TZ-SSAP: Security-Sensitive Application Protection on Hardware-Assisted Isolated Environment

Yanhong He[1,2(✉)], Xianyi Zheng[1,2], Ziyuan Zhu[1,2], and Gang Shi[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{heyanhong, zhengxianyi, zhuziyuan, shigang}@iie.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China

Abstract. In the current operating systems (OS), the kernel has complete access to and control over all system sources. However, there are many secure vulnerabilities in kernel because it has the large code base and attack surfaces. Thus, an attacker can attack sensitive applications running on OS by exploiting kernel vulnerabilities. Unfortunately, there are various shortcomings for the existing applications protection mechanisms, such as ignoring the integrity of kernel code, relying on special compiler and et al. In this paper, we have proposed a security-sensitive application (*SSApp*) protection mechanism called *TZ-SSAP* on TrustZone enabled platforms. *TZ-SSAP* introduces four protection modules altogether to provide a safe executable environment for *SSApp* during the system is running. The first one is the *SSApp* protection module which takes advantage of the existing page table mechanism to protect the integrity of code executed by *SSApp* as well as the confidentiality and integrity of *SSApp*'s data. The second is the security arrangement which prevents an attacker from compromising *SSApp* protection module by depriving the kernel authority of the *ROS* (Rich OS). The third is the page table update verification module in *TOS* (Trusted OS) which traps the update of page table in *ROS* and handles with it based on the predefined security policies. The last one is the security policies module which prevents an attacker from tampering the code and data of *SSApp*. At the same time, it keeps the memory of *SSApp* from an attacker to guarantee the confidentiality of critical data. We have evaluated our prototype on a simulation environment by using ARM FastModel and presented our implementation on a real development by using ARM CoreTile Express A9x4. Our security analysis and experimental results show that *TZ-SSAP* can ensure the *SSApp* execute as expected even if the kernel is compromised.

Keywords: Security-sensitive application · *TZ-SSAP* · TrustZone

1 Introduction

As is known to all, applications are managed by OS (operating system), which always use large monolithic kernels that have complete access to and control over all system resources, including memory management, process scheduling and communication,

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2017
R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 538–556, 2017.
DOI: 10.1007/978-3-319-59608-2_30

device management, file management and so on [1]. A large amount of security defense systems are implemented based on the view that the OS is the trusted root.

However, there is no denying that there are many secure vulnerabilities [2–4] due to the large attack surfaces existed in the current OS kernel. This leads to the result that the application running on the OS is no longer safe since an attacker can exploit kernel vulnerabilities to escalate privilege or execute a rootshell. For example, attacker can tamper the kernel code or insert some malicious code through the data segment via PTMA [5] attack which can modify the attribute of physical pages by modifying the content in page table entry. Once the kernel is compromised, attacker can control the kernel to compromise applications. For example, it can manipulate the return value of system services to attack applications, which is named Iago attack [6]. Furthermore, attackers can freely acquire all the sensitive information belonging to security-sensitive applications by accessing main memory or intercepting the control or data flow of the applications, which can be achieved by the address mapping manipulation attacks [7], such as mapping overlap attack, double mapping attack, mapping reorder and mapping release attack. Even worse, the kernel has become an equally attractive attacked target in the recent years. It is in urgent need of protection mechanism to make security-sensitive application remain safe even if the OS is compromised.

Previous research about application protection mechanism widely relies on hypervisor [14, 15, 17, 18]. Most of them use extended page table to provide an isolated environment for sensitive applications. When the application interacts with the OS, hypervisor has to verify the legitimacy of the operation. However, they ignore the integrity of kernel code. Despite the fact that Virtual Ghost [19] interposes a thin hardware abstraction layer to intercept instruction of kernel which can prevent unauthorized code from executing, it still have drawbacks. For example, it depends on new instruction set and compiler and all operating system software has to be compiled again.

In this paper, we have proposed a secure framework named *TZ-SSAP* based on hardware-assisted environment provided by TrustZone technology. It provides a strong protection mechanism for security-sensitive applications (*SSApp*s). Unlike previous protection mechanisms which need to establish an external page table to protect sensitive applications, our prototype does not modify the existing page table and our design is suitable to the current commercial OSes. *TZ-SSAP* traps all updates of the page table in the *ROS* (Rich OS). The result is that the *ROS* has no right to tamper the page table limited to its write protection mechanism. And each update of page table all follow those rules: Firstly, physical pages of code and static data in kernel space is mapped read only; and the rest of data pages is mapped non-executable forever. Second, the physical pages belonging to *SSApp*'s user space will never be mapped to the normal applications, vice versa. In the end, kernel stack of *SSApp*s will be mapped read only when their state switches from running to other during process scheduling.

To summarize, we make the following contributions:

- We enforce our security policies based on the existing page table mechanism in the current OSes without extending page table, which has little modification to the *ROS*. Therefore, our prototype is suitable to the existing commercial OSes.

- We secure the execution environment for *SSApp*s. Our design framework ensure the integrity of all the code and the kernel static data used by *SSApp*. In other words, the *SSApp* will always remain safe even if the *ROS* is compromised or even crashed. Furthermore, *TZ-SSAP* can guarantee the confidentiality of *SSApp*'s data because it prevents malicious process from accessing the memory of *SSApp*.
- *TZ-SSAP* does not need to encrypt and hash any application pages which is accessed when the *ROS* is running. And it also does not need to validate the legitimacy of the parameter when it interacts with the *ROS*.
- *TZ-SSAP* is safer than previous work which relies on hypervisor. It has been implemented in the hardware-assisted isolated environment, so it is enough safe to defense these attacks from the malicious *ROS*.

In the next section, we introduce something about application and our experiment platform. Section 3 gives our threat model and some assumptions of *TZ-SSAP*. Then Sect. 4 describes the *TZ-SSAP* design while Sect. 5 presents its implementation mode. Section 6 discusses the security and performance of *TZ-SSAP*. Finally, we also describe the related works at present in Sect. 7 and give a conclusion in Sect. 8.

2 Background

2.1 Application Analysis

There is no doubt that applications are made up of codes and data, which is illustrated in Fig. 1. The first one states the implementation of the functionality it absolutely needs while the last one is the carrier of sensitive information and the direction of control flow. Therefore, it can ensure the security of applications if we can guarantee the safe of its codes and data.

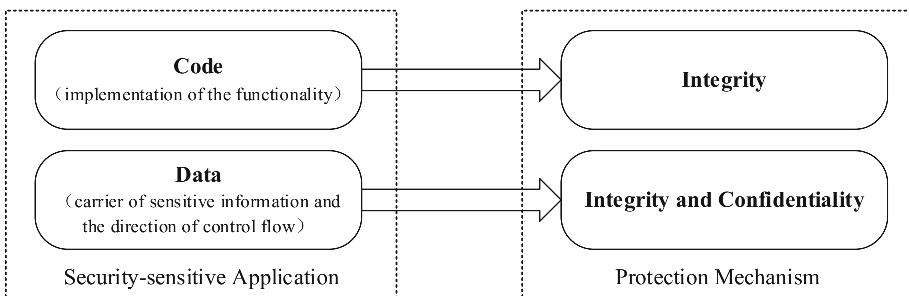

Fig. 1. Application architecture

Each application's code consists of three parts: basic code written by developer, the standard library code and the kernel code when it is in kernel mode. It is clear that the basic code is private and the kernel code must be shared with other applications. As for standard library code, it will be shared with other applications when compiled dynamically, and will be private like basic code while compiled statically.

For application's data, it consists of two parts: the kernel data and the user data. The kernel data contains the dynamic data in application's kernel stack and the static data. Meanwhile, the static data is initialized when the OS starts and shared with other applications, such as the system call table and the exception vectors table. Despite the fact that each process has unique kernel stack to store dynamic data produced by application during the run time, the data in kernel stack still can be accessed by other processes because all processes share the same kernel page tables. Moreover, the user data belonging to application is private as the basic code of application since they are stored in the user space which is separated from other applications normally by OS.

2.2 ARM TrustZone

ARM TrustZone [10, 11] technology is a set of security extensions first added to ARMv6 processors. Its architecture is illustrated in Fig. 2. Based on hardware logic present in AMBA bus fabric, peripherals and processors, it partitions the computing platform into two execution domains: *SW* (the Secure World) and *NW* (the Normal World) and partitions system resources into two parts: the non-secure resources and the secure resources. The OS running in *NW* is named *ROS* while *TOS* is running in *SW*. *ROS* can only access non-secure resources whereas *TOS* can see all resources. *TZ-SSAP* uses this feature to manage the page table mechanism in *ROS*.

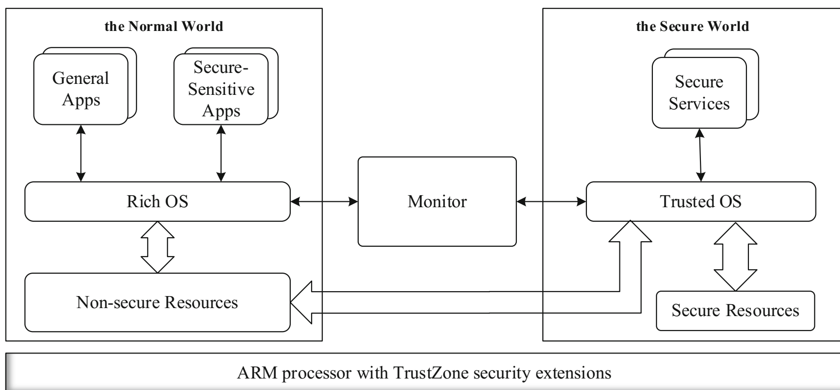

Fig. 2. TrustZone architecture

To control the context switch between the two worlds, a special processor mode, known as the monitor mode, is added by TrustZone. The monitor mode resides in *SW*, and maintains the processor state during the world switch. To trigger the entry to monitor mode, *ROS* or *TOS* can execute a Secure Monitor Call (*SMC*) instruction. Therefore, *TZ-SSAP* can switch between *ROS* and *TOS* through *SMC* instruction. Monitor can acquire the current context of the domain through the Non-secure (NS) bit from the Secure Configuration Register (SCR). That's to say, monitor can access *NW* if

the NS bit is set. *TZ-SSAP* takes advantage of this to save the coprocessor CP15 of *NW* when it acquires the context of data abort exception.

Our software is the OV [13], which is the first open source and free implementation for ARM TrustZone. When the system is power on, it starts from BootROM security. Once the *TOS* is running, it will establish security perimeter and perform key operations such as decrypting *NW* OS images. And before activating *NW* bootloader, keys, media and other assets are fully protected.

3 Threat Model and Assumption

We briefly describe our threat model and assumption in this section. Our goal is to protect *SSApp*s by guaranteeing the integrity of *SSApp*s' code and data as well as the confidentiality of *SSApp*s' data. We assume the *SSApp* has strong sense to protect its derived data using encryption techniques, for example, it can encrypt the file contents before writing to disk, and use existing secure I/O path schemes like [8, 9] to protect I/O data which used by peripheral devices, such as the fingerprint reader and keyboard. Thus, attacks against the *SSApp* itself are not in our consideration. We assume that *SSApp*'s base code is bug-free, thus to say, it will be carefully designed and tested in order to achieve high confidence in its own security.

Besides, we also assume that *SSApp* is static-compiled. In other words, *SSApp* will not share the standard library code with other processes, which can avoid malicious applications attacking it by tampering the standard library code. And the LKM (Loadable Kernel Module) is outside the scope of the current work.

We assume that the hardware implements the TrustZone extensions, and can be trusted with no Trojan-Horse circuits and no bus traffic interception and so on. Both *ROS* and *TOS* have been loaded securely which is guaranteed by the trusted boot. The worst thing of all is that it cannot guarantee the security of *ROS* during its run time since the kernel in *ROS* is vulnerable because of those discovered vulnerabilities, such as [5–7]. The attacker may use existing attack methods to damage *SSApp*, such as PTMA attack, Iago attack and address mapping manipulation attacks.

Moreover, the security-sensitive feature varies with the user. In this paper, we assume that the user has established a whitelist about *SSApp*s that should be protected by *TZ-SSAP*. To prevent attacker modifying the whitelist, the user uses the *TOS* to encrypt and hash it in a relatively safe environment. And *TOS* will decrypt the whitelist and check the hash whenever it starts.

4 TZ-SSAP Design

TZ-SSAP is implemented on TrustZone which provides a hardware-assisted isolation environment. According to application analysis in background, we can know that *SSApp* consists of code and data. In the following, we put forward the *SSApp* protection module in accordance with the characteristics of *SSApp*'s code and data. Then we present the security arrangement including page table update and process schedule in *ROS*. Afterwards, we propose how *TZ-SSAP* traps all updates of the page table and the

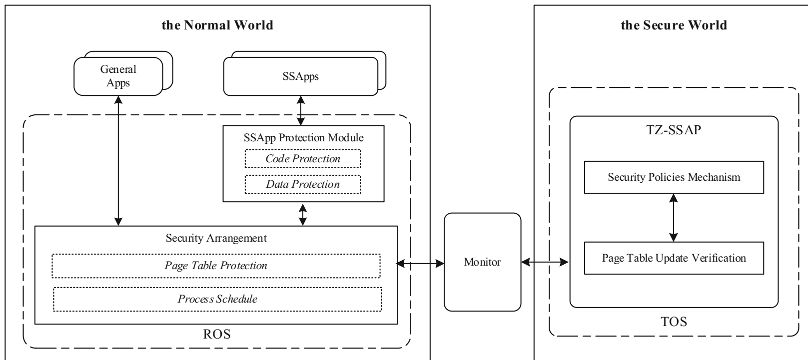

Fig. 3. TZ-SSAP system architecture

security policies mechanism that *TZ-SSAP* uses them to verify the operation of the kernel in *ROS*. *TZ-SSAP* prototype as illustrated in Fig. 3.

4.1 SSApp Protection Module

Code Protection. *SSApp* is static-compiled according to assumption. Thus, it only contain the private code and the kernel code. The private code will not be changed since it has become an executable file through compiling and linking. However, it is different for the kernel code. In Linux, administrator may load LKM due to some special requirements. In that case, additional code will be injected into the kernel at run time. For convenience, we do not take LKM into consideration. Therefore, the kernel code will remain unchanged during the system operation as well as the private code.

In order to protect integrity of the code executed by *SSApp*, *TZ-SSAP* maps all physical pages of code to read-only. Meanwhile, it must make sure that there won't be any writable map of them to prevent attacker tampering them. Those can be achieved through the page table management mechanism, which defines the virtual to physical address mapping and the access permissions of virtual memory in *ROS*. That's to say, *TZ-SSAP* modifies the access permissions of those physical pages so that they are write-protected and traps all updates of page table to avoid writable mapping of them.

Data Protection. According to application analysis, the user data is private in the whole operation period. Therefore, the integrity and confidentiality of the user data can be guaranteed as long as *TZ-SSAP* prevents malicious process from accessing them. We know that the memory can be accessed by OS only if it is mapped to the virtual address space. Therefore, *TZ-SSAP* can keep the *SSApp*'s user data from attacker by preventing double mapping the memory holding *SSApp*'s user data to general applications.

TZ-SSAP takes different measures to protect the integrity and security of *SSApp*'s kernel data. For static data, *TZ-SSAP* protects it by mapping its physical pages to

read-only as the kernel code. And for dynamic data in kernel stack, it cannot be write-protected since it may be changed when *SSApp*'s process is running. Similarly, it cannot prevent other process accessing them because they share the same kernel page tables. For those reasons, *TZ-SSAP* makes the *SSApp*'s kernel stack to be write-protected when the process's state switches from running to other during process scheduling. As a result, other process cannot tamper it via kernel page table, which can ensure the integrity of *SSApp*'s control flow in kernel mode.

4.2 Security Arrangement

Page Table Protection. As mentioned above, both code protection and data protection of *SSApp* are based on the page table management mechanism which is managed by the kernel in OS. When a new process is created, kernel establishes a new set of page tables for it. And after that, the kernel always keep updating it whenever process wants to change its map, such as applying for new physical pages, releasing old physical pages or modifying the attribute in page table entry. Unfortunately, because of the kernel ability to manage page table management, attacker can make *SSApp* protection module be out of control by attacking kernel through privilege escalation or PTMA attack.

To guarantee the security of *SSApp* protection module, we have to protect the page table management mechanism. Moreover, page tables are also in the form of physical pages in kernel. The updates of page table are normal memory writes which can be controlled by memory access permissions like code protection. And as described in background, *TOS* is capable of accessing the non-secure physical memory in *NW*. As a result, we replace kernel with *TZ-SSAP* to control the update of page tables. This can be achieved by modifying the access permissions of page tables to be read only in *ROS*. Besides, a great part of *TZ-SSAP* is realized in *SW*. Details are as follows.

In order to deprive the kernel's ability to update page tables, *TZ-SSAP* makes page tables of every process read-only once the process is scheduled into the *ROS* at the first time. Then *TZ-SSAP* intercepts the update of page tables through data abort whenever the kernel attempts to modify them. At that point, *TZ-SSAP* acquires the intention of kernel by decoding the instruction that generates the data abort exception in *ROS*. Furthermore, *TZ-SSAP* verifies the legitimacy of the instruction on the basis of the security policies. If it goes against any point of them, it will be rejected. On the opposite, *TZ-SSAP* will write the value to the corresponding address in *SW*.

Process Schedule. Trapping the update of page tables is critical to *TZ-SSAP*. Since *TZ-SSAP* completely makes page tables write-protected, the attacker can bypass *TZ-SSAP* through one possible method which is forging the whole page tables of process. In other words, attacker can forge a whole page table via rootkit and then set the base physical address of it to *TTBR0* (translation table base register 0) which stores the base address of page table in ARM Linux. On this occasion, *TZ-SSAP* will lose the ability to trap the update of fake page table since it not be write-protected by *TZ-SSAP*. As a result, attacker can steal the sensitive information and tamper the critical data structure or code of *SSApp* by double mapping *SSApp*'s physical page into fake page table.

To prevent these attack, *TZ-SSAP* must make sure that the value in *TTBR0* keep the same with which transferred to *TZ-SSAP* at the first time. *TZ-SSAP* enforces this policy by depriving the kernel from its own ability to set the value of *TTBR0* during the process scheduling. Once the kernel tries to schedule process, it must request *TZ-SSAP* for changing the value in *TTBR0*. In that case, *TZ-SSAP* can use this opportunity to verify the legitimacy of kernel operation in *ROS*.

4.3 Page Table Update Verification

TZ-SSAP gets the ability to update all page table of process running in the *ROS* since the physical pages of page table are write-protected. In order to verify the legitimacy of kernel operation, *TZ-SSAP* must get the content of the instruction that caused the fault. Then, *TZ-SSAP* decodes this instruction and retrieve it on the basis of the data abort exception context. Finally, *TZ-SSAP* takes action based on the result whether it goes against security policies mechanism. Security policies mechanism is described in Sect. 4.4 and implementation is described in Sect. 5.

4.4 Security Policies Mechanism

TZ-SSAP enforces the following policies for protecting the *SSApp* whenever the kernel attempts to update page table or schedule process.

- Write-protection. The access permissions of all process's page tables play an important role in *TZ-SSAP*. Not only physical pages of code and static data in kernel mode should be write-protected, but also all page tables should be non-writable in *ROS* at run time. Any writable maps of write-protected pages or operations changing the read only attribute of kernel space into writable should be prohibited.
- Double-mapping. It's important to make sure that any physical page of *SSApp* does not exist double-mapping to general applications. When the *SSApp* tries to map a new physical page to its address space, *TZ-SSAP* must guarantee that it does not exist virtual to physical mapping. In this way, we can easily keep security-sensitive data from attacker, which ensures the integrity and confidentiality of those data.
- Executable attribute. In order to prevent attacker from inserting some malicious code through its data area and then making it executable by attacks such as PTMA, operations that change those non-executable attribute to executable should be prohibited.
- Inherited attribute. It's clear that child process which is created by parent process belonging to *SSApp* is also security-sensitive, and the same with normal process. Any protection mechanism for parent process is still suitable for its child process.
- Share property. All *SSApp*s can share protected physical pages with each other. However, it should be rejected between *SSApp*s and general applications since it may damage protection mechanism.
- Zero clearing. Whenever *SSApp* requests physical page frames from the *ROS*, it's necessary to verify that all virtual to physical mappings for this frames have been removed. Afterwards, it cannot be neglected to clear the frame's contents since

malicious applications may have injected some threat thing. Moreover, *TZ-SSAP* also zero out the physical frame's contents when it's no longer needed by *SSApp*s.

5 Implementation

We have implemented *TZ-SSAP* on the ARM platform which supports TrustZone hardware extensions. In order to facilitate debugging, we used Fastmodel to emulate ARM Cotex-A15 in the beginning. Finally, we presented *TZ-SSAP* on a TrustZone-enabled development board ARM CoreTile Express A9x4 [12]. Our software experimental environment is Open Virtualization [13], which is the first open source and free implementation for ARM TrustZone.

5.1 Foundation Work of TZ-SSAP

Splitting Section Mapping. According to Sect. 4, we learn that *TZ-SSAP*'s design idea is mainly concentrated on write-protection method to deprive the kernel from its ability to update the page tables. Only under such circumstance can *TZ-SSAP* prevent malicious process from accessing the private physical pages of *SSApp* and tampering the share content in kernel space via damaging the page table mechanism. Unfortunately, the kernel space is mapped in section, which just use the first-level table and each entry of it consists of 1 MB blocks of memory in ARM-Linux OS. However, the user space is converted to small page mapping through the second-level table. In other words, the first-level descriptors contains the pointers to a second-level table for a small page, which consists of 4 KB blocks of memory, instead of the base address and translation properties for a section. Meanwhile, the page tables allocated by kernel are page-aligned small page, and the physical page of page tables must be mapped in kernel space to prevent unauthorized tampering with the user. Therefore, if *TZ-SSAP* makes the memory area where the page table information is stored read only through translation properties in first-level table, the 1 MB blocks of memory mapped by the entry will all become read-only. The trouble is that we cannot make sure the content stored in the 1 MB blocks of memory are page table information since all small pages are allocated dynamically. It may cause the system to crash if there are some dynamic data.

That problems described above can be solved by either one of two ways. First, we can aggregate the small page belonging to page tables of all process so that they are in a 1 MB blocks of memory. Only in this way can we make the 1 MB blocks of memory read-only through first-level descriptors. Second, we can change the section map into small page map in kernel space by modifying the kernel initialization code. In order to facilitate subsequent operation, *TZ-SSAP* directly change the map mode into small page and map the kernel code and static data to read only at the same time.

TZ-SSAP Interaction Between the ROS and TOS. In our implementation, *TZ-SSAP* uses *SMC* instruction to switch between *ROS* and *TOS* and passes parameters through

general-purpose register. Specific implementation details are as follows. When switching from the *ROS* to *TOS*, *TZ-SSAP* uses ARM core register R0 to transfer the variable corresponding to the *SMC* handler while register R1 transfers the parameters that required by *SMC* handler. Examples as below.

```
register u32 r0 asm(“r0”) = CALL_TRUSTZONE_FAULT;
register u32 r1 asm(“r1”) = param;
```

The content `CALL_TRUSTZONE_FAULT` in register R0 indicates that *TOS* will invoke the function corresponding to `CALL_TRUSTZONE_FAULT` after monitor mode switch to *SW*. The content param in register R1 indicates the parameters that *SMC* handler in *TZ-SSAP* requires. When the number of parameters is greater than one, we can transfer the parameters via memory since *TOS* is able to access all physical memory including the source in *NW*. In other words, *TZ-SSAP* in *ROS* declares a parameter structure, and uses register R1 to transfer the physical base address of this structure. *TZ-SSAP* in *TOS* can acquire those parameters through mapping these physical address to its virtual address space.

The same goes with *TOS* switch to *ROS* when *TZ-SSAP* completed *SMC* handler.

Getting the Context of Data Abort Exception. It will generate a data abort exception whenever the kernel of *ROS* tries to update the page tables since they are read only. Next, the kernel saves the context of data abort exception and jumps to exception handler which has been set in the exception vector table. As a result, it will execute *SMC* instruction which has been added in the `__do_kernel_fault()` by *TZ-SSAP* to enter to monitor mode. Then the monitor saves the context of the *ROS* and restores the context of the *TOS*. Finally, *TZ-SSAP* invokes the *SMC* handler related to the parameters transferred by the *ROS*.

The context of the *ROS* does not contain the coprocessor CP15 because most of the CP15 register are banked in *NW* and *SW*. It’s necessary for *TZ-SSAP* to access the coprocessor CP15 of *NW*. On one hand, *TZ-SSAP* has to get the value in *TTBR0* of *NW* to traverse the page table in *ROS*. On the other hand, *TZ-SSAP* has to set the value in *TTBR0* of *NW* due to it removes the ability to set *TTBR0* from kernel in *ROS* to prevent attacker from forging page tables. This can be achieved in monitor mode because monitor can access the coprocessor CP15 of *NW* when the *NS* bit in *SCR* is set. Pseudocode is as follows.

```
scr_nsbit_set          // set ns bit, NW
save_cp15_context
scr_nsbit_clear       // clear ns bit, SW
```

The procedures above proves that *TZ-SSAP* gets the context of *ROS* is the context of *SMC* exception instead of the context of the data abort exception. Therefore, we

must transfer them to the *TOS* in the form of parameter relating to *SMC* exception in order to get the specific content of the instruction which generated the data abort.

In ARMv7, OS will invoke `__dabt_svc` assemble function when it generates data abort in kernel mode. Moreover, the `__dabt_svc` calls the `svc_entry` assemble function to save the context of data abort exception in stack in the form of global structure `pt_regs` in kernel firstly. As a result, *TZ-SSAP* introduces a global variable `svc_dabt_sp` to store the base address of the context in stack. *TZ-SSAP* in *TOS* can acquire the context of data abort exception in *ROS* by mapping `svc_dabt_sp` into its virtual address space. In order to get the instruction that has generated the data abort exception, *TZ-SSAP* must get the address of the instruction. Besides, the register *LR* is a special register which holds return link information. However, in ARMv7 architecture, the *LR* register is banked in supervisor mode and abort mode, which of them are named `lr_svc` and `lr_abt`. Despite the fact that the monitor has saved both `lr_svc` and `lr_abt`, the value in `lr_abt` is no longer the address to be restored since it has been used as a general register in abort mode after kernel saved it in abort stack. To solve this problem, *TZ-SSAP* in *ROS* changes the *CPSR* (the current program status register) mode field into abort mode through *CPS* (change processor state) instruction and then recovers `lr_abt` from the stack in `__do_kernel_fault()`. After doing it, *TZ-SSAP* changes the *ROS* mode into supervisor and executes *SMC* instruction to enter monitor mode.

5.2 Security Policies Implementation

It's obvious that our main work is to ensure page tables all write-protected and secure update at the same time. In our implementation, we built an array named `ns_phy` to store the information of physical pages in *ROS*, which is similar to physmap in TZ-RKP [21]. Each entry of `ns_phy` corresponds to a 4 KB physical page of *ROS*. The value of the entry is a 32 bits integrity that indicates the state of this physical page. Besides, `ns_phy` is initialized to zero by *TZ-SSAP* at beginning.

TZ-SSAP divides the physical pages into four types by using a flag which is the bits [1:0] in the value. The value format as illustrated in Fig. 4.

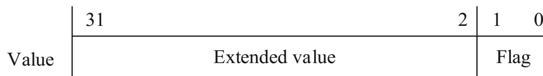

Fig. 4. Value format

- 0b00: Unmapped. The physical page is unmapped. The extended value is invalid.
- 0b01: Normal. The physical page is mapped by general applications and the extended value represents how many virtual page mappings the physical page has.

- 0b10: Protected. The physical page is mapped by *SSApp*s or used by the code or static data of kernel. The extended value is the same with 0b01.
- 0b11: Page table. The physical page is mapped as the page table of each process. In this case, the front 20 bits of extended value marks the base address of its first-level page table.

Process Schedule. To prevent attacker from forging the whole page table, *TZ-SSAP* must guarantee that the value wrote to *TTBR0* is authorized during process schedule in *ROS*. And when the security-sensitive process is scheduled to go out, in other words, its state is switched from running to other, *TZ-SSAP* must map the kernel stack of *SSApp* read only to avoid being tampered by malicious process.

In *TOS*, *TZ-SSAP* establishes a read-black tree structure `process_info` to store critical information of running process in *ROS*.

```
struct process_info{
  struct rb_node rb_node;
  Type process_task;
  Type process_ttbr0;
  Type process_thread;
  …
}
```

Variable `process_task` represents the base address of the data structure `task_struct` in *ROS*, which is the process descriptor in Linux system. Variable `process_ttbr0` represents the base address of the process's first-level page table in *ROS*. There are four small pages as the first-level page table for each process owing to the size of each process virtual address space is four Gigabytes. What counts is the four physical pages of first-level page tables are continuous. It's capable of *TZ-SSAP* in *TOS* to get the physical address of the process's first-level page tables in *ROS* according to the base address and size of each page. As pages are aligned, *TZ-SSAP* uses the last bit of `process_ttbr0` as flag to indicate whether the process is security-sensitive. Variable `process_thread` represents the virtual base address of the process's `thread_union`, which contains the process's kernel stack in *ROS*.

In *ROS*, the kernel will invoke `switch_mm()` function to complete the switch of process's address space by writing the physical base address of the first-level page table, belonging to the process to be executed, into *TTBR0*. The MMU (memory management unit) will convert the virtual address to physical address according to the value in *TTBR0*. To prevent attacker from forging the page table, *TZ-SSAP* inserts *SMC* instruction during the process is scheduled in *ROS*. Namely, *TZ-SSAP* updates *TTBR0* of *NW* instead of the kernel in *ROS*. When kernel invokes `switch_mm()` function, it will execute the *SMC* instruction and then the processor will switch to *SW*.

Algorithm 1. Process schdule

Input: *SMC parameter (process_task_n, process_ttbr0_n, process_task_pre …)*
1: *NodeN = FindInProcess_info(process_task_n)*
2: **if** *NodeN* != *NULL* **then**
3: *NodeNInfo = GetNodeInfo(NodeN)*
4: *UnEqual0 = Compare(process_ttbr0_n, NodeNInfo.process_ttbr)*
5: *UnEqual1 = Compare(process_ttbr0_n, NodeNInfo.process_thread)*
6: **if** *UnEqual0 or UnEqual1* **then**
7: *Record and Recover(NodeNInfo.process_ttbr, NodeNInfo.process_thread)*
8: **else**
9: *InsertNodeInProcess_info*
10: *PageTableWalk(process_ttbr0_n) and SetValueInNs_phy*
11: *MakeReadOnly(page table)*
12: *SetTTBR0(process_ttbr0_n)*
13: **endif**
14: *NodeP = FindInProcess_info(process_task_pre)*
15: **if** *NodeP* $\in$ *SSApps*
16: *MakeReadOnly(kernel stack)*
17: **endif**

Algorithm 1 is the *SMC* handler in *TZ-SSAP* when the process is scheduled by kernel in *ROS*. The `process_task_pre` is represented the process whose state will be non-running while the `process_task_n` will be scheduled to run in *ROS*.

Page Table Update. In ARM Linux, a data abort exception will be generated if the OS tries to access memory that it has no right to access it. As a result, the kernel will generate a data abort exception when it attempts to update page tables which is write-protected. To solve this exception, the kernel in *ROS* will call the corresponding exception handling function named `do_page_fault()`. The last function to be executed is `__do_kernel_fault()` because the virtual address that caused the data abort exception are in kernel space. In order to update the page table, *TZ-SSAP* inserts *SMC* instruction in the `__do_kernel_fault()` function of *ROS*. When the processor switches to *TZ-SSAP* through *SMC* instruction, it will invoke the *SMC* handler.

Algorithm 2. Page table update

```
Input: SMC parameter ( phy_base_addr….)
1:  ttbr0 = GetTTBR0
2:  context = GetContextOfDataAbort( phy_base_addr )
3:  content = Decode( fault instruction )
4:  Node = FindInProcess_info( ttbr0 )
5:  GetNodeInfo(Node)
6:  Pa = VaToPa( GetDFAR )
7:  if Pa ∈ FirstLevelPageTable then
8:        contentP = GetContent( Pa )
9:        if content is invalid then
10:             SetValueInNs_phy
11:             SetValueInPa( content )
12:       else
13:             VerifyWithSecurityPolicies( content, contentP )
14:             if authorized then
15:                   SetValueInPa( content )
16:             else
17:                   Record and Reject
18:             endif
19:       endif
20: else
21:             PaInfo = GetInfoInNs_phy( Pa )
22:             if (PaInfo ∈ page table ) and (Pa ∈ CurrentProcess )
23:                   VerifyWithSecurityPolicies( content, PaInfo )
24:                   if authorized
25:                         SetValueInPa( content )
26:                   else
27:                         Record and Reject
28:                   endif
29:             endif
30: endif
```

Algorithm 2 is the *SMC* handler in *TZ-SSAP* when kernel tries to update page table in *ROS*. The `phy_base_addr` is the base physical address of `pt_regs` which stores the context of data abort exception in *ROS*. When kernel tries to unmap a physical page, *TZ-SSAP* must change the value in `ns_phy` array to maintain consistency.

5.3 Performance Enhancement

One method that can improve performance is locality principle. During the process running, it may generate data abort exception successional because of page fault interrupt. Under this circumstances, it will always update page table entry located in

one physical page. So *TZ-SSAP* can apply a variable `fault_addr_latest` to record the latest address which generated the data abort exception and mark its attribute to indicate the address belongs to the second-level page table in *ROS* or not. When the data abort exception is generated again next time, *TZ-SSAP* will compare the `fault_addr_latest` with the content in *DFAR* (the Data Fault Address Register). It will reduce some operations to get the content in `ns_phy` array if the `fault_addr_latest` and the value in *DFAR* are in the same physical page.

6 Evaluation

TZ-SSAP provides a safe execution environment for *SSApp* to ensure it executes as expected even if the kernel is compromised. It achieves that by protecting the integrity of *SSApp*'s control flow and data flow. During runtime, *TZ-SSAP* keeps the *SSApp*'s memory from attacker through inserting *SMC* instruction and depriving the kernel of *ROS* from its own ability to set *TTBR0* and update page table in *NW*. In that case, *TZ-SSAP* can enforce our security policies mechanism whenever it traps *SMC* exception from the kernel. As a result, *TZ-SSAP* can prevent the attacker from tampering the code executed by *SSApp* and stealing critical information of *SSApp*.

TZ-SSAP can keep *SSApp* from the PTMA attack because the update of page tables must obey the security policies. For example, when the attacker attempts to change the read only attribute into writable in page table entry via PTMA attack, *TZ-SSAP* will intercept and verify it on the basis of security policies. It's clear that it will be rejected due to write protection in security policies, and the same with executable attribute in page table entry. As a result, *TZ-SSAP* can prevent attacker from tampering kernel code directly because of write protection. Besides, *TZ-SSAP* also prevent attacker from executing unauthorized code which is inserted into kernel space through malicious process's data segment since it is non-executable.

TZ-SSAP can defense the Iago attack. There is no doubt that *TZ-SSAP* can prevent attacker from modifying the kernel code and the static data in kernel as well as the data in kernel stack of the *SSApp*s. If the attacker tries to tamper those, it will generate a data abort exception. According to our implementation, *TZ-SSAP* will handle with this exception. At the beginning of the *SMC* hander in *TZ-SSAP*, it will verify whether the address generated the data abort exception belongs to the address range of current process's page table in *ROS*. *TZ-SSAP* will refuse to modify the content in the fault address if the operation is unauthorized.

What's more, *TZ-SSAP* can keep *SSApp* from the address mapping manipulation attacks. When *SSApp* requests physical page frames from *ROS*, *TZ-SSAP* will verify that there is not any virtual to physical mapping for this physical pages and zero out the content of this physical page. In additional, when general applications attempt to map new physical page, *TZ-SSAP* guarantees that the new physical page has not been mapped by *SSApp* or used as page table. Moreover, *TZ-SSAP* zeroes the physical page's content when changes its attribute as non-protected page.

To evaluate the security of *TZ-SSAP*, we built a malicious LKM that attempts to attack the TEST application which is designed as *SSApp* in our platform. The LKM

tries to directly tamper the code and static data in kernel space, such as `sys_call_table`. Also, LKM takes advantage of its privilege to write a fake value into *TTBR0*. Moreover, it also tries to double map the protected physical page into its virtual space. Firstly, we can get the base address of physical page in TEST through traversing its page table. Then, we change the source code of LKM to map the physical page to its virtual address. Our experimental results show that *TZ-SSAP* can prevent those attacks effectively.

And for performance evaluation, we define the consuming time in *ROS* from invoking the *SMC* instruction to backing from *SW*. Details as Table 1.

Table 1. Consuming time

Operations	Switching time	Traversing page tables	Update page table	Find page table entry
Consuming time	8us	257ms	345us	754us

According to Table 1, traversing page tables costs the most time. But it only take 6.7% for *SSApp*s whose executed time is only one minute. This percentage will be smaller and smaller as the executed time increases. Therefore, *TZ-SSAP* will not cause too much time loss for *SSApp*s.

7 Related Work

In recent years, there are several systems attempt to protect security-sensitive application code and data. We divide them into two classes in term of the way used to protect applications. One is the whole application protection which regards all the application code and data as a whole. The other is the split application protection achieved by protecting the critical part of the application instead of the whole.

7.1 Whole Application Protection

InkTag [14] is a virtualization-based architecture that uses a trusted hypervisor to isolate the *HAP* (high-assurance process) from OS which is achieved by the *EPT* (extended page tables). InkTag uses two separate *EPT*: the trusted *EPT* is installed for *HAP* execution while the untrusted *EPT* is used for OS and other applications. *HAP* updates the trusted *EPT* through hypercall. InkTag can defend again Iago attacks because of its paraverification to ease verifying of OS. Also it protects the confidentiality of secure page via encryption technology and detects corruption of the secure page through digital signing. However, it cannot keep the encrypted pages from reading and modifying by OS as well as Overshadow [15, 16] system which also need complex encryption and decryption technologies to protect the whole application execution. In addition, Overshadow cannot prevent the address mapping manipulation attacks.

Sego [17] is similar with InkTag for the *EPT*. But it is faster than InkTag since it does not need encryption technology. All secure data stays in plain text which is protected by hardware memory protection to ensure OS cannot access them. AppShield [7] is a hypervisor-based approach that reliably safeguards code, data and execution integrity of a critical application. It consists of two parts: a transit module in kernel space mediating control flow between the *CAP* (critical application) and OS, and a trusted shim in user space assisting the data flows between *CAP* and shared buffer. It can defense the address mapping manipulation attacks since the hypervisor and shim code jointly to protect the address space of *CAP* when OS updates the page tables. What is worse is that both of them cannot guarantee the integrity of kernel code.

7.2 Split Application Protection

SeCage [18] retrofits commodity hardware virtualization extensions to support efficient isolation of sensitive code manipulating critical secrets from the remaining code. It decomposes the applications into two parts. One is the secret compartment which contains a set of secrets defined by user as well as its corresponding code. Another is the main compartment that deals with the rest of the application logic. Firstly, it uses an analysis framework CI to discover potential functions related to secrets statically. And then it combines the `mprotect` and debug exception together to dynamic analysis with different workloads. Once the analysis result comes to a fixed point, SeCage decomposes the application to secret and main compartments on the basis of them. It also use the extended page tables to guarantee two compartments isolation. Despite the fact that SeCage can keep applications from many attacks, such as PTMA attack and the address mapping manipulation attacks, there are still some weaknesses. The closure related to secrets may be not complete since designers cannot be exhaustive of all possible input about applications. Besides, it also cannot guarantee the effectiveness of non-secret data accessed by secret functions due to no protection against them.

Virtual Ghost [19] is different with the above systems. It protects secret data and code with the ghost memory instead of *EPT*. The ghost memory is achieved by adding a compiler-based virtual machine (VM) between OS and hardware. All system software is compiled to the virtual instruction set based on the LLVM compiler intermediate representation [20]. Therefore, Virtual Ghost can prevent OS accessing the ghost memory since those virtual instruction set are implemented by VM and need to be validity verification, which can defense the address mapping manipulation attacks and prevent repurposing existing instruction sequences because of its control flow integrity enforcement. However, it depends on the virtual instruction set and compiler which are not always practical for current infrastructures.

8 Conclusion

In this paper, we have presents *TZ-SSAP*, which provides a safe execution environment for security-sensitive applications in the face of the OS may be compromised because of the kernel vulnerability. *TZ-SSAP* is implemented based on the hardware-assisted

isolated environment TrustZone. The general OS is installed in *NW* while *TZ-SSAP* is mainly located in *SW*. *TZ-SSAP* takes advantage of the page table mechanism instead of the extended page tables. *TZ-SSAP* makes *SSApp* perform as expected since it can guarantee the integrity of both the code and its control flow during run time. In addition, it also guarantee the confidentiality of *SSApp*s' data through preventing attacker from double mapping the *SSApp*'s physical page, which leads to keep the security-sensitive information of *SSApp*s from attacker.

In future work, we can add safeguard to protect the code of standard library and LKM in *TZ-SSAP*. For example, we can verify the integrity of the library file or LKM module before it is loaded into memory, and make them write-protected once they are authorized and loaded into physical page.

Acknowledgments. This work is supported by the National High-tech R&D Program (863 Program) of China under Grant No. 2012AA01A401 and National Science and Technology Major Project of China under Grant No. 2013ZX01029003-001.

References

1. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel. O'Reilly Media, Sebastopol (2005)
2. https://www.exploit-db.com/
3. http://www.security-database.com/
4. http://www.securityfocus.com/
5. Lee, J.S., Ham, H.M., Kim, I.H., et al.: POSTER: page table manipulation attack. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 1644–1646. ACM (2015)
6. Checkoway, S., Shacham, H.: Lago attacks: Why the system call api is a bad untrusted rpc interface. ACM (2013)
7. Cheng, Y., Ding, X., Deng, R.H.: 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. ACM (2015)
8. 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
9. Zhou, Z., Gligor, V.D., Newsome, J., et al.: Building verifiable trusted path on commodity x86 computers. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 616–630. IEEE (2012)
10. ARM A R M.: Security Technology Building a Secure System Using TrustZone Technology (white paper). ARM Limited (2009)
11. ARM: TrustZone Introduction. http://www.arm.com/zh/products/processors/technologies/trustzone/index.php
12. ARM Limited: ARM CoreTile Express A9X4 Cotex-A9 MPCore (V2P-CA9) Technical Reference Manual (2014)
13. Sierraware: Open Virtualization Build and Boot Guide for ARM V7 and ARM V8 (2014)
14. Hofmann, O.S., Kim, S., Dunn, A.M., et al.: Inktag: secureapplications on an untrusted operating system. ACM SIGARCH Comput. Archit. News ACM **41**(1), 265–278 (2013)

15. Chen, X., Garfinkel, T., Lewis, E.C., et al.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. ACM SIGARCH Comput. Archit. News ACM **36**(1), 2–13 (2008)
16. Ports, D.R.K., Garfinkel, T.: Towards application security on untrusted operating systems. In: HotSec (2008)
17. Kwon, Y., Dunn, A.M., Lee, M.Z., et al.: Sego: pervasive trusted metadata for efficiently verified untrusted system services. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 277–290. ACM (2016)
18. Liu, Y., Zhou, T., Chen, K., et al.: Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 1607–1619. ACM (2015)
19. Criswell, J., Dautenhahn, N., Adve, V.: Virtual Ghost: Protecting applications from hostile operating systems. ACM SIGARCH Comput. Archit. News **42**(1), 81–96 (2014)
20. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004 (CGO 2004), pp. 75–86. IEEE (2004)
21. Azab, A.M., Ning, P., Shah, J., et al.: Hypervision across worlds: real-time kernel protection from the arm trustzone secure world. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 90–102. ACM (2014)