor, we use stack to save the values of different registers. For general registers, the hypervisor uses `PUSH` instruction directly. Meanwhile, for banked registers, the hypervisor uses `MRS` instruction to load their values and then `PUSH` them. Using stack has several advantages as follows:

- Hypervisor does not need to allocate extra space for saving the registers.
- Although hypervisor can read or write the banked registers with the `MRS` and `MSR` instruction respectively, using stack can reduce the number of instructions as well as improve the efficiency.

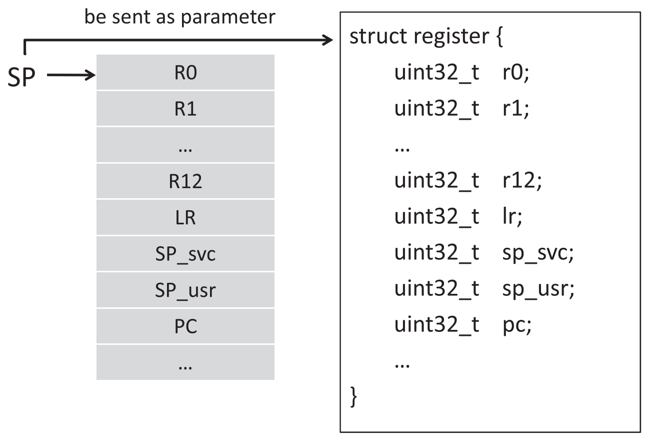


**Fig. 7 – The stack and the struct of saved registers.**

- For every register in the stack, it has an offset to the first register. As a result, the hypervisor can access these registers as a structure conveniently. The relations between the stack and the *struct register* structure are shown in Fig. 7.

Before the hypervisor returns to the upper mode with `ERET` instruction, the hypervisor uses `POP` instruction to restore the general registers and uses `MSR` instruction to restore the banked registers.

After the status is saved, the hypervisor will start to handle the events. TinyVisor mainly handles two kinds of interceptions: hypervisor calls and prefetch/data aborts. As the basis, the hypervisor reads the value of `HSR[31:26]` which is the EC value in order to differentiate the trap causes.

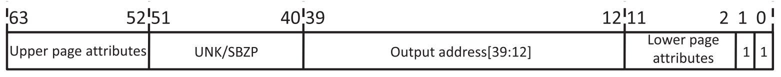
*Hypervisor Calls Handling*. If the EC value equals to 0x12, it means there is a hypervisor call exception. The `R12` is used to save the hypervisor call number and `R0-R4` are for saving the parameters needed. A hypervisor call issued from the svc mode is shown in Fig. 8.

In the HVC handler, the hypervisor jumps to the corresponding functions according to the call number in `R12`. Then it will execute the hypervisor call function using the parameters saved in the stack.

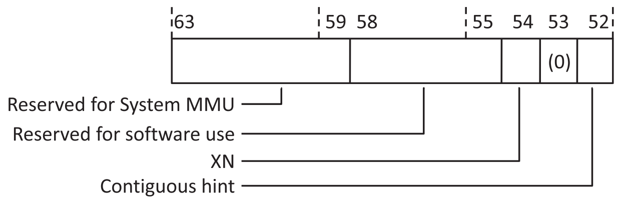
*Prefetch/Data Aborts Handling*. The EC value is 0x20 (0x24), representing the occurrence of a prefetch (data) abort. A prefetch abort exception will be raised when the CPU attempts to fetch an instruction from a page whose Stage-2 PTE is actually set as non-executable. On the other hand, a data abort exception will be raised when there is memory access permission violation. These two exceptions are the main methods for the hypervisor to intercept needed events.

```
push {r0-r2, r12}
mov r12, #40
mov r0, #0
mov r1, #1
mov r2, #2
hvc #0xea1
pop {r0-r2, r12}
```

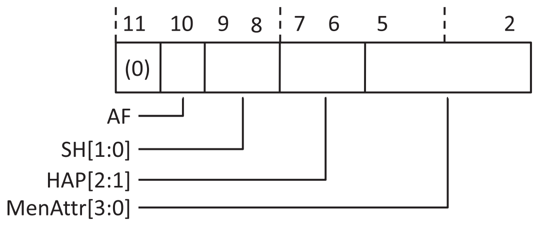
**Fig. 8 – A hypervisor call issued from the svc mode.**



(a) The format of the Stage-2 PTE



(b) The upper page attributes



(c) The lower page attributes

**Fig. 9 – The format of the Stage-2 PTE and its attribute fields.**

In the prefetch and data abort exception handlers, the hypervisor reads the Hyp Instruction Fault Address Register (HIFAR) and the Hyp Data Fault Address Register (HDFAR) to get the address of exception respectively. Then the exceptions will be handled by the hypervisor according to the registers saved in the stack (such as SP_usr or SP_svc[1]).

### 3.4. *Setting of 2-stage table*

TinyVisor uses Stage-2 page table to implement memory virtualization. The guest OS can maintain the Stage-1 page table and translate the VAs to the IPAs, while the hypervisor is responsible for translating the IPAs to the PAs, which is totally transparent to the guest OS. As shown in Fig. 9(a), there are two fields of page attributes in the PTE, including upper page attributes and lower page attributes. The XN bit in Fig. 9(b) is the Execute-never bit which illustrates that this page cannot be executed with the value set to 0x1. The HAP[2:1] bits in Fig. 9(c) are Stage-2 Access Permission bits, whose configuration is shown in Table 2. With the configuration of the PTEs, the hypervsior can decide whether the memory is readable, writable or excutable. Hence, the guest OS can only access the memory which is set accessible by the hypervsior in advance.

## 4. Innovative technology

### 4.1. *Issuance interception of alternative supervisor call*

Leveraging the technique of ARM virtualization extensions, every supervisor call will be routed into the hyp mode if the TGE bit of the HCR is set to 0x1. Using this method, however, the hypervisor traps all the supervisor calls instead of the specified supervisor calls we need and then simulates them. Therefore, the above factors seriously affect the performance of the system. In TinyVisor, a piece of hook code is applied to intercept the appropriate supervisor calls. Hence some unrelated supervisor calls will not be trapped into the hyp mode.

#### 4.1.1. *The original supervisor call interrupt workflow*

The original system uses the SVC instruction to issue a supervisor call from the usr mode to the svc mode. The start address of the interrupt vector table is stored at 0x00000000 or 0xFFFF0000 with the different setting of the V bit of the System Control Register (SCTLR). Fig. 10 displays the content of the vector table.

Regardles of Thumb instructions, a standard ARM instruction takes four bytes. Considering the offset of the SVC instruction is 0x08, a supervisor call interrupt will execute the code in Line 4. Running the code, the PC will be assigned with the content stored at *__vectors_start+0x1000*, which is the supervisor call handler's entry. Then it will go on to run the supervisor call handler.

#### 4.1.2. *Hook the supervisor call interrupt*

TinyVisor inserts the hook code before the control flow gets into the supervisor call handler's entry. If the supervisor call

**Table 2 – Stage-2 control of access permissions.**

| HAP[2:1] | Access permission |
|---|---|
| 00 | No access permitted |
| 01 | Read-only |
| 10 | Write-only |
| 11 | Read/write |

[1] SP_usr means the Stack Pointer used in the usr mode, while SP_svc means the Stack Pointer used in the svc mode.

```
__vectors_start:
  b vector_rst
  b vector_und
  ldr pc, __vectors_start+0x1000
  b vector_pabt
  ...
```

**Fig. 10 – Exception vectors stored in the vector page at either 0x00000000 or 0xFFFF0000 based on SCTLR.V bit.**

is needed, the hook code will issue a hypervisor call to trap into the hyp mode. At the boot time, TinyVisor will allocate a page in the kernel and place the hook code in it. Then TinyVisor will set up the access entry of the Stage-2 page table so as to make the page read-only from the svc mode. The content stored at *__vectors_start+0x1000* will be modified to the start address of the hook. In this way, when the program issues a supervisor call, the PC is assigned with the start address of the hook code. Therefore the program will finally run the hook code. In the hook code, it will first check the Register 7 (R7),which stores the supervisor call number. If the supervisor call is the very thing we need, it will issue an HVC instruction to get into the hyp mode. After the hypervisor call returns, the PC is assigned with the original content stored at *__vectors_start+0x1000* for running the original supervisor call handler. The interception flow is shown in Fig. 11.

#### 4.1.3. *Fasten the hook*

In order to assure that the hook code works correctly and will not be bypassed, TinyVisor uses two methods to fasten the hook code. On one hand, TinyVisor will ensure that the physical address of the vector page cannot be remapped by the kernel. On the other hand, the contents stored in the pages will not be modified by the kernel. TinyVisor therefore uses the methods as follows.

1. The value of the Trap Virtual Memory (TVM) bit of HCR should be set to 0x1 due to the goal of intercepting the access to SCTLR from the kernel as well as to the Translation Table Base Register 1 (TTBR1)[2]. In this way, the hypervisor could impede the change process to SCTLR.V bit and TTBR1.
2. TinyVisor uses TTBR1 to walk the vector page at the virtual address 0x00000000 or 0xFFFF0000 which is decided by SCTLR.V. The hypervisor will make any page used in the walking process read-only in the Stage-2 page table. Therefore, TinyVisor can guarantee the mapping of the vector page from VA to IPA cannot be changed.
3. Similar to Step 2, TinyVisor will make the hook code page walk read-only. Any modification on this mapping will lead to a Stage-2 page fault which will be trapped into the hyp mode.
4. Both the vector page and the hook code are set to be read-only at the Stage-2 page table by TinyVisor in order to ensure the contents of kernels without being altered.

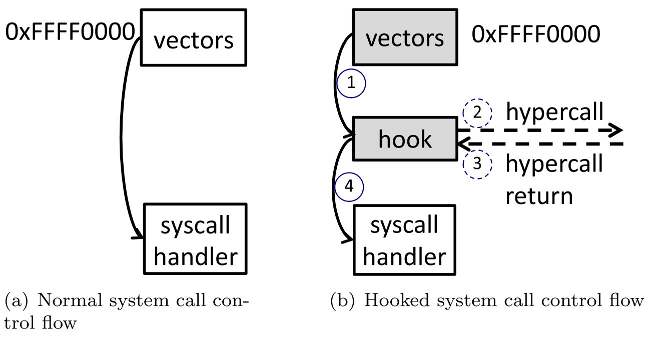


**Fig. 11 – Illustration of hooking the supervisor call control flow where the shadowed boxes refer to pages that are read-only to the kernel and whose addresses cannot be changed. Step 2 and 3 are executed when the intercepted supervisor call needs to be trapped.**

Hence the hardware will position the vector page and hook page to the predefined address once a supervisor call is generated. In addition, the hook code could get better security status and ensure the essential readability because of the read-only characteristic of the vector page and the hook page.

### 4.2. *Thread level interception for supervisor call return*

The supervisor call return is very different from the supervisor call issuance because there will not be an exception when the supervisor call returns. Therefore, it is necessary to inject an exception for the purpose of intercepting the event. The difficulty in realization is the generation of a thread-specific event. The entire threads of the application will be impacted by the process-level interception which is hard to be adopted. For example, if a code page is set non-executable, a page fault will occur for all the threads that try to execute the code, because of the sharing code and data between threads within a process.

This article demonstrates the methodology aimed at controlling the correlative thread's user space stack on account of the theory that stacks of threads are non-shared. The intent is to enable the hypervisor to trap the stack operation when the supervisor call returns.

To begin with, an empty physical memory page will be mapped by an untrusted module, which is called *vault page* in order to engender the necessary exception. Besides, the page is disused for the application and is unregarded in the system.

[2] In ARM architecture, TTBR1 points to the root of the translation table used by kernel and the Translation Table Base Register 0 (TTBR0) points to the root of the translation table used by the current running user process.

For the purpose of blockage of the access and bringing the page fault in the system call return, the Stage-2 page table of the vault page is set *inaccessible*. It is significant to stress that the vault page is compulsive to be mapped in the application's virtual address space. If not, the hypervisor is unavailable to trap the exception caused by the vault page, which will be trapped into the kernel.

Then, the hypervisor uses the aforementioned technique to intercept the supervisor call. The original value of the thread's `SP_usr` will be stored in the hypervisor space, and then the value of `SP_usr` will be updated to the address of the application's vault page. Due to the pass of the parameters and the return address of the supervisor call are all using registers, the kernel's executive condition is not affected by the stack manipulation.

Finally, when the supervisor call returns, the program will read the stack of the thread which `SP_usr` points to. Because the vault page which `SP_usr` points to is inaccessible, there will be a page fault which will be then intercepted by the hypervisor.

## 5. Binder transaction protection module

TinyVisor is an extensible hypervsior framework, which additional function modules could be applied to. Some virtualization techniques like Overshadow (Chen et al., 2010), InkTag (Hofmann et al., 2013), TrustPath (Zhou et al., 2012), AppShield (Cheng et al., 2015) can be grafted into the ARM platform and with the addition in TinyVisor.

In this section, a Binder transaction protection module called H-Binder is put forward, which can protect sensitive Binder transaction data against rootkit on ARM platforms. H-Binder interposes on the Binder transactions ensuring the secrecy and integrity of the transaction data against the rootkit's malicious accesses.

### 5.1. *The binder framework*

The features of Android platform which could offer the different resources management and the extended capabilities for various user applications have been the bright spots in a broad scope. In order to interact and collaborate with system services, the Binder IPC is designed to complete the previous task for user applications. Moreover, the Binder IPC also provides the integrations or cooperations inside themselves in order to perform their given tasks. There is one good example that one user application can get the data of the current location by interacting with Android's LocationManager.

The traditional client-server model is applied in the Binder transactions, mainly including three parts. The role of the server is a thread of a resource manager app. The part of the client refers to a thread of a user application. Meanwhile, the character of the Binder driver is in the kernel. Android's ServiceManager can help the application searching the related service to achieve the Binder transaction, which is likely the function of a registry service. The interactions between client and server threads occur in the Binder transaction with the

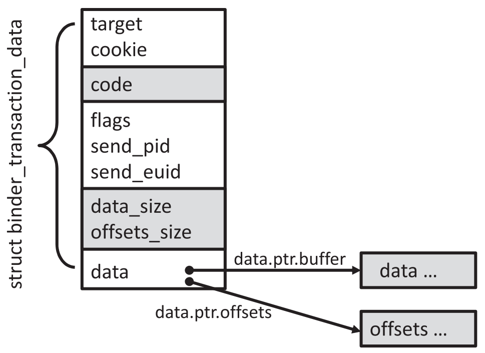


**Fig. 12 – Binder_transaction_data. The shadowed regions refer to the Binder transaction raw data which are the actual payload of a Binder communication.**

tips below. Another thing to emphasize is that several worker threads are at the waiting states for dealing with the requests in sleeping mode.

1) The thread of the client initiates a request to the server of the service. It uses the supervisor call *ioctl* with corresponding parameters and the Binder driver will handle it.
2) After receiving the request, the Binder driver stores the information of the thread of the client and then sends it to the service thread.
3) The service thread is waken up and handles the request. Then it passes the reply using the supervisor call *ioctl* to the Binder driver.
4) The Binder driver loads the information of the client and sends the reply back to the client thread.

The Binder framework owns a critical data structure named the *binder_transaction_data*. Meanwhile, The user-space threads will transmit it as a parameter of *ioctl* to the Binder driver. The shadow areas depicted in Fig. 12 refers to the bytes without changes from the honest kernel. Substantially, the method of execution for receiving app is pointed out by `code`. Moreover, `data.ptr.buffer` points to the buffer, which saves the parameters used by the method and corresponding objects. While the `data_size` is the length of the `data.ptr.buffer`. In order to achieve convenience of expression, the bytes of shadowed boxes represent the *transaction raw data* in the entire article.

It is important to explain the concrete way of finding the service application that will go into operation for a client application, due to the relevance to certification process of Binder transaction. The process of search also belongs to Binder transaction.

The `target` field shown in Fig. 12 is a *local* handler delivered to the Binder driver to determine the target location. The client will set `target` as 0 and add the corresponding text string in Binder request in order to find a service application. The request will then be transponded by the Binder driver to Android's ServiceManager with a handler returned to the client. At last, the client will take the handler as the value of `target` placed in Binder request, which is used for the transaction with the service application.

**Table 3 – The format of the Transaction Table.**

| ClientID | SApp | SThread | ReqID | AckID | State |
|---|---|---|---|---|---|
| 0x96206f40 | 0x960b4280 | 0x76ef2000 | 0x47aa6d75 | 0xb0aacdf4 | 2 |
| … | … | … | … | … | … |

## 5.2. *The approach*

Even though encrypting the Binder raw data of the sending application straightly is a direct and oversimplified way, it is difficult to be correctly decrypted. If the recipient authentication would go through without examining, an inappropriate decryption may reveal the plaintext to an forged target application. Consequently, data security and the authentication of applications are associated. The semantic gap issue may arise potential difficulties in identifying the recipient thread for the hypervisor. At run time, the practical destination of Binder transaction is under the control of Binder driver instead of user threads. The data handled by the Binder driver may be tampered by a rootkit, leading to a bad consequence of wrong data transmission from Binder to an imposter.

Overall speaking, H-Binder does the essential interventions in the four steps of the Binder data transaction, using the techniques introduced in Section 4. When one step is intercepted, the hypervisor will decide to save or restore the data according to the status of sending or receiving the data. However, the hypervisor does not know the relationship between the Binder data and the Binder transaction based on the intercepted data. Therefore, it is necessary for the hypervisor to trace the data flow of Binder transaction, so as to execute the saving or restoring properly, including the Binder transaction of searching a service. To be specific, when the client issues a request, the hypervisor will save the data and replace it with a random number as an ID, which is different from all the current existing IDs. Once the request reaches the server, the first step is to check the received request ID. The hypervisor will carry on restoring corresponding client's request if the ID is correct. Hence, the hypervisor has the ability to know the sending destination when the server's worker thread issues the reply. When the reply arrives at the client, the hypervisor will check whether the current application is the target thread in order to make the decision of recovering the relevant data.

## 5.3. *Details*

We elaborate the details of H-Binder by interpreting a protection cycle of a Binder transaction between a client app and a server app.

### 5.3.1. *Initialization*

When a client app is launched, a kernel module will allocate a vault page and pass the virtual address of this page to the hypervisor. The hypervisor will then configure the Stage-2 page table of the vault page to make it *inaccessible*. Moreover, the hypervisor will record a pair $\langle ttbr, addr\rangle$ representing the value of `TTBR0` and the physical address of the vault page, respectively. This page is used to intercept supervisor call return as described in Section 4.2 and save the Binder data.

The hypervisor also maintains a Service Table whose entry pairs a service description with the `TTBR0` value of the corresponding system service application, e.g., LocationManager. For every user application, the hypervisor also maintains a Handler Table whose entry pairs a handler value with the `TTBR0` value of the corresponding service. The Handler Table of a client app is initialized with an entry $\langle 0, ttbr^*\rangle$ where $ttbr^*$ is the `TTBR0` of the ServiceManager.

The hypervisor establishes the *Transaction Table* shown in Table 3 to save the data related to every Binder transaction such that each intercepted event can be linked to a Binder transaction. In this table, *ClientID* represents the `TTBR0` value of the client app. *SApp* identifies the server app by using its `TTBR0` value while *SThread* means the server's worker thread by using the virtual address of its stack base. *ReqID* and *AckID* save the ID information of the request and reply as their respective identifiers. *State* records the present transaction state.

### 5.3.2. *Runtime*

H-Binder uses the techniques described in previous sections to intercept the entire four steps of the Binder transaction. The workflow of H-Binder proceeds in four phases as depicted in Fig. 13 wherein a client app requests data from a server app through a Binder IPC channel.

We can use a series of set operations to express the entire process. We define the Transaction Table as a set $\mathcal{S}$. At the initialization time, $\mathcal{S} = \varnothing$. For each phase in Fig. 13, we define corresponding operation $\triangleright$ as follows:

**Phase 1**:

$$\begin{aligned} &\mathcal{S} \triangleright_1 \langle ClientID, SApp, ReqID\rangle \\ &= \mathcal{S} \cup \{\langle ClientID, SApp, \varnothing, ReqID, \varnothing, 0\rangle\} \end{aligned}$$

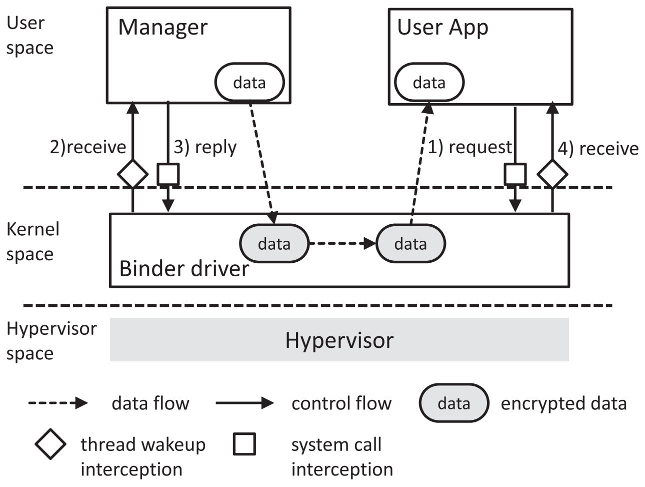


**Fig. 13 – Overview of H-Binder work flow.**

**Phase 2**:

$$\begin{aligned}&\mathcal{S} \rhd_2 \langle ReqID, SApp, SThread\rangle \\ &= (\mathcal{S} \cup \{\langle u, SApp, SThread, ReqID, \varnothing, 1\rangle \mid \\ &\quad \exists w, y(\langle u, SApp, w, ReqID, y, 0\rangle \in \mathcal{S})\}) \\ &\quad \setminus \{\langle u, v, w, x, y, 0\rangle \in \mathcal{S} \mid x = ReqID\}\end{aligned}$$

**Phase 3**:

$$\begin{aligned}&\mathcal{S} \rhd_3 \langle SThread, SApp, AckID\rangle \\ &= (\mathcal{S} \cup \{\langle u, SApp, SThread, x, AckID, 2\rangle \mid \\ &\quad \exists y(\langle u, SApp, SThread, x, y, 1\rangle \in \mathcal{S})\}) \\ &\quad \setminus \{\langle u, v, w, x, y, 1\rangle \in \mathcal{S} \mid v = SApp \wedge w = SThread\}\end{aligned}$$

**Phase 4**:

$$\begin{aligned}&\mathcal{S} \rhd_4 \langle AckID, ClientID, \varnothing\rangle \\ &= (\mathcal{S} \setminus \{\langle u, v, w, x, y, 2\rangle \in \mathcal{S} \mid y = AckID\}\end{aligned}$$

Next, we will explain these operations in details combining with the workflow.

**Phase 1: User app sending request.**

$$\begin{aligned}&\mathcal{S} \rhd_1 \langle ClientID, SApp, ReqID\rangle \\ &= \mathcal{S} \cup \{\langle ClientID, SApp, \varnothing, ReqID, \varnothing, 0\rangle\}\end{aligned}$$

Leveraging the techniques introduced in Section 4.1, the hypervisor will intercept the client app's *ioctl* call right after the program is trapped 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 vault page of the client app 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`. As a result, $\mathcal{T} = \langle ClientID, SApp, \varnothing, ReqID, \varnothing, 0\rangle$.

**Phase 2: Manager receiving request.**

$$\begin{aligned}&\mathcal{S} \rhd_2 \langle ReqID, SApp, SThread\rangle \\ &= (\mathcal{S} \cup \{\langle u, SApp, SThread, ReqID, \varnothing, 1\rangle \mid \\ &\quad \exists w, y(\langle u, SApp, w, ReqID, y, 0\rangle \in \mathcal{S})\}) \\ &\quad \setminus \{\langle u, v, w, x, y, 0\rangle \in \mathcal{S} \mid x = ReqID\}\end{aligned}$$

When the request is passed to the server app by the Binder driver, the worker thread of the server app is wakened up to handle it. Using techniques described in Section 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. The set $\mathcal{S}_2$ of the target $\mathcal{T}$ is $[\langle u, v, w, x, y, z\rangle \in \mathcal{S} \mid x = ReqID]$.
2) If $\mathcal{S}_2 = \varnothing$, it means that no matching record is found so that the hypervisor will return an error to the guest OS. Because the ReqID is unique, there will be only one $\mathcal{T} \in \mathcal{S}_2$ if $\mathcal{S}_2 \neq \varnothing$. As assigned in **Phase 1**, $\mathcal{T}$ should be $\langle ClientID, SApp, \varnothing, ReqID, \varnothing, 0\rangle$. In this phase, the hypervior knows the current `TTBR0` as *SApp*. Therefore, if the current `TTBR0` does not equal to $\mathcal{T}$.SApp 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) If $\mathcal{T}$ satisfies with Step 2, the hypervisor loads the data from the client's vault page to recover its original Binder request, saves $\mathcal{T}$.SThread with `SP_usr`&0xFFFFE000 to record the worker thread's identity, and lastly sets $\mathcal{T}$.State to 1 to indicate that the request is received by the server. As a result, $\mathcal{T} = \langle ClientID, SApp, SThread, ReqID, \varnothing, 1\rangle$.

**Phase 3: Manager sending reply.**

$$\begin{aligned}&\mathcal{S} \rhd_3 \langle SThread, SApp, AckID\rangle \\ &= (\mathcal{S} \cup \{\langle u, SApp, SThread, x, AckID, 2\rangle \mid \\ &\quad \exists y(\langle u, SApp, SThread, x, y, 1\rangle \in \mathcal{S})\}) \\ &\quad \setminus \{\langle u, v, w, x, y, 1\rangle \in \mathcal{S} \mid v = SApp \wedge w = SThread\}\end{aligned}$$

After handling the request of the client app, the worker thread of the server app returns a reply to the client app. Using the hook in Section 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 and $\mathcal{T}$.SApp equals to the present `TTBR0`. The set $\mathcal{S}_3$ of the target $\mathcal{T}$ is $\{\langle u, v, w, x, y, z\rangle \in \mathcal{S} \mid v = SApp \wedge w = SThread\}$.
2) If $\mathcal{S}_3 = \varnothing$, the hypervisor will return an error because no matching data is found. Using SApp and SThread, an exclusive working thread can be determined. Hence $\#(\mathcal{S}_3) = 1$ if $\mathcal{S}_3 \neq \varnothing$. The only $\mathcal{T} \in \mathcal{S}_3$ should be $\langle ClientID, SApp, SThread, ReqID, \varnothing, 1\rangle$. The hypervisor 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 into 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. As a result, $\mathcal{T} = \langle ClientID, SApp, SThread, ReqID, AckID, 2\rangle$.

**Phase 4: User app receiving reply.**

$$\mathcal{S} \triangleright_4 \langle AckID, ClientID, \varnothing \rangle$$
$$= (\mathcal{S} \setminus \{\langle u, v, w, x, y, 2 \rangle \in \mathcal{S} | y = AckID\}$$

When the Binder driver passes the server's reply to the client app, it wakes up the user's blocked thread described in Phase 1. Using the techniques in Section 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. The set $\mathcal{S}_4$ of the target $\mathcal{T}$ is $\{\langle u, v, w, x, y, z \rangle \in \mathcal{S} | y = AckID\}$.
2) If $\mathcal{S}_4 = \varnothing$, the hypervisor discards the reply as no matching record is found. As the AckID is unique, there is only one $\mathcal{T} \in \mathcal{S}_4$ if $\mathcal{S}_4 \neq \varnothing$. $\mathcal{T}$ should be $\langle ClientID, SApp, SThread, ReqID, AckID, 2 \rangle$. 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 the 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.

## 5.4. *Analysis of security*

We provide a series of analysis to explain the theory of data protection during the Binder transaction. The analysis contains three aspects: confidentiality, integrity and availability.

### 5.4.1. *Confidentiality*

The confidentiality of the Binder data is ensured 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 supervisor call issuance. As shown in Phase 2 and 4, restoring is only performed after a successful authentication of the recipient app. Therefore, only the intended applications can access those data.

During the process of data transaction, the data is saved in the fault page which is inaccessible to the guest OS because of the Stage-2 page table. Any interception of the attackers can only get the ID of the transaction which is not sensitive.

### 5.4.2. *Integrity*

In Binder transaction, the integrity mainly contains two aspects. On one hand, the transaction data must be sent to the intended receiver. On the other hand, the data cannot be modified or reused by the attackers. Next, we will focus the analysis on these two aspects mentioned above.

*Recipient Authenticity.* Recipient authenticity is about whether a Binder request/reply is passed to the expected destination.

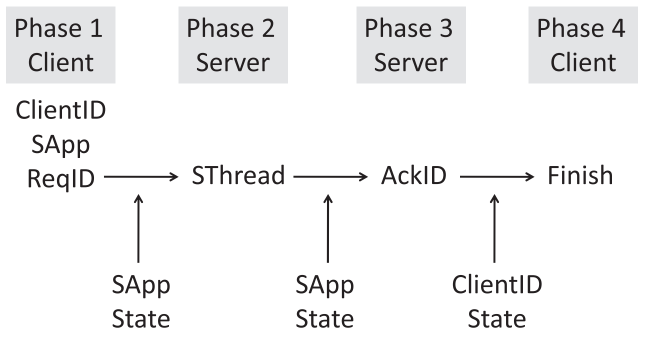


**Fig. 14 – The workflow of the operation ▷.**

For the flow from the client to the server, the hypervisor extracts the expected 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.

In the operation ▷, the parameter is a tuple $\langle p_1, p_2, p_3 \rangle$. For all the four phases, $p_1$ and $p_2$ can be read from the guest OS by the hypervisor and $p_3$ is generated by the hypervisor. To the hypervisor, the tuple can be shown as $\langle input_1, input_2, output \rangle$. In Phase 1, $input_1$ and $input_2$ are all used to add into the table. In the other phases, $input_1$ is used to search in the table, and $input_2$ is used to check the correctness. Therefore, the hypervisor uses $input_2$ as a key. The parameter can be $\langle input, key, output \rangle$.

Fig. 14 shows the workflow. In Phase 1, the two keys *ClientID* and *SApp* are used as the input and the output is *ReqID*. In the following phases, the input is the output of the previous phase. The key has different values according to the receiver, *SApp* for the server and *ClientID* for the Client. Using operation ▷, the hypervisor develops a trust chain $ReqID \rightarrow SThread \rightarrow AckID \rightarrow Finish$. If the target is wrong, the `TTBR0` will not match the corresponding key, which will lead to an error returned. In this way, the hypervisor has sufficient knowledge to decide the intended recipient for a Binder reply from the server app.

*Application Data Integrity.* Binder data integrity is ensured by $\mathcal{T}$.ReqID and $\mathcal{T}$.AckID as shown in Fig. 14. If the attackers intercept the transaction data and modify it to destory the integrity, such as modifying the value of `code` in *binder_transaction_data*, the corresponding ReqID or AckID will be changed which will lead to a mismatch. Therefore, a fraudulent Binder request can be detected in Phase 2 and 4 before the recipient app processes it.

The attackers can also reuse the intercepted transaction Binder request to make the server run some functions again. In H-Binder, the transaction's state stored in $\mathcal{T}$.State is used to detect replay attacks. After H-Binder intercepts a request, the value of $\mathcal{T}$.State adds 1. Therefore, the value of $\mathcal{T}$.State is determined in different phases. If the replay attack happens in the middle phases, the value of $\mathcal{T}$.State will not match which will lead to an error. If the replay attack happens in the last phase, there will not be any corresponding entry because H-Binder will delete the entry when the transaction completes.

#### 5.4.3. *Availability*

The availability of H-Binder can be protected by the hardware. The hyp mode is transparent to the guest OS so that the rootkits do not know the existence of the hypervisor. Therefore, the attackers cannot break nor stop H-Binder. H-Binder can run correctly as long as there is no vulnerability in itself. Furthermore, the small TCB can reduce the probability of vulnerability.

## 6. Implementation and performance evaluation

We have implemented a prototype of TinyVisor with H-Binder running in the hyp mode. The runtime TCB of TinyVisor only consists of 1813 SLOC (1144 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 @3.6GHz processor and 16 GB main memory. In this platform, we run ARM FastModels (ARM, 2011) with FVP which emulates a tablet with a Cortex-A15x1 processor. TinyVisor runs in the emulated tablet as a bare-metal hypervisor. On top of the hypervisor, it runs Android 4.1[3] 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 TinyVisor performance.

### 6.1. *Evaluation of TinyVisor*

Testing the overhead of running the guest system is the most significant evaluation. Several kinds of Java benchmarks which will be introduced in detail in the following contents are used because the running guest system is Android. If the overhead in Android is insignificant, the result will prove the acceptable overhead incurred by TinyVisor.

In our implementation, each system call triggers the hook module to check `R7` and `TTBR0`. Hence, we expect TinyVisor to make a negative performance impact on the Android system as a whole. To obtain its actual performance drop, we use SciMark 2.0 (Pozo and Miller, 2004) and CaffeineMark 3.0 (Pendragon Software Corporation, 1997) as the Java Microbenchmark to test the running speed of Java programs in the system.

Table 4 and Fig. 15 shows the two benchmark results respectively. In Table 4, the numbers in the table are the scores of different tests in different environments. The higher score means the better performance. The composite score of Android is 1.0913 while the score of TinyVisor is 1.0687. Meanwhile in Fig. 15, the taller bar stands for the better performance. The "Overall" bars show the comprehensive benchmark scores. The average score of Android is 104 while the score of TinyVisor is 102. Combining the two results, the overhead of TinyVisor is only about 2%.

[3] Because of the limitation of the environment, we use an old version instead of Android 7.0. But the evaluation results won't be affected too much because the interceptions mainly aim at the kernel.

**Table 4 – SciMark 2.0 benchmark.**

| Scimark2 | Android | TinyVisor |
|---|---|---|
| Fast Fourier Transform | 0.1427 | 0.0804 |
| Jacobi Successive over-relaxation | 2.6077 | 2.5633 |
| Monte Carlo Integration | 0.2492 | 0.2456 |
| Sparse Matrix Multiply | 0.9773 | 0.9754 |
| Dence Lu Matrix factorization | 1.4795 | 1.4789 |
| Composite | 1.0913 | 1.0687 |

CAVEAT. The performance overhead is to intercept all supervisor calls. Nevertheless, the issuance interception of alternative supervisor call described in Section 4.1 allows to intercept needed supervisor calls in order to improve the overhead ulteriorly. It can be further extended to select the critical applications and services to protect.

### 6.2. *Evaluation of H-binder*

Because TinyVisor framework only intercepts the supervisor calls without actually handling them, the overhead is very tiny. With H-Binder, TinyVisor can act the real protection. In this part, we then evaluate the overhead of H-Binder.

#### 6.2.1. *Component cost*

The overall time overhead caused 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 measure the former cost, we evaluate the turnaround time of an empty hypervisor call which causes the CPU to enter 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.

We also measure the CPU time spent in each of the four phases described in Section 5. The average CPU cycles spent in each of the phases are listed in Table 5 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.

As shown in Section 5, a Binder IPC upon H-Binder involves 4 traps into the hypervisor. Therefore, the overall H-Binder

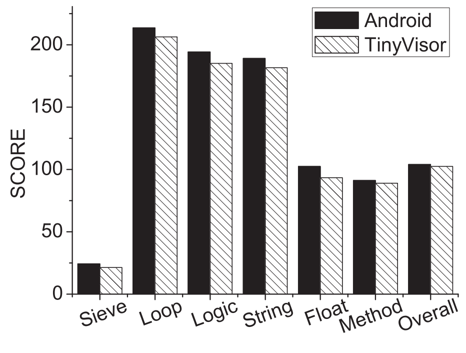


**Fig. 15 – CaffeineMark 3.0 benchmark.**

**Table 5 – 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 |

**Table 6 – Turnaround time (in CPU cycles) needed to obtain the location in different settings.**

| | Android | TinyVisor |
|---|---|---|
| Read Location | 68,577 | 77,344 |
| Overhead | - | 8767 |

cost for protecting a Binder transaction is the sum of mode switch costs and the hypervisor's processing time, which amounts to 3353 CPU cycles. For a mobile phone with 1GHz CPU frequency, the time latency for one Binder transaction is about 3.4$\mu$s, which is very tiny.

### 6.2.2. *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*[4] as the client requesting for the tablet'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 two different environments: the native Android and the Android running on TinyVisor with H-Binder. Note that both the environments are hosted by ARM FastModels emulation. The results are presented in Table 6 below.

It shows that H-Binder incurs about 9000 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 1GHz CPU frequency, the time latency incurred by H-Binder is less than 9$\mu$s. Note that the geographical 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 15m/s, the shortest time interval of location update is 67 ms. The latency of 9$\mu$s 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 100 ms (Shneiderman, 1998).

### 6.2.3. *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 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

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

**Table 7 – A whole binder transaction time in CPU cycles with different sizes of transferred data.**

| Bytes | Android | TinyVisor | Overhead |
|---|---|---|---|
| 4 | 94,848 | 95,170 | 0.3% |
| 8 | 95,070 | 95,781 | 0.7% |
| 12 | 95,670 | 96,812 | 1.2% |
| 20 | 96,070 | 96,960 | 0.9% |
| 40 | 97,196 | 102,871 | 5.8% |
| 80 | 100,349 | 107,118 | 6.7% |

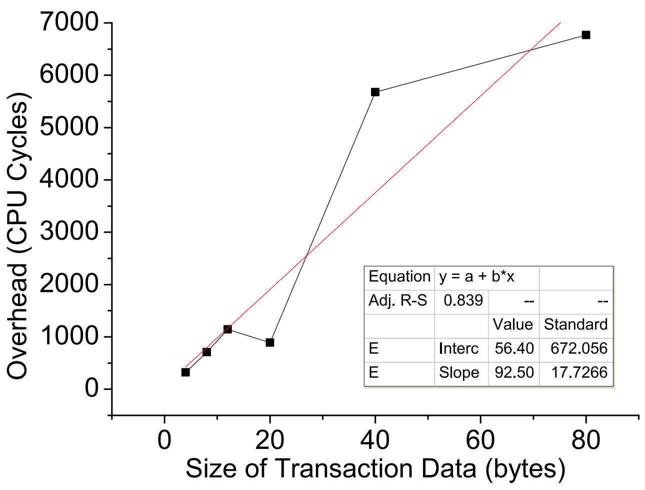
the data, including the time spent for the Binder channel setup. Table 7 and Fig. 16 report the experiment results in two different platforms.

Fig. 16 turns out that the time cost rises linearly as the size of data increases. Using linear fitting, the equation between the overhead ($Y$) and the size of the transaction data ($X$) is $Y = 92.5X + 56.4$. The 92.5 CPU cycles overhead per byte contains the overhead for output as well. 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 7%. If the situation of large data size occurs, the Ashmem will be used, which will be discussed in Section 7.1.

## 7. Discussion

### 7.1. *Binder transactions with Ashmem*

For bulky data communication between applications, Binder with Anonymous Shared Memory (Ashmem) is used instead of wrapping the data within the *binder_transaction_data* structure. In a nutshell, the typical workflow of Binder with Ashmem is as follows. The sender app first maps a buffer in its virtual address space as a shared memory in the form of a device file. It then issues a Binder command to the Binder driver with a file descriptor referring to the shared area. At the receiver's end, the receiver app also maps the shared area into its own file descriptor assigned by the Binder driver. After both applications



| Equation | y = a + b*x | | |
|---|---|---|---|
| Adj. R-S | 0.839 | -- | -- |
| | | Value | Standard |
| E | Interc | 56.40 | 672.056 |
| E | Slope | 92.50 | 17.7266 |

**Fig. 16 – The overhead with different data sizes and linear fitting.**

obtain their respective file descriptors, they exchange data by accessing the shared memory using file operations.

The H-Binder module in Section 5 cannot be directly applied to Binder transactions with Ashmem, since the application data is not transported by the kernel driver using the Binder data structure. We propose to make a few changes on H-Binder in order to protect shared memory based Binder IPC. The hypervisor still intercepts *ioctl* as in the previous scheme. To determine whether a Binder call uses Ashmem or not, the hypervisor locates the *flat_binder_object* structure via the parameter and checks whether the structure's `type` field is set as BINDER_TYPE_FD. If Ashmem is in use, the hypervisor records the file descriptors used by both the client and the server. It then intercepts all file read and write operations of the client and the server. For write operation, it encrypts the data to write before trapping to the kernel while for the read operation, it decrypts the ciphertext in the shared memory. Since our hook has the ability to filter system calls of target processes, the proposed approach does not intercept all file operations in the platform. The implementation for protecting Ashmem is left for our future work.

### 7.2. *Binder commands*

Android's Binder framework defines five *ioctl* commands related to the Binder. The most important command among them is `BINDER_WRITE_READ`, which carries a series of Binder commands and data. The other four commands are used to instruct the kernel to manage the threads and do not entail any data exchanges between processes. For instance, `BINDER_SET_IDLE_TIMEOUT` sets a timer for the waiting thread. Hence, H-Binder only escorts *ioctl* calls with `BINDER_WRITE_READ`.

Most types of Binder commands issued via `BINDER_WRITE_READ` are for book-keeping purposes. For instance, `bcINCREFS` instructs the Binder driver to increase the object reference. These commands do not carry application-level data. Although attacks on those operations do harm the success or performance of a Binder transaction (which is out of our scope of protection), they do not compromise secrecy or integrity of exchanged application data. In the Binder framework, `BC_TRANSACTION`, `BC_REPLY`, `BR_TRANSACTION` and `BR_REPLY` are the only four Binder commands used for IPC and are duly monitored and safeguarded by H-Binder as shown in Section 3.

### 7.3. *Future work*

TinyVisor provides an extensible hypervisor framework, which can add different modules to achieve different protection. In our future work, we will enrich its functions with different modules.

1) Private Information Protection. As people become increasingly dependent on mobile devices, a large amount of private information is saved in the mobile devices. To protect this information, the system must protect all the paths including input, transaction, storing and output. H-Binder provides the protection of data transaction, but the other paths are also needed to be protected, which is the part of our future work.
2) Trusted User Interface. Current TinyVisor can only achieve aptotic functions. Any turn on/off of some function needs to be recompiled. In our future work, we will design a trusted user interface, which is used for users to control the hypervisor.
3) The Combination with TrustZone. In TinyVisor, we use TrustZone to check the integrity of the hypervisor image. We have compared the two techniques in Section 2.3. In our future work, we will combine TinyVisor with TrustZone to build a more secure and efficient system protection framework. TrustZone is used to save the keys and the encryption programs, while the hypervisor is responsible for managing the guest OS and intercepting the exceptions.

## 8. Related work

Early research on virtualization technology is mainly focused on full-virtualization and para-virtualization technologies due to the lack of the support on hardware virtualization. QEMU (Bartholomew, 2006; Bellard, 2005) is a well-known study using full-virtualization technology (Suzuki and Oikawa, 2011). QEMU can run an unmodified OS using instruction simulation. It supports x86, ARM, MIPS and many other platforms. Meanwhile, it can achieve the simulation across the platforms, such as running Windows on ARM platform. Oh et al. proposed ViMo (Oh, 2010), which is a hypervisor running in the mobile system based on full-virtualization technology. Compared to the hypervisors using full-virtualization technology, TinyVisor has a distinct advantage on the overhead because TinyVisor does not need to simulate the instructions.

Because of using binary translation (Sites, 1993) and instruction simulation technologies, the overhead of full-virtualization technology is significant. On the contrary, the performance of para-virtualization and hardware-assisted virtualization technologies improves apparently. Before the appearance of the hardware virtualization extensions, researchers focused on para-virtualization technology and proposed many technical frameworks, such as Xen on ARM (Hwang et al., 2008), KVM for ARM (Dall and Nieh), OKL4 microvisor (Heiser and Leslie, 2010) and so on. Many studies on para-virtualization (Ding et al., 2012; Lee et al., 2008; Li, 2011; Park et al.; Rossier; Zhao, 2011; Zhong, 2013) are based on these frameworks. Compared to the studies using para-virtualization, TinyVisor has a similar running overhead to them. However, the guest system running in TinyVisor does not need to be modified, while using para-virtualization needs to alter the guest systems.

After ARM virtualization extensions were proposed, many groups proposed the virtualization frameworks based on ARM virtualization extensions. Xen (XenProject, 2013) added the support to ARM virtualization extensions in Xen4.3.0. Lengyel et al. proposed a multi-tiered security architecture (Lengyel et al., 2014) for ARM via the virtualization and security extensions. KVM/ARM (Dall and Nieh, 2014) leverages the functions in Linux kernel to manage the guest OSes. Paolino et al. proposed T-KVM (Paolino et al., 2015) based on KVM/ARM, which includes four independent components: ARM virtualization extensions, TrustZone, TEE interface and SELinux module. Compared to

TinyVisor, each of Xen and KVM has a host VM which leads to a huge TCB. Varanasi exported OKL4 into the hyp mode to achieve a hypervisor (Varanasi, 2010). However, the hypervisor is too simple to be extended, while TinyVisor leaves the interfaces to be extensible.

There are also many researchers using hardware-assisted virtualization technology to solve security problems. Yang et al. proposed DroidVisor which was a lightweight monitor for Android kernel protection. DroidVisor could check the module loading and protect the integrity of the key objects. It could also check the hidden modules and processes. Horsch et al. used hardware-assisted virtualization technology to build a hypervisor (Horsch and Wessel, 2015) for tracing the control flows of programs and used the hypervisor to protect the control flows based on page-table hashing. Rordholz et al. proposed XNPro (Nordholz et al., 2015), which protected the OS from the code injection attack using the strict limit on the execution of the kernel codes. Compared to TinyVisor, all the schemes above were individual but not modularized. As a result, these schemes couldn't work together very well.

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 (Rosa, 2011). More recent attacks (Artenstein and Revivo, 2014) presented in the Black Hat conference further demonstrated the cruciality of Binder security. It was shown in Artenstein and Revivo (2014) that a malware which controlled the Binder framework by attacking the *ioctl* system call could access and manipulate a variety of sensitive data, including keystrokes, in-app data and SMS messages. ComDroid (Chin et al., 2011) proposed a tool to detect the vulnerabilities in Binder transaction, but it could not provide a runtime protection. AppFence (Hornyack et al., 2011) was built based on TaintDroid (Enck et al., 2010), 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.

## 9. Conclusion

We have proposed TinyVisor which leverages the recent ARM hardware virtualization techniques to protect the guest OS. TinyVisor can intercept the supervisor calls and leave different interfaces for extension. We also propose a module called H-Binder for TinyVisor to secure the Binder transaction in Android. H-Binder ensures secrecy and integrity of the sensitive data transported between two application threads interacting via Binder IPC. We have implemented a prototype of TinyVisor with H-Binder on ARM FastModels. Our experiments show that the overhead incurred by TinyVisor is not significant. Our future work is to enrich the framework to achieve more protections.

## Acknowledgment

This research work is supported in part by the National High Technology Research and Development Program of China (No.2015AA016004), the National Natural Science Foundation of China (Grand Nos. U1636211, 61672081, 61602237, 61370126), and the National Research Foundation Singapore under the NCR Award Number NRF2014NCR-NCR001-012.

## REFERENCES

Architecture Reference Manual (ARMv7-A and ARMv7-R edition), ARM DDI C, 2008.
Aroca RV, Gonçalves LMG. Towards green data centers: a comparison of x86 and ARM architectures power efficiency. J Parall Distrib Comput 2012;72(12):1770–80.
Artenstein N, Revivo I. Man in the binder: He who controls IPC, controls the droid, Black Hat; 2014.
Arzt S, Huber S, Rasthofer S, Bodden E. Denial-of-app attack: Inhibiting the installation of android apps on stock phones, in: Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, pp. 21–26; 2014.
ARM, Fast Models – ARM; 2011. Available from: http://www.arm.com/products/tools/models/fast-models/. [Accessed 30 August 2015].
Azab AM, Ning P, Shah J, Chen Q, Bhutkar R, Ganesh G, et al., Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world, in: Proceedings of the 21st ACM Conference on Computer and Communications Security, pp. 1028–1031; 2014.
Bartholomew D. QEMU: a multihost multitarget emulator. Linux J 2006;2006(145):68–71.
Bellard F. QEMU, a fast and portable dynamic translator, in: Conference on Usenix Technical Conference, pp. 41–46; 2005.
Chen X, Garfinkel T, Lewis EC, Subrahmanyam P, Waldspurger CA, Boneh D, et al. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. ACM SIGOPS Operat Syst Rev 2010;42(2):2–13.
Cheng Y, Ding X, Deng RH. Efficient virtualization-based application protection against untrusted operating system, in: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, pp. 345–356, 2015.
Chin E, Felt AP, Greenwood K, Wagner D. Analyzing inter-application communication in android, in: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, pp. 239–252; 2011.
Dall C, Nieh J. Kvm for arm, Proceedings of the Ottawa Linux Symposium. 2010.
Dall C, Nieh J. KVM/ARM: the design and implementation of the linux ARM hypervisor. ACM SIGARCH Comput Architect News 2014;42(1):333–48. doi:10.1145/2541940.2541946. Available from: http://doi.acm.org/10.1145/2541940.2541946.
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.
Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, et al. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. Commun ACM 2010;57(3):99–106.
Heiser G, Leslie B. The okl4 microvisor: convergence point of microkernels and hypervisors, in: ACM SIGCOMM Asia-Pacific Workshop on Systems, Apsys 2010, New Delhi, India, pp. 19–24, August, 2010.
Hofmann OS, Kim S, Dunn AM, Lee MZ, Witchel E. Inktag: Secure applications on an untrusted operating system, in: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 253–264. doi:10.1145/2451116.2451146; 2013. Available from: http://doi.acm.org/10.1145/2451116.2451146.

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: ACM Conference on Computer and Communications Security, pp. 639–652; 2011.
Horsch J, Wessel S. Transparent page-based kernel and user space execution tracing from a custom minimal arm hypervsior, in: The IEEE International Conference on Trust, Security and Privacy in Computing and Communications, pp. 408–417, 2015.
Hwang J-Y, bum Suh S, Heo S-K, Park C-J, Ryu J-M, Park S-Y, et al., Xen on ARM: System virtualization using xen hypervisor for ARM-Based secure mobile phones, in: Proceedings of the 5th IEEE Consumer Communications and Networking Conference, pp. 257–261. doi:10.1109/ccnc08.2007.64; 2008.
Lee SM, Suh SB, Jeong B, Mo S. A multi-layer mandatory access control mechanism for mobile devices based on virtualization, in: Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE, pp. 251–256, 2008.
Lengyel TK, Kittel T, Pfoh J, Eckert C. Multi-tiered security architecture for arm via the virtualization and security extensions, in: International Workshop on Database and Expert Systems Applications, pp. 308–312; 2014.
Li D. Multi-platform extension of lightweight virtual machines [Master’s thesis]. Huazhong University of Science and Technology; 2011.
Mulliner C, Robertson W, Kirda E. Virtualswindle: an automated attack against in-app billing on android, pp. 459–470; 2014.
Nordholz J, Vetter J, Peter M, Junkerpetschick M, Danisevskis J. XNPro: low-impact hypervisor-based execution prevention on ARM, in: International Workshop on Trustworthy Embedded Devices, pp. 55–64, 2015.
Oh SC. Vimo (virtualization for mobile) : A virtual machine monitor supporting full virtualization for arm mobile systems, 48–53; 2010.
Ou Z, Pang B, Deng Y, Nurminen JK, Hui P. Energy- and cost-efficiency analysis of ARM-based clusters, in: The IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 115–123; 2012.
Paolino M, Rigo A, Spyridakis A, Fanguède J, Lalov P, Raho D. T-KVM: A Trusted architecture for KVM ARM v7 and v8 Virtual Machines Securing Virtual Machines by means of KVM, TrustZone, TEE and SELinux, in: CLOUD COMPUTING 2015: The Sixth International Conference on Cloud Computing, GRIDs and Virtualization, 2015.
Park M, Yoo SH, Yoo C. Real-time operating system virtualization for xen-arm. 2008.
Pendragon Software Corporation, CaffeineMark 3.0; 1997. Available from: http://www.benchmarkhq.ru/cm30/. [Accessed 20 October 2015].
Pozo R, Miller B. Java SciMark 2.0; 2004. Available from: http://math.nist.gov/scimark2. [Accessed 18 March 2017].
Rosa T. Android binder security note: On passing binder through another binder; 2011.
Rossier D. EmbeddedXEN: A revisited architecture of the xen hypervisor to support ARM-based embedded virtualization, White paper, Switzerland. 2012.
Shen D, Zhang Z, Li Z, Ding X, Deng R. H-Binder: a hardened binder framework on android systems, in: 12th EAI International Conference on Security and Privacy in Communication Networks, pp. 24–43; 2016.
Shneiderman B. Designing the user interface: strategies for effective human-computer interaction. 3rd ed. Addison-Wesley; 1998.
Sites RL. Binary translation. Commun ACM 1993;36(2):69–81.
Smith B. ARM and Intel battle over the mobile chip’s future. Computer 2008;41(5):15–18.
Spaulding J, Krauss A, Srinivasan A. Exploring an open wifi detection vulnerability as a malware attack vector on ios devices, in: International Conference on Malicious and Unwanted Software, pp. 87–93; 2012.
Suzuki A, Oikawa S. Implementing a simple trap and emulate vmm for the arm architecture, in: IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 371–379; 2011.
Uhlig R, Neiger G, Rodgers D, Santoni AL, Martins FCM, Anderson AV, et al. Intel virtualization technology. Computer 2005;38(5):48–56.
Varanasi P. Implementing hardware-supported virtualization in okl4 on arm [Ph.D. thesis]. The University of New South Wales; 2010.
Wen-Xin LI, Wang JB, De-Jun MU, Yuan Y. Survey on android rootkit, Microprocessors. 2011.
XenProject, Supported Xen Project 4.4 series; 2013. Available from: http://www.xenproject.org/downloads/xen-archives/xen-44-series.html. [Accessed 7 May 2014].
Yang Y, Qian Z, Huang H. A lightweight monitor for android kernel protection, Computer Engineering. 2014.
You DH, Noh BN. Android platform based linux kernel rootkit, in: International Conference on Malicious & Unwanted Software, pp. 79–87; 2011.
Zhao Y. Study of linux kernel virtual machine technology implemented on arm platform [Master’s thesis]. Huazhong University of Science and Technology; 2011.
Zhong M. Study of embedded system security assurance based on virtualization technology [Master’s thesis]. Nankai University; 2013.
Zhou Z, Gligor VD, Newsome J, McCune JM. Building verifiable trusted path on commodity x86 computers, in: Proceedings of the 33rd IEEE Symposium on Security and Privacy, pp. 616–630, 2012.

**Dong Shen** received his bachelor’s degree in 2012 from Beihang University, Beijing, China. He is currently working toward the PhD degree in the School of Computer Science and Engineering at Beihang University, Beijing, China. His research interests include mobile security and virtualization security. He has published papers in securecomm2016 and Journal of Software.

**Zhoujun Li** received his PhD degree in 1999 from National University of Defense Technology, Hunan, China. He is a professor of the School of Computer Science and Engineering at Beihang University, Beijing, China. His main research interests include concurrency theory and process algebra, formal analysis and verification of security protocols, information security, data mining.

**Xiaojing Su** is a computational lithography engineer of Integrated Circuit Advanced Process Center of IMECAS (Institute of Microelectronics of Chinese Academy of Science). She received her bachelor’s degree in 2012 from University of Electronic Science and Technology of China (UESTC) and master degree from University College London in 2013 respectively. Since December 2013, she was working in the IMECAS. Her research experience and interest covers optical proximity correction (OPC), computational lithography technology, IC design and IC high reliability.

**Jinxin Ma** received his PhD in 2014 from Beihang University. He is currently working in China Information Technology Security Evaluation Center, Beijing, China. He has been the project manager of NSFC(61502536). He has applied for 2 patents and published more than 15 EI indexed papers. His research interests include software security and vulnerability analysis.

**Robert H. Deng** has been a Professor at the School of Information Systems, Singapore Management University since 2004. His research interests include data security and privacy, multimedia security, network and system security. He is currently Associate Editor of IEEE Transactions on Dependable and Secure Computing, and member of Editorial Board of the Journal of Computer Science and Technology (the Chinese Academy of Sciences) and the International Journal of Information Security (Springer), respectively. He is the chair of the Steering Committee of the ACM Symposium on Information, Computer and Communications Security (ASIACCS).