SCIENCE CHINA
Information Sciences

TZ-Container: Protecting Container from Untrusted OS with ARM TrustZone

Hua Zhichao, Yu Yang, Gu Jinyu, Xia Yubin, Chen HaiBo and Zang Binyu

Citation: SCIENCE CHINA Information Sciences; doi: 10.1007/s11432-019-2707-6

View online: http://engine.scichina.com/doi/10.1007/s11432-019-2707-6

Published by the Science China Press

SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

TZ-Container: Protecting Container from Untrusted OS with ARM TrustZone

Zhichao Hua[1], Yang Yu[2], Jinyu Gu[1], Yubin Xia[1*], Haibo Chen[1] & Binyu Zang[1]

[1]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai* 200240*, China;*
[2]*Shanghai Gejing Information Technology Co., Ltd., Shanghai* 200240*, China*

Abstract Containers are widely deployed on cloud platforms because of their low resource footprint, fast start-up time, and high performance, especially compared with its counterpart virtual machines. However, the Achilles' heel of container technology is its weak isolation. For an attacker, jailbreaking into a host OS from a container is relatively easier than attacking a hypervisor from a virtual machine, because of its notably larger attack surface and larger trusted computing base (TCB). Researchers have proposed various solutions to protect applications from untrusted OS; yet, few of them focus on protecting containers, especially those hosting multiple applications and shared by multiple users. In this paper, we first identify several new attacks that cannot be prevented using the existing solutions. Furthermore, we systematically analyze the security properties that should be maintained to defend against these attacks and protect a full-fledged container from a malicious host OS. We then present the TZ-Container, a TrustZone-based secure container mechanism that can keep all these security properties. The TZ-Container Specifically leverages TrustZone to construct multiple isolated execution environments (IEEs). Each IEE has a memory space isolated from the underlying OS and any other processes. By interposing switching between the user and the kernel modes, IEEs enforce security checks on each system call according to its semantics. We have implemented TZ-Container on the Hikey development board ensuring that it can support running unmodified Docker images downloaded from existing repositories such as *hub.docker.com*. The evaluation results demonstrate that the TZ-Container has a performance overhead of approximately 5%.

Keywords System Software, System Security, Linux Container, ARM, ARM TrustZone

Citation Zhichao Hua, Yang Yu, Jinyu Gu, et al. TZ-Container: Protecting Container from Untrusted OS with ARM TrustZone. Sci China Inf Sci, for review

1 Introduction

Container technologies such as LXC [25] or Docker [35] are often used in the cloud because of their low resource footprint, fast start-up, and ease of deployment. With ARM platforms gaining momentum in the server market [1,4,36], many companies have deployed ARM servers that run containers at scale [38,42]. It is natural for these companies to deploy containers on their ARM platforms. In fact, there have been many efforts to popularize containers for ARM platforms [2,5,8].

Unfortunately, compared with virtual machines, containers have a weaker isolation that depends on many security properties offered by the host OS. The problem is that the host OS kernel usually contains tens of millions of lines of code (thus thousands of bugs [6]) and becomes a single point of failure of the entire system. Hence, container isolation should be enforced *without* trusting the OS kernel.

Considerable research exists on how to protect applications from untrusted OSs [11,18,19] that can be repurposeed to protect containers. Systems such as CHAOS [18], Overshadow [19] and SP^3 [48]

* Corresponding author (email: xiayubin@sjtu.edu.cn)

prevent the OSs from reading or tampering with application's data (aka., *direct attacks*) by isolating them in different execution environments with a trusted hypervisor. SCONE [11] runs a Docker instance in a trusted execution environment based on Intel SGX [3]. However, most of these work focus on the isolation of applications' memory and the protection of applications' I/O data, while few of them consider attacks issued through legal interfaces of an OS, which are also known as Iago attacks [17].

Iago attacks leverage the application's assumptions on the system calls' semantics to let the application harm itself. For example, a malicious OS may return a wrong pointer as the return value of an *mmap* system call, which points to some return addresses on the application's stack. If the application does not verify the pointer, it may unintentionally change the return address and further violate its own control flow integrity. InkTag [28] and Sego [31] consider Iago attacks by verifying the results of system calls. However, these systems focused on protecting a single application instead of a more complicated container execution environment and did not fully consider the interactions between the OS kernel and containers, such as inter-process communication (IPC) semaphores.

In a container environment, multi-user and multi-application are essential. To offer an illusion that a container is the only environment running on a machine, Linux kernel introduces *namespace* mechanisms to let a container have its own namespaces of user, process, file system, etc. Many container applications depend on these namespaces for security. An example is enforcing *intra-container isolation* using a user access control mechanism. However, a malicious OS may leverage these assumptions to attack a container. For example, if a container is running a *sshd* service, an attacker may first login as a normal user *Eve* and then try to access */etc/passwd*. The operation should be denied because the file is only accessible to the root user of the container. However, if the attacker also controls the OS kernel, she can just give the user *Eve* the root privilege, so that *Eve* can access any file within the container, even if all the files are encrypted outside the container. We call such attacks *MUMA* (Multi-User Multi-Application) attacks and older systems cannot defend themselves from these attacks, as shown in Figure 1.

In this paper, we first analyze the current security mechanisms of containers and their dependency on the underlying OS kernel. Furthermore, we present new attacks that a malicious OS kernel may issue by breaking these mechanisms, including attacks on multi-application synchronization, inter-application communication, and user access control. We conclude a list of general security properties that should be ensured for container's protection. Then, we propose the TZ-Container, a system enforcing these properties by using the widely deployed ARM TrustZone hardware feature. The TZ-Container specifically leverages TrustZone to construct an isolated execution environment (IEE) for each container process. It also intercepts all interactions between processes and the kernel, verifies semantics of the interactions between multiple processes/applications and ensures the integrity of user access control.

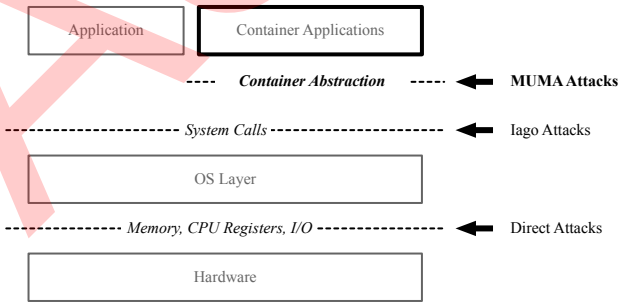


Figure 1 Layers of attacks. Previous researches usually focused on *direct attacks* and *Iago attacks [17]*. In this paper we target the *MUMA (multi-user multi-application) attacks* at the layer of container abstraction.

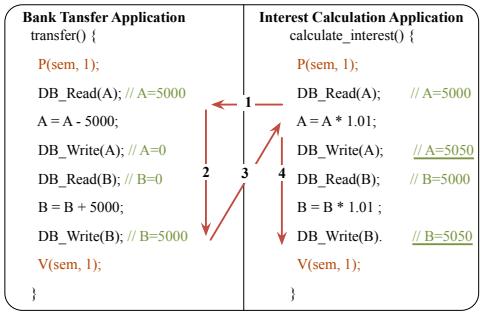


Figure 2 Sample code of multi-process synchronization attacks. The malicious OS ignores P() and V() operations of an IPC semaphore to violate the mutual exclusiveness of the two code snippets.

We have implemented TZ-Container on Hikey ARMv8 development board and integrated it with Docker-v1.10. TZ-Container can directly run unmodified container images. The evaluation results demon-

Table 1 Attack considerations. (✓ means one system considers the attack. ◯ means one system partially considers the attack.)

	Direct Attacks		**Iago Attacks**	**MUMA Attacks**		
	Memory/Context Attacks	Disk I/O Attacks		Multi-application Synchronization Attacks	Inter-application Communication Attacks	User Access Control Attacks
Attack Apps	Has	Has	Has			
Attack Containers	Has	Has	Has	Has	Has	Has
SICE [13]	✓					
Fides [40]	✓					
TrustICE [41]	✓					
Overshadow [19]	✓	✓				
SP^3 [48]	✓	✓				
Virtual Ghost [22]	✓	✓				
MiniBox [32]	✓	✓				
InkTag [28]	✓	✓	✓			◯
Sego [31]	✓	✓	✓			◯
SecureME [20]	✓	✓			◯	
Haven [14]	✓	✓	✓			
SCONE [11]	✓	✓	✓			
Graphene-SGX [44]	✓	✓	✓			◯
TrustShadow [26]	✓	✓	✓			
gVisor [9]			✓			
TZ-Container	✓	✓	✓	✓	✓	✓

strate that the proposed system introduces only a negligible performance overhead. The performance slowdown is approximately 5% for common server applications (e.g., Apache and Redis).

In summary, this paper makes the following contributions:

- A systematic analysis on protecting containers from an untrusted OS. We highlight the presence of MUMA attacks that previous systems do not explicitly consider.
- A method for constructing multiple isolated execution environments (IEEs) for different container processes in the normal world using ARM TrustZone technology.
- Design of the TZ-Container to protect containers on untrusted OSs from MUMA attacks without requiring any modifications of existing hardware or container images.
- Implementation of the TZ-Container on real hardware and software to demonstrate the effectiveness and efficiency of the proposed design.

2 Motivation

The Linux container is an OS-level virtualization technology that has become increasingly popular for packaging and deploying services such as key/value stores and comprehensive web services. To enforce isolation between containers, the Linux kernel introduces six namespace mechanisms that isolate 1) the hostname and domain name, 2) the root file system, 3) users and groups, 4) inter-process communication (IPC) instances, 5) process ID, and 6) the IP address and port. This is because traditional process abstraction is not adequate for containers, which require an environment with *multiple users and multiple applications*.

Our goal is to protect containers from the untrusted OS. One straightforward solution is to retrofit existing work on protecting single application from untrusted OS (e.g., Overshadow [19] and InkTag [28]). However, this is not adequate to protect a full-fledged container. We will demonstrate the differences between protecting an application and protecting a container and highlight the presence of some new attacks such as **MUMA attacks** that are not explicitly considered by previous work.

2.1 OS Attacking a Single Application

A malicious OS has various methods to attack a user application. They can be divided into the two classes: direct attacks and Iago attacks. Other attacks, including side-channel attacks and DoS attacks, are not considered in this paper.

Direct Attacks: A malicious OS can directly access or control the memory pages, CPU context or I/O data to attack an application. The memory and CPU context can be protected by maintaining an execution environment isolated from the OS. Previous researchers have proposed many systems to defend against direct attacks [11, 18–20, 22, 28, 31, 48]. The disk I/O can be protected by encrypting and hashing all file contents [18, 19]. The network I/O can be protected by applications with end-to-end protocols such as SSL.

Iago Attacks: An OS can attack an application by providing malicious return values of syscalls, which are also known as Iago attacks [17]. Syscalls, such as *getpid* and *mmap* can be used to perform Iago attacks. Existing works [28, 31] propose a defence against Iago attacks by verifying the results of some syscalls.

2.2 OS Attacking a Container

Unlike a single application, a container is a *multi-user, multi-application* environment relying on OS's services (i.e., multi-application synchronization, inter-application communication and user access control) to make all users and applications inside the container to correctly cooperate with each. By controlling these services, an untrusted OS can issue **MUMA (Multi-User Multi-Application) attacks**, which includes: *Multi-application Synchronization Attacks*, *Inter-application Communication Attacks* and *User Access Control Attacks* (as shown in Table 1).

Multi-application Synchronization Attacks: Different applications use synchronization interfaces provided by the OS kernel (e.g., IPC semaphores) to control the execution flow. A malicious OS may trigger race conditions by violating the synchronization semantics. For example, consider two server applications running in a container, where one is responsible for bank transfers, while the other is responsible for interest calculation.

They access the same database and use a semaphore to prevent a race condition. An attacker A, who has already compromised the OS kernel, sends a request to transfer \$5000 to B (initially, the account balance of B is 0 and that of A is 5000). The compromised OS may not keep the semantics of the IPC semaphore, which may lead to the control flow shown in Figure 2. As a result, both A and B will obtain \$5050 in the end.

Besides semaphore, many other synchronization interfaces exist such as signal, *wait* and *flock*. Sego [31] protects unnamed semaphores that cannot be used between multiple applications. Graphene-SGX [44] provides a secure semaphore between parent and child processes.

Inter-application Communication Attacks: Two applications, A and B, can build a communication channel, e.g., message queue or shared memory, for exchanging data. When message queue is used, messages are vulnerable to a malicious kernel since all the data are delivered through the kernel. If using shared memory, a malicious OS could fool both A and B that they have established a shared memory region but actually not, and further performs *forking attacks*. For example, consider A and B as two processes that share a database of bank accounts. Data are stored in shared memory. However, an untrusted kernel can make A and B have their own copies of data (i.e., no sharing). Thus, an attacker may withdraw money first from A and then from B to obtain twice as much as she actually owns.

Overshadow [19] and SP^3 [48] claim to support IPC including shared memory. However, the granularity of sharing is too coarse-grained: two processes can either share nothing or everything. It means that if A and B share one memory page, the malicious OS can map any pages of A to B. This coarse-grained method cannot be used between different applications. SecureMe [20] claims to protect IPC shared memory with more fine-grained granularity of sharing. However, it cannot defend against the *forking attacks* mentioned above, and it does not protect other communication channels, e.g., message queue.

Table 2 Security properties for protecting a container.

	Security properties to be enforced
Memory & CPU Context	**P-1.1**: OS cannot access container process's memory.
	P-1.2: OS cannot tamper with container process's CPU context.
	P-1.3: OS can only enter the container process from fixed points.
Disk I/O	**P-2.1**: OS cannot break the confidentiality and integrity of container file.
	P-2.2: One container's file cannot be accessed by any other container.
Defending Against Iago Attacks	**P-3.1**: OS cannot arbitrarily return value for syscalls.
Multi-application Synchronization	**P-4.1**: OS cannot tamper with the functionality of semaphore.
	P-4.2: OS cannot arbitrarily inject signal to container process.
	P-4.3: OS cannot tamper with the functionality of *flock*/*futex* syscalls.
Inter-application Communication	**P-5.1**: Enforce the integrity and confidentiality of the communication data
User Access Control	**P-6.1**: The permission bit of file and IPC instance cannot be tampered with.
	P-6.2: The permission of each container process cannot be tampered with.
	P-6.3: Only the process with correct permission can access a file or an IPC instance.
	P-6.4: Only the process with correct permission can send a signal.

User Access Control Attacks: The security of a container heavily depends on the access control mechanisms provided by the kernel. For example, both the Apache and Nginx run worker processes under a new user, *www-data*, which has limited permissions to handle user requests. Meanwhile, the master process may run with root permission. This is a lightweight sandboxing mechanism ensuring that even if a worker process has security vulnerabilities and is controlled by an attacker, it is still restricted. However, a malicious OS may collude with a malicious application and deliberately loose the control over it, e.g., to grant it root user privileges.

Mainly, the three access control mechanisms in Linux: are for the file system, IPC, and signal. InkTag [28] and Sego [31] implement file system access control in a trusted hypervisor. Graphene-SGX [44] only allows applications to access files specified by a manifest. However, they require the user to additionally claim the access permission and cannot protect other access control mechanisms such as IPC instance or signal delivery.

2.3 Goals of TZ-Container

To enforce the security of containers, both single application attacks (direct attacks and Iago attacks) and MUMA attacks must be considered. However, as shown in Table 1, none of the existing work propose a defence against the three types of MUMA attacks. This is mainly because all them focus on protecting a single application (or a container with a single application).

The goals of the TZ-Container are to defend against direct attacks, Iago attacks and the MUMA attacks. We list the required security properties in Table 2. To defend against direct attacks, multiple isolated execution environments must be created to protect the memory and CPU context, and the disk I/O must be protected. To detect Iago attacks, the return values of syscalls should be verified. Currently, we have identified three types of MUMA attacks, which are mentioned above. To defend against them, the TZ-Container must enforce the security of multi-application synchronization, inter-application communication, and user access control.

Besides these security properties, the TZ-Container must offer high performance and good compatibility. Furthermore, it should support existing container images to make the security mechanism transparent to end users.

3 System Overview

3.1 Background on ARM TrustZone

TrustZone [10] is a hardware security mechanism covering the processor, memory and peripherals. The processor is split into two execution environments, a normal world and a secure world. Both worlds have their own user mode and kernel mode, together with cache, memory and other resources. The normal

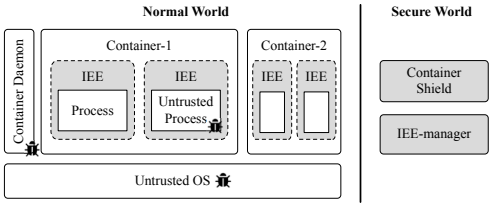


Figure 3 Design overview of TZ-Container. Each container process is protected by an isolated execution environment (IEE) in the normal world. Each IEE is maintained by an *IEE-manager* running in the secure world. The *container shield* defends against Iago attacks and MUMA attacks.

world cannot access the secure world's resources (e.g., secure memory), while the latter *can* access all resources. Based on this asymmetrical permission, the normal world is used to run a commodity OS. Meanwhile, the secure world can locate a secure service. The two worlds can switch to each other using a special instruction called "secure monitor call" (*smc*).

3.2 Threat Model

We assume that the OS is completely untrusted, and may try to read or tamper with containers' memory and CPU registers, as well as data en route to I/O devices. Additionally, it may try to manipulate the return values of any system calls issued by containers, or violate the semantics of container abstractions to perform MUMA attacks. We consider a case where a container has multiple users and multiple applications. Some of the non-root users or processes may be controlled by the attacker. Moreover, a malicious process inside the container may collude with the untrusted OS to perform further attacks, e.g., obtaining higher privilege.

We assume that the hardware implementation is correct. Secure boot technology is used to protect the code integrity of Linux kernel during system boot. After that, the buggy kernel can be compromised. We also assume that applications in containers adopt protocols such as SSL to protect data transferred over the network. We trust the container client running on the user side. The TZ-Container does not consider the container application leaking its data, DoS attacks, side-channel attacks, physical attacks and reorder/speculative execution-based attacks (e.g., Meltdown [33]).

3.3 Design Overview

Figure 3 shows an overview of the design. ARM TrustZone only provides a single secure world. To protect the memory and CPU context of container processes (**P-1.1**∼**P-1.3**), the TZ-Container constructs multiple isolated execution environments (IEEs) in the normal world through an *IEE-manager* running in the secure world. The *IEE-manager* exclusively controls the entire system's memory mapping and enforces memory isolation. Additionally, it intercepts all switches between the user and the kernel in the normal world for protecting IEEs' registers and hooks all the system calls.

The *container shield* in the secure world enables parameter delivery from the IEEs to the untrusted OS and checks the system calls. It ensures the integrity and confidentiality of the disk I/O by cryptographic methods and defends against existing Iago attacks by checking the return values of corresponding syscalls (**P-2.1, P-2.2, P-3.1**). It also prevents *MUMA attacks*, including protecting *multi-application synchronization*, *inter-application communication* and *user access control*, by tracing corresponding syscalls and verifying their behaviors (**P-4.1**∼**P-4.3, P-5.1, P-6.1**∼**P-6.4**). Once the *container shield* detects malicious behaviors, which violates the security properties, it stops the execution of the container. The *container shield* requires information across different IEEs; therefore, we place it in the secure world as an individual module instead of integrating it within an IEE. For better compatibility, the *container shield* provides interfaces for integrating with Docker.

4 Isolated Execution Environment

An IEE protects the memory and CPU context of a container process. Each IEE requires the following security properties:

- **Memory and CPU Context Isolation**: Both the memory and CPU context of an IEE should be isolated from other processes (including other IEEs) and the OS (**P-1.1** and **P-1.2**).
- **Fixed Entry Points**: An IEE can only start/resume from some fixed entry points (**P-1.3**).
- **Secure Identification**: An IEE should be securely identified by the *container shield* to prevent impersonation.

4.1 Memory Isolation of IEE

The *IEE-manager* isolates each IEE's memory by exclusively controlling the memory mappings and enforcing two policies: 1) an IEE's memory cannot be mapped to the OS and 2) an IEE's memory cannot be mapped to any other processes (except the IPC shared memory whose details are in Section 5.4).

To exclusively control all memory mappings, the *IEE-manager* deprives the OS of the ability to modify them. On ARM platform, the number of ways to modify mappings are limited. 1) Enabling/disabling a page table by maintaining instructions [1)] and 2) modifying page table entries. We modify the kernel to replace all page table maintaining instructions with invocations to the *IEE-manager*. The *IEE-manager* then marks the enabled page table as read-only. Thereafter, the kernel must invoke the *IEE-manager* to modify the table entries.

To prohibit the compromised OS from injecting page table maintaining instructions during runtime, the *IEE-manager* maps the kernel text section as read-only and enforces that it does not contain page table maintaining instructions. All the kernel data pages are mapped as eXecuted-Never and checked whether the kernel remaps them as executable, so that the compromised kernel cannot inject page table maintaining instructions to the data pages and jump to execute them. The user space memory is mapped as Privileged eXecute Never (PXN) to defend against return-to-user attacks. We remove all return-oriented programming (ROP) gadgets or jump-oriented programming (JOP) gadgets that can be used to form new page table maintaining instructions, which is relatively easy to perform on ARM platform because all the instructions are aligned. The kernel modules are checked before being installed.

4.2 CPU Context Isolation of IEE

The *IEE-manager* hooks all the switches between an IEE process and the kernel, and protects the privacy and integrity of the CPU context. On ARM platform, the only way to switch from the user to kernel mode is with an exception, which is handled by multiple exception handlers stored in an exception table. The table is pointed by an exception table base register (*VBAR_EL1*). We modify the kernel to invoke the *IEE-manager* to modify this register and mark the enabled exception table as read-only. Then, we inject a hook in each exception handler to interpose all kernel enter operations. Switching from the kernel to user mode is performed by some specific instructions (e.g., *eret*). We substitute all these instructions with invocations to the *IEE-manager*. The *IEE-manager* saves an IEE's context and clears it before switching to the kernel. Thus, the untrusted OS can only see a synthetic context and cannot tamper with the real one.

4.3 Fixed Entry Points of IEE

The TZ-Container defines three types of entries for an IEE: 1) *init entry*: the start function of the application; 2) *runtime entry*: it occurs during runtime where the IEE exits the user mode (e.g., where an interrupt happens); and 3) *user-defined entry*: the user-defined signal handler. The *IEE-manager* allows an IEE to be started only from these entries.

1) E.g., "*MSR TTBR0_EL1, Xt*" is used to enable a page table.

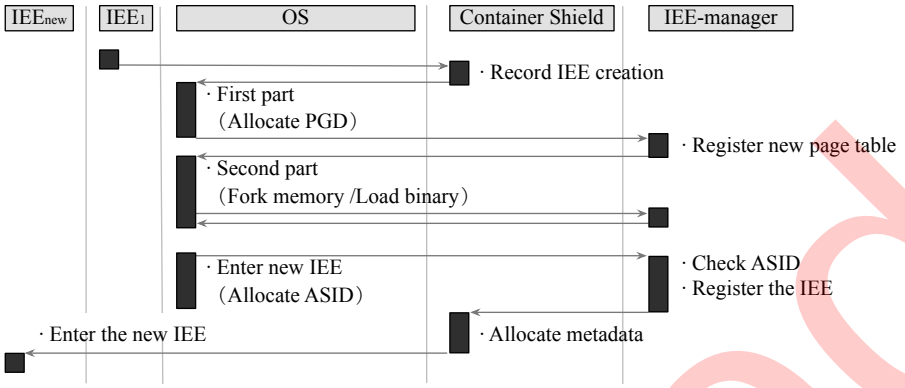


Figure 4 The procedure of creating an IEE. The kernel is responsible for creating a process, including constructing the page table. The created page table must be registered in the *IEE-manager*. Before entering a new process, the *IEE-manager* checks the page table and enforces the memory isolation.

4.4 IEE Creation and Identification

An IEE can be created by two methods: 1) invoking the *fork*/*exec* syscall by an existing IEE and 2) invoking the *exec* syscall with a new *IEE* flag by any process.

As depicted in Figure 4, the *container shield* records the IEE creation before forwarding the request to the untrusted OS. Then the OS handles the request as normal, including allocating the new page table. Subsequently, the OS registers this page table to *IEE-manager*, which checks whether there exists a matching IEE creation record and marks the new page table as read-only to the OS. Only when the registration succeeds, the OS can continue the creation of an IEE by mapping an existing IEE's memory (fork) or loading an encrypted executable binary from the container image (exec) with the help of the *IEE-manager*. When an IEE is entered for the first time, the *IEE-manager* checks and saves its address space identifier (ASID), so that it can be identified. This helps the kernel to handle page faults that may occur in the created IEE. The *container shield* further transfers the syscall arguments between an IEE and the kernel.

5 Securing the Container

This section provides details on how the *container shield* secures containers, including protecting the file system, multi-application synchronization, inter-application communication, and user access control, defending against Iago attacks, and how the TZ-Container can be integrated with Docker.

5.1 Container Process Creation

Secure *fork* and *exec* can be used to locate a container process within an IEE. Meanwhile, the *container shield* initializes metadata for each container process, including the user ID (*uid*), group ID (*gid*), process ID (*pid*), process group ID (*pgid*), and container ID, which will be used to perform access control.

During *fork*, all these IDs are inherited except the *pid* which is allocated by the OS and checked by the *container shield*. During *exec*, all the IDs are not changed by default. However, if the executable binary contains *SUID* or *SGID* attributes, the *uid* and *gid* will be set to the IDs of the binary owner.

5.2 File System

The *container shield* enforces the integrity and confidentiality of the disk I/O using a cryptographic method. After downloading a container image, the *container shield* encrypts the image and generates a metadata file, which contains the hash values and permissions of all other files. At runtime, all read and write syscalls are intervened. For the *read*, encrypted data are read by the OS and the *container shield* decrypts the data and checks the hash value. The *write* syscall is handled similarly. To defend against

replay attacks, a hash tree of the metadata file is maintained in the secure world. The hash tree is stored in a secure storage device, e.g., Replay Protected Memory Block (RPMB). For the memory mapped I/O, the *container shield* helps the OS to load file data to the memory.

A per-container *container-key* is used to encrypt files. All the keys are stored in a *key file* and are encrypted by a root key and protected by a hash value. Both the root key and hash value are stored in a secure storage device. All the container files can be encrypted and hashed, so the property **P-2.1** is guaranteed. The *container shield* identifies the container to which an IEE process belongs and enforces the property **P-2.2**.

5.3 Multi-application Synchronization

There are multiple methods for different applications to synchronize their execution flows. After analyzing the syscalls, have identified that the OS provides three main methods for multi-application synchronization: 1) IPC semaphore; 2) signal; and 3) *flock*/*futex*.

IPC Semaphore: The OS provides two syscalls, *semget* and *semctl*, to create an IPC semaphore and initialize it. Then, *semop* is used to perform P(n) and V(n) operations on it. P(n) will wait until the semaphore value is not less than n, and V(n) will add the semaphore value with n.

The *container shield* interposes all the three syscalls and provides their functionalities instead of the kernel. It maintains a semaphore value for an IPC semaphore instance and performs P(n) and V(n) operations on this value. A spin-lock is used to protect the update atomicity. When the P(n) operation cannot obtain adequate resources, the *container shield* will mark current IEE as *WAIT* and ask the untrusted OS to schedule out current IEE. A *WAIT* IEE cannot be executed. After the V(n) operation, it will choose a *WAIT* IEE, remove the *WAIT* flag and ask the untrusted OS to schedule it.

Signal Delivery Verification: Applications can use the user-defined signal handler to synchronize the execution flow. The *container shield* checks all the signals being injected into an IEE. It generates a signal record when a signal happens, and checks whether an injected signal corresponds to a signal record when the kernel enters an IEE from a user-defined signal handler.

A legal signal is raised by either an invocation of *kill* syscall or a system event. The former is interposed by the *container shield*, which finds all target processes and checks the permission of this invocation. For the latter, we divide system events into two types: the *hardware event*, which is raised by an exception (e.g., page fault), and the *software event* (e.g., child process termination and invoking *alarm, abort* syscalls). The former can be detected by hooking all exception handlers. The latter is detected by interposing syscalls.

Secure *flock*/*futex*: Different processes can acquire an advisory lock with *flock*/*futex* syscall. Same as IPC semaphore, the *container shield* protects *flock* by maintaining a lock for the file which an IEE process acquires *flock* on, and enforces its correctness. For the *futex* syscall, the *FUTEX_WAIT* operation allows a process to wait on a lock variable, while *FUTEX_WAKEUP* wakes up the processes waiting on the variable. We allow the untrusted OS to perform these functionalities. The *container shield* checks all enter operations of the container processes and enforces that a process waiting on a variable will not be executed until another process wakes it.

The secured IPC semaphore, signal delivery verification and secured *flock*/*futex* syscall enforce all the security properties about multi-application synchronization (**P-4.1**∼**P-4.3**).

5.4 Inter-application Communication

Apart from passing data during a file transfer, many methods exist for inter-application data passing: *pipe*, *message queue* and *shared memory*. They can be divided into two types: *message passing* and *shared memory*. The *container shield* protects their integrity and confidentiality.

Message Passing: *Pipe*, *message queue* and *socket* are included in message passing. We protect them by transparently encrypting the communication data.

For a named channel, an identity token is needed. The *container shield* interposes them and identifies each channel by the token passed from an IEE. Subsequently, it asks the OS to create a communication

channel, and generates an encryption key for it. All data passed through these channels are encrypted and hashed by the *container shield*, and a nonce is used to defend against replay attacks.

An unnamed channel does not need a token and is used for processes that have the same memory view. The *container shield* generates a key for each of them and combines every key with the channel's descriptor. Both the key and descriptor propagate during *fork*.

Shared Memory: An application can create an IPC shared memory instance and map the instance to its address space using *shmget* and *shmat* syscalls. The *container shield* interposes these two syscalls, asks OS to allocate physical memory for the shared memory instance and helps the OS to map it to the IEE.

For each shared memory instance, the *container shield* records its physical memory region and a list of mapped virtual memory regions within different processes. Furthermore, it leverages the *IEE-manager* to verify all mappings of shared memory and enforce that: 1) in different processes, the virtual memory corresponding to the same shared memory instance must be mapped to the same physical memory and 2) this physical memory can only be mapped to these virtual memory regions.

By securing the two types of inter-application communication methods, the *container shield* enforces property **P-5.1** in Table 2.

5.5 User Access Control

The *container shield* performs user access control on file system access, IPC instances access and signal delivery.

File and IPC Instance Access Control: Both the file and IPC instance employ user-based access control. When a file or an IPC instance is created for the first time, the caller process needs to set its access permission. Both the owner user and owner group of the created file/instance are inherited from the process. After that, the permission can be changed by *chmod* syscall. The *container shield* hooks the creation and *chmod* syscall, and saves the permission in the metadata file for each container. Hence, the property **P6.1** is enforced.

At runtime, the *container shield* maintains each container process's UID and GID during the process creation (as mentioned in Section 5.1). It also updates these IDs by tracing and checking *setuid* and *setgid* syscalls. The standard user-based access control of Linux is performed. Each access to a file or an IPC instance is checked according to the UID and GID of the IEE. Subsequently, properties **P-6.2** and **P-6.3** are enforced.

Signal Delivery Control: Our system enforces the permission control of signal delivery during *kill* syscall. The *container shield* traces each process's UID, GID, PID, and PGID. For each *kill* syscall, it first identifies all target processes using the PID/PGID. Then, the permission check is performed based on the UID or GID of the caller and the targets: process A can send a signal to process B when 1) process A is a privileged process or 2) processes A and B have the same UID. Then, property **P-6.4** can be enforced.

5.6 Preventing Iago Attacks

The *container shield* prevents Iago attacks using existing solutions by checking the return value of the syscall [11]. The existing practical Iago attacks [17] include memory-based Iago attacks and *getpid()*-based Iago attacks. For memory maintaining syscalls (e.g., *mmap*), we enforce that the returned address cannot overlap with the existing memory regions. For *getpid*, we check whether the returned ID is the same as the traced one.

5.7 Integrating with Docker

In this section, we describe how we have integrated our system with Docker, a widely used container platform. The Docker daemon running on the server side is untrusted; however, we assume that the Docker client is running on the user's platform that is trusted. We modify the image download procedure

Table 3 Single operation overhead.

Test Case	Docker (μs)	TZ-Container (μs)
null sys call	0.21	1.85
open/close	7.37	12.2
mmap	252	404
page fault	1.24	2.53
fork+exit	1865	6712
fork+exec	3334	8875
ctxsw 2p/0k	8.82	14.1

of Docker to run the existing Docker image from the Docker Hub. Then, we change the container start-up procedure.

Pulling Docker Image: We modify the Docker daemon to invoke the *container shield* to download the image. It uses the SSL channel to protect the image downloading from the Docker Hub. Subsequently, it generates a *container key* as well as a metadata file, and encrypts the required files inside the image. Finally, the image is passed to the Docker daemon.

Starting a Container: The Docker client sends a start request, including the container image name and the execution command, to boot a container. We modify it to send this request to both the Docker daemon and the *container shield* via the SSL channel.

The Docker daemon invokes the *exec* syscall with *IEE* flag, to start the execution command in an IEE. The *container shield* verifies whether the rootfs and the execution command correspond to the user's command, and sends a message to tell Docker client that the container is started.

Communication: After starting a new container, the *container shield* exchanges a communication key with the Docker client, which is used to protect the communication between the client and its container.

6 Evaluation

We implemented a prototype of the TZ-Container on the Hikey ARMv8 development board which has eight 1.2 GHz cores and 2 GB of physical memory. We modified the Linux kernel 3.18.0 and Docker v1.10.2 to integrate them with our system. All the modules located in the secure world were implemented as runtime services of ARM Trusted Firmware (ATF) [7], so that the TZ-Container did not monopolize the usage of TrustZone. We allocated 64 MB of memory for the *IEE-manager* and *container shield* to store metadata for all IEEs. We used AES-128 to perform the file system encryption. The entire TCB (code in the secure world) was about 4,500 LoC.

During evaluation, we tied to answer the following three questions:

- *Question-1*: How does the TZ-Container influence the performance of kernel critical operations (e.g., syscalls)?
- *Question-2*: How does the TZ-Container influence the performance of real container applications?
- *Question-3*: How does the TZ-Container influence the performance of multiple containers?

6.1 Micro Benchmark

LMBench: LMBench is a series of portable micro-benchmarks for measuring individual OS operations. We used it to test the overhead of syscalls, process creation, memory manipulation and context switching. The results are shown in Table 3. The null syscall shows the overhead caused by hooking all the switches between the user and the kernel, which also switches the user page table and flushes the TLB. The overhead of pagefault is mainly caused by switching to the *IEE-manager* and the verification of the page table modification. The high overhead of *fork* and *exec* is caused by initializing the new page table, which requires frequent switches to the *IEE-manager* and verification.

http://engine.scichina.com/doi/10.1007/s11432-019-2707-6

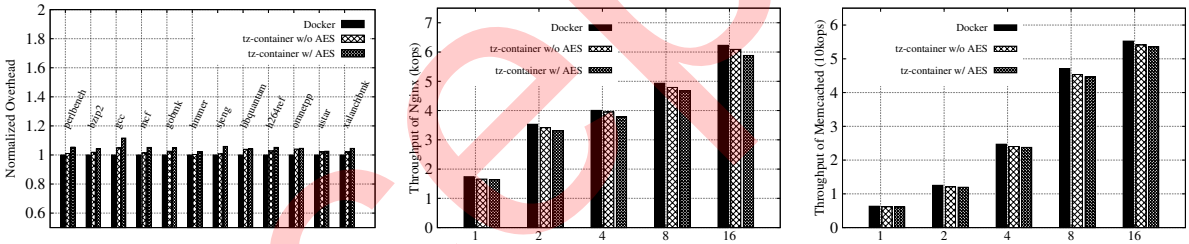
Although there is a large overhead on the single operation, it does not dramatically influence the performance of real applications. All these overheads are constant (several thousand cycles) for operations that are not frequently used, and they are small when compared with I/O operations or arithmetical operations.

SPEC_CPU 2006: We evaluated *all* SPEC_CPU 2006 INT applications under three systems: unmodified Docker (as the baseline), the TZ-Container without file system encryption and the TZ-Container with file system encryption. As shown in Figure 5(a), the average performance overhead of these applications is about 4% for the TZ-Container with file system encryption, while the gcc benchmark, which accesses the file system more frequently, has the largest overhead of 11%.

6.2 Application Overhead

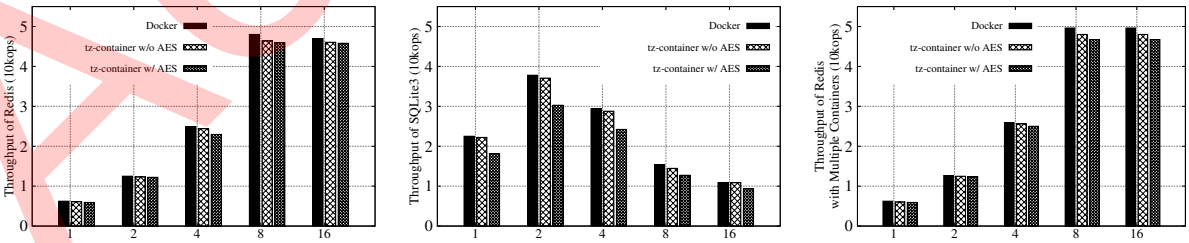
To demonstrate the performance overhead for real-world applications, we tested four different server applications: Nginx, Memcached, Redis and SQLite3. We ran these applications with different numbers of processes/threads. Furthermore, we tested multiple application instances in multiple containers. All the applications were tested in different systems: Docker (as the baseline), the TZ-Container without file system encryption, and the TZ-Container with file system encryption.

For Nginx, Memcached and Redis, we ran both the client and server on the Hikey board to eliminate the fluctuation of network. SQLite3 is a C-language library. We compiled the client together with the SQLite3 engine. The database file was stored in a temporal file system to bypass the disk overhead.



(a) SPEC_CPU INT **(b) Nginx** **(c) Memcached**

Figure 5 Figure (a) shows the overhead of all INT applications in SPEC_CPU 2006 benchmark, lower the better. Figures (b) and (c) show the throughput of Nginx and Memcached, higher the better. The **x-axis** of figures (b) and (c) represents the number of processes/threads used by the application.



(a) Redis **(b) SQLite3** **(c) Redis-Container**

Figure 6 Figure (a) and (b) shows the throughput of Redis and SQLite3. Figure (c) shows the throughput of Redis with different numbers of containers. The **x-axis** represents the number of processes/threads/containers used by the applications. Higher the better.

Nginx: We configured the Nginx server to use 1-16 worker processes. A client was used to send requests to the server. The thread number of the client was the same as the process number of the server. As shown in Figure 5(b), the overhead is approximately 3% and 6% for the TZ-Container without and with file system encryption, respectively. Because the Nginx server rarely accesses the file, file system encryption causes very limited overhead.

Memcached is an in-memory database. We configured Memcached to use at most 512 MB of memory to store the database. We ran Memcached with different server threads and used a multi-thread client to send requests to the server. The number of threads used by the server and client was the same. The test workload comprised 50% set operations and 50% get operations. Figure 5(c) shows the throughput of Memcached, and the overhead of the TZ-Container is less than 5%.

Redis is an in-memory database that can leverage the disk to provide persistence. We configured it to synchronize data into the disk every 10 seconds. Since Redis is a single-thread application, we started multiple Redis instances (1-16) listening on different ports. A multi-thread client (the number of threads was equal to that of the server) sent requests to the server. The workload comprised 50% set operations and 50% get operations. Figure 6(a) shows the throughput of the Redis server, the overhead of our system is less than 6%.

SQLite3 is an on-disk SQL database engine. We used a client, compiled together with the SQLite3 engine, to insert values into the database. The workload comprised 100% insert operations. The client ran with different numbers of threads (1 to 16), and used the Linux temporal file system to store the database file. Figure 6(b) shows the throughput of SQLite3; the overhead of the TZ-Container without file system encryption is about 4%. We used the temporal file system to eliminate the fluctuation of the disk, which provided a higher throughput than the real disk. For that reason, the AES encryption/decryption of each file system access caused an average performance overhead of 18%.

Multi-container: We ran Redis servers in different containers and each of them held a Redis server. We used the same workload as that used in the single-container test. Figure 6(c) shows the throughput, the overhead of the TZ-Container is less than 7%.

7 Security Analysis

The TZ-Container defends against direct attacks, Iago attacks and MUMA attacks. In this section, we first analyze attacks on containers and the attacks directly on our system components, namely, the *IEE-manager* and *container shield*. After that, we discuss the limitation of the TZ-Container.

7.1 Attacking Containers

This paper divides all the attacks from an untrusted OS to a container into single application attacks (direct attacks and Iago attacks) and MUMA attacks. To protect container applications against direct attacks, the TZ-Container constructs multiple **IEE**s with ARM TrustZone technology. To protect against Iago attacks, we borrow ideas from existing works to create the defence mechanism of the TZ-Container. MUMA attacks are new attacks introduced in the container scenario, and they have not been studied well in previous works. Leverage the *container shield*, the TZ-Container defends against the three types of MUMA attacks.

One way for an untrusted kernel to perform MUMA attacks is by tampering with the control data of a process, which is maintained in the kernel space (e.g., kernel stack, kernel heap objects and opened file handlers). Although the kernel is allowed to modify these data, the TZ-Container checks all user-kernel interactions (e.g., all syscalls) and defends against malicious behaviors from the kernel. For example, no matter how the kernel tampers with the opened file handlers, the *container shield* could protect the file system functionalities used by the container applications.

7.2 Attacking TZ-Container

In this section, we analyze how the TZ-Container protects itself.

Hacking System Code Integrity: During system booting, an attacker may try to modify the code of the kernel or even the codes of the *IEE-manager* and the *container shield*. Secure boot technology is used to ensure their integrity during system boot. After booting, the *IEE-manager* ensures that the kernel code is write-protected.

Code-reuse Attacks: An attacker may try to reuse the code of the kernel or let the kernel jump to the user space memory region to execute critical instructions (e.g., page table maintaining instruction) and bypass the *IEE-manager*. The TZ-Container ensures that there is no ROP gadget that can be used to construct critical instructions (e.g., switching the page table) under all ARM ISAs (which is relatively easy on ARM platform because instruction alignment is required). Meanwhile, the *IEE-manager* ensures that the user's memory is mapped as Privileged eXecute Never, thereby preventing return-to-user attacks.

DMA Attacks: An attacker may leverage direct memory access (DMA) to access the container process' memory or inject code into the kernel memory. The TZ-Container defends against these attacks by controlling the system memory management unit (SMMU), which performs address translation for DMA. SMMU is controlled by certain memory-mapped registers, and the *IEE-manager* will enforce that these regions are only mapped in its own address space. After exclusively controlling the SMMU, the *IEE-manager* can forbid DMA access to the container process' memory or the kernel's code section.

7.3 Security Limitation

The TZ-Container cannot defend against side-channel attacks [27, 47], DoS attacks and physical attacks. It also does not consider that an application itself leaks its data. For new covert-channel attacks which is based on reorder or speculative execution, such as Meltdown [33], Spectre [30] and Foreshadow-NG [46], TZ-Container suppose that they should be solved by existing defense method. For the MUMA attacks, currently, TZ-Container solves three kinds of them, which are introduced in the paper.

8 Related Work

Protecting applications and their data from untrusted privileged software is a long-standing research objective. In this section, we discuss both the software-based and the hardware-based systems used to protect applications.

Software-based solutions: In the first place, the initial work such as Proxos [43] and NGSCB [37] executes one small trusted OS together with the original untrusted OS using virtualization, and the security-sensitive applications will be located in the trusted OS. Different with Proxos and NGSCB, the following work, including Overshadow [19], SP^3 [48], SICE [13], Fides [40], InkTag [28] and Virtual Ghost [22], directly executes the application on the untrusted OS and try to protect their memory from being accessed by OS. TrustVisor [34] leverages the system management mode (SMM) to protect the execution of a piece of code. Sego [31] extends these methods by protecting data handling with trusted metadata. MiniBox [32] leverages a hypervisor to implement a two-way sandbox and provides the isolation between the native application and the guest OS. Nested Kernel [24] provides an intra-kernel privilege separation method, this technology is also borrowed by us to protect the memory mapping. Unlike the TZ-Container, these systems focus on protecting single application or a piece of code. They cannot secure a container and provide a defence against MUMA attacks. gVisor [9] protects container applications by assigning a secure libOS called Sentry for each container. Dan et al [23] ran unikernel as a process, which can also be used to isolate a container application. However, both of them do not target on protecting containers from untrusted host kernel. JointCloud computing [16,39,45] protects user services by locating them in different clouds, but cannot defend against malicious cloud provider.

Hardware-based solutions: There exist many trusted hardware, which can protect security-sensitive applications, with different performance and security functionalities. ARM TrustZone [10] extension secures the application by providing an isolated execution environment called secure world. Many existing systems leverage ARM TrustZone to enforce system security [12, 21, 29, 41]. TZ-RKP [12] protects the kernel by hacking all memory mapping modifications. OSP [21] and TrustICE [41] use TrustZone and virtualization to securely execute multiple pieces of code in normal world. vTZ [29] leverages TrustZone and virtualization to securely construct multiple virtual secure worlds. These systems can neither provide multiple secure environments to protect native applications nor defend against MUMA attacks. SANCTUARY [15] leverages TrustZone to construct multiple enclaves in normal world. However, it requires modifications to the hardware and an enclave will monopolize a core. It also does not consider

the MUMA attacks. Intel SGX [3] provides multiple trusted execution environments called enclaves on the X86 platform. Haven [14] ports a libOS to run into enclave. SCONE [11] protects Linux containers by running the user-level part in an enclave. However, it can only execute one process in each container, and cannot support *fork* and *exec* syscalls. Graphene-SGX [44] also leverages SGX to protect different applications. It can protect the communication between parent and child processes. Still, none of them targets on securing the OS services for the containers with multiple users and multiple processes.

9 Conclusion

In this paper, we focused on the problem of protecting applications within containers and highlight the presence of new attacks called MUMA attacks. Furthermore, we present the TZ-Container, a system that can protect containers against an untrusted OS with ARM TrustZone. The TZ-Container constructs multiple isolated execution environments (IEEs) to locate different container processes. Based on the IEE, the TZ-Container checks OS services by hooking syscalls and defends all presented attacks including MUMA attacks. The TZ-Container is integrated with Docker and can directly run unmodified Docker images. We implemented the TZ-Container on the LeMaker Hikey ARMv8 development board. The evaluation results demonstrate that our system has a performance overhead of approximately 5% for common server applications.

Acknowledgements This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000104), the National Natural Science Foundation of China (No. 61772335) and the Program of Shanghai Academic Research Leader.

References

1 Amd launching "hierofalcon" 64bit arm embedded chips in 1h 2015 - zen and k12 next year. http://wccftech.com/amd-launching-arm-serves-year-wip/#ixzz3Yef58mtq, 2015.
2 Docker on arm. https://github.com/umiddelb/armhf/wiki/Installing,-running,-using-docker-on-armhf-(ARMv7)-devices, 2015.
3 Software guard extensions programming reference. https://software.intel.com/site/default/files/329298-001.pdf, 2015.
4 Amd opteron a1100. http://www.amd.com/en-gb/products/server/opteron-a-series, 2016.
5 Kubernetes on arm. http://kubecloud.io/kubernetes-on-arm-cluster/, 2016.
6 Linux cve. http://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/Linux-Linux-Kernel.html, 2016.
7 Arm trusted firmware. https://github.com/ARM-software/arm-trusted-firmware, 2017.
8 Rancher-labs. http://rancher.com/rancher-labs-2017-predictions-rapid-adoption-and-innovation-to-come/, 2017.
9 gvisor. https://github.com/google/gvisor, 2018.
10 T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.
11 S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016.
12 A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. 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*, pages 90–102. ACM, 2014.
13 A. M. Azab, P. Ning, and X. Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388. ACM, 2011.
14 A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.
15 F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. Sanctuary: Arming trustzone with user-space enclaves. 2019.
16 D.-G. Cao, B. An, P.-C. Shi, and H.-M. Wang. Providing virtual cloud for special purposes on demand in jointcloud computing environment. *Journal of Computer Science and Technology*, 32(2):211–218, 2017.
17 S. Checkoway and H. Shacham. *Iago attacks: Why the system call api is a bad untrusted rpc interface*, volume 41. ACM, 2013.
18 H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P.-c. Yew, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. *Parallel Processing Institute Technical Report*, (FDUPPITR-2007-08001), 2007.
19 X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. ASPLOS*, pages 2–13. ACM, 2008.
20 S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: A Hardware-Software Approach to Full System Security. In *ICS*, 2011.

21 Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 565–578. USENIX Association, 2016.

22 J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News*, 42(1):81–96, 2014.

23 W. Dan, L. Martin, K. Ricardo, and P. Nikhil. Unikernels as processes. In *2018 ACM Symposium on Cloud Computing*, 2018.

24 N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206. ACM, 2015.

25 S. Graber and S. Hallyn. Lxc linux containers, 2014.

26 L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. *arXiv preprint arXiv:1704.05600*, 2017.

27 M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 299–312, 2017.

28 O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. *ACM SIGPLAN Notices*, 48(4):265–278, 2013.

29 Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing arm trustzone. 2017.

30 P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

31 Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel. 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*, pages 277–290. ACM, 2016.

32 Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 409–420, 2014.

33 M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

34 J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

35 D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

36 T. P. Morgan. Arm servers: Cavium is a contender with thunderx. https://www.nextplatform.com/2015/12/09/arm-servers-cavium-is-a-contender-with-thunderx/, 2015.

37 M. Peinado, Y. Chen, P. England, and J. Manferdelli. Ngscb: A trusted open system. In *Australasian Conference on Information Security and Privacy*, pages 86–97. Springer, 2004.

38 J. Rath. Baidu deploys marvell arm-based cloud server. http://www.datacenterknowledge.com/archives/2013/02/28/baidu-deploys-marvell-arm-based-server/, 2013.

39 P. SHI, H. WANG, Z. ZHENG, and H. YIN. Collaboration environment for jointcloud computing. *SCIENTIA SINICA Informationis*, 47(9):1129–1148, 2017.

40 R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 2–13. ACM, 2012.

41 H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 367–378. IEEE, 2015.

42 Y. Sverdlik. Paypal deploys arm servers in data centers. http://www.datacenterknowledge.com/archives/2015/04/29/paypal-deploys-arm-servers-in-data-centers, 2015.

43 R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 279–292. USENIX Association, 2006.

44 C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, page 8, 2017.

45 H. Wang, P. Shi, and Y. Zhang. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, pages 1846–1855. IEEE, 2017.

46 O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, Technical report, 2018.

47 Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

48 J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80. ACM, 2008.