RContainer: A Secure Container Architecture through Extending ARM CCA Hardware Primitives

Qihang Zhou*, Wenzhuo Cao*†, Xiaoqi Jia *† ✉, Peng Liu‡, Shengzhi Zhang§,
Jiayun Chen*†, Shaowen Xu*† and Zhenyu Song*
*Institute of Information Engineering, Chinese Academy of Sciences, CHINA
Email: {zhouqihang, caowenzhuo, jiaxiaoqi, chenjiayun, xushaowen, songzhenyu}@iie.ac.cn
†School of Cyber Security, University of Chinese Academy of Sciences, CHINA
‡The Pennsylvania State University, USA, Email: pliu@ist.psu.edu
§Metropolitan College, Boston University, USA, Email: Shengzhi@bu.edu

***Abstract*—Containers have become widely adopted in cloud platforms due to their efficient deployment and high resource utilization. However, their weak isolation has always posed a significant security concern. In this paper, we propose RContainer, a novel secure container architecture that protects containers from untrusted operating systems and enforces strong isolation among containers by extending ARM Confidential Computing Architecture (CCA) hardware primitives. RContainer introduces a small, trusted mini-OS that runs alongside the deprivileged OS, responsible for monitoring the control flow between the operating system and containers. Additionally, RContainer uses shim-style isolation, creating an isolated physical address space called conshim for each container at the kernel layer through the Granule Protection Check mechanism. We have implemented RContainer on ARMv9-A Fixed Virtual Platform and ARMv8 hardware SoC for security analysis and performance evaluation. Experimental results demonstrate that RContainer can significantly enhance container security with a modest performance overhead and a minimal Trusted Computing Base (TCB).**

I. Introduction

Containers are widely used in cloud-native environments to provide a lightweight solution for packaging, distributing, and running applications on shared cloud infrastructure. As ARM-powered machines gain popularity in the cloud server market, companies are exploring the use of containers on ARM server platforms. However, the security of containers, compared to traditional virtual machines (VMs) is questioned due to inadequate isolation caused by the shared host Operating System (OS) kernel. The shared OS kernel, often Linux, contains numerous vulnerabilities that can be exploited by compromised containers, risking disastrous security breaches (e.g., data breaches, data manipulation) to all other containers. Even worse, the problem is beyond merely protecting a container against a compromised host OS. It has been widely recognized that the real problem is how to simultaneously achieve the objective of strong isolation, the objective of being lightweight, and the objective of having minimum amount of trusted code running with highest privileges.

Without taking the objective of being lightweight into consideration, the objective of strong isolation can be achieved by a hypervisor. These approaches enhance container isolation by incorporating additional security mechanisms into the hypervisor [1], [2], [3] or running the container within an independent lightweight-VM [4], [5]. However, none of the existing hypervisor-based VM security solutions achieve the objective of being lightweight. Furthermore, these hypervisor-based solutions often expand the system's Trusted Computing Base (TCB) due to the integration of virtualization layers, thereby increasing the attack surface. In order to simultaneously achieve the objective of strong isolation and the objective of being lightweight, researchers have been exploring Trusted Execution Environment (TEE)-based mechanisms. Without suffering from heavyweight virtualization caused by a hypervisor, security-oriented hardware features of processor chips, such as ARM TrustZone [6], can create two distinct Physical Address Spaces (PAS), resulting in physical isolation which is adequately strong for protecting an application from a malicious OS. TZ-container [7] and TrustShadow [8] enhance the security of containers (or applications) by adding a shield or runtime system in TrustZone. However, such solutions may suffer from at least two drawbacks. First, all containers or trusted applications (TAs) running in the TrustZone have the same permissions, enabling malicious containers to attack other legitimate ones. Second, the increasing number of TAs as well as containers and the growing complexity of trusted OS (TOS) expands the attack surface of commercial TrustZone-based systems [9].

Recently, ARM introduced Confidential Computing Architecture (CCA) in ARMv9-A, which includes a new PAS for confidential VMs (Here, each VM runs one OS and several containers/applications). Although ARM CCA has great potential, the existing CCA-based designs still fall short of simultaneously achieving the aforementioned three objectives (i.e., strong isolation, lightweight, tiny code running with highest privileges). For example, Shelter [10] attempts to extend ARM CCA to isolate one application from another (i.e., protect userspace isolation): they introduce the needed security functionalities into EL3. However, although the performance overhead of Shelter is significantly reduced compared to hypervisor-based VMs, the amount of code running with

✉Xiaoqi Jia is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240328
www.ndss-symposium.org

highest privileges (note that EL3 is the highest exception level in the ARM architecture) is unfortunately still substantial (e.g., the Shelter security monitor has 2K lines of code running with EL3 privileges). The Shelter's monitor possesses higher privileges than any software in Normal World, Realm World, or Secure World. Any issues occurring within EL3 could have a disastrous impact on the entire platform. Therefore, it is advisable to minimize the amount of new code introduced in EL3 as much as possible.

To simultaneously achieve the three aforementioned objectives, we introduce RContainer, a novel secure container architecture based on the Realm Management Extension (RME) hardware primitives of ARM CCA, to enable comprehensive container protection against an untrusted OS while enforcing strong isolation between containers. RContainer separates critical resources from the host OS and introduces a trusted mini-OS in Normal World EL1. This mini-OS, which logically has higher privileges, is responsible for managing interactions between containers and the deprivileged OS. The mini-OS employs a novel mixed-pagetable approach to isolate itself from the deprivileged OS, providing security capabilities like memory isolation and control flow protection to safeguard the containers. Leveraging mixed-pagetable isolation, RContainer achieves strong isolation between the mini-OS and deprivileged OS and makes it possible to securely move the majority of the container isolation functionalities, which would usually be running within EL3, into EL1. Furthermore, RContainer adopts a shim-style isolation approach, which not only isolates container userspace but also isolates kernel space by instantiating the container-related data plane and monitoring the control plane (through mini-OS) in the kernel. RContainer establishes an isolated domain in EL1, termed con-shim, for each container to isolate and limit the scope of different containers in the kernel space. Isolation between different con-shims (and their containers) is achieved by assigning a separate Granule Protection Table (GPT) called Shim-GPT. In each container's Shim-GPT, RContainer restricts container access by eliminating the current container's access to extraneous memory, including the mini-OS, deprivileged OS, and other containers. Through these methods, RContainer achieves lightweight bidirectional isolation that safeguards the container from the untrusted host OS while simultaneously strengthening the isolation between multiple containers and minimizing the amount of newly added code running within EL3.

We have implemented two prototypes of RContainer on ARM Fixed Virtual Platform (FVP) [11] that support ARMv9-A architecture and ARMv8 SoC development board [12], respectively, for security and performance evaluation. RContainer introduces a TCB of 2.6K Source Lines of Code (SLoC) without encryption (only 130 SLoC in EL3), and an additional 2.3K SLoC when encryption is enabled. We analyzed 30 vulnerabilities related to containers and Linux, including privilege escalation, information leakage, memory corruption, and denial of service, and found that the important prerequisite for exploiting these vulnerabilities is a system design with weak memory isolation and unrestricted permissions. Our evaluation results show that RContainer can defeat all the aforementioned exploits with modest performance overhead.

Contributions. Our contributions are summarized as below:

- We designed RContainer, a novel secure container architecture via ARM CCA hardware primitives, which protects containers on untrusted OS while enforcing strong isolation among containers both in userspace and kernel space with minimal TCB.
- We proposed a novel mixed-pagetable approach that isolates different components running at the same exception level through extending RME, and moved many security functionalities to the EL1 to balance security, performance, and the amount of code running with highest privileges.
- We implemented two prototypes of RContainer on ARMv9-A FVP and ARMv8 hardware SoC, respectively, and evaluated Rcontainer's security features and performance overhead.

II. Preliminary

A. Security Insights

The security considerations of RContainer come from the following three insights.

Security Insight 1: Isolation between containers in both userspace and kernel space. It is crucial to have isolation in not only userspace, but also kernel space between containers. Different containers often have complex shared dependencies on kernel data structures (e.g., abstract resources [13]). This leads to the possibility of more widespread attacks between containers by exploiting kernel space data resources.

Security Insight 2: Minimizing the highest-privilege code. Running less code at a higher privilege level is a fundamental principle for enhancing system security and safety. In modern systems, code running at the highest privilege has extensive access and control over system resources (e.g., memory, CPU). Consequently, any issues that occur may have a catastrophic impact on the whole system.

Security Insight 3: Scalable security features. Due to the ever-evolving nature of attack methods, the design of security systems should be scalable when new attack methods keep on emerging, while avoiding violations of Security Insight 2 as much as possible.

B. Introduction of ARM Confidential Computing Architecture

ARM Confidential Computing Architecture (CCA) [14] is a novel security technique featured in ARMv9-A. It is designed to protect data and code from leaking or being tampered with during runtime. ARM CCA introduces a new trusted isolation environment known as the Realm World, which allows applications to execute within a secure memory space, isolated from the regular Normal World and Secure World. The Realm World is more versatile than traditional confidential VMs, supporting both the confidential VM and OS. Furthermore, ARM CCA introduces a novel memory protection mechanism called Granule Protection Check (GPC), which independently checks the access permission of physical memory, separate from the Memory Management Unit (MMU). ARM CCA segregates all Physical Address Spaces (PAS) into four distinct states: Normal PAS, Secure PAS, Realm PAS, and Root PAS, corresponding to the memory in the four respective worlds. The Normal World can only access the Normal PAS, while

the Secure World and Realm World can access both their own PAS and the Normal PAS. The Root World, however, can access all PAS. Furthermore, ARM CCA maintains a table called the Granule Protection Table (GPT) that manages the security state of physical addresses in coordination with GPC. Operations related to GPT can only be performed by EL3, including loading GPT to `GPTBR_EL3` and updating GPT. Through the above mechanisms, ARM CCA has implemented fine-grained strong isolation in a more flexible manner.

C. Challenges

Challenge 1 (C1): Containers are not really suitable for deployment in Realm World. ARM CCA offers a new confidential computing environment, known as the Realm VM, to protect data and code running in it from any privileged software or firmware. An intuitive idea is to directly use the Realm VM to protect containers, either placing multiple containers in one Realm VM or using a separate Realm VM to run each container. However, both approaches are problematic. Since multiple containers still share a single Realm OS kernel, the first approach is unlikely to provide adequate isolation between containers, thus violating **Security Insight 1**. The second approach mirrors traditional hypervisor-based solutions (e.g., microVMs); accordingly, it encounters the same performance issue faced by existing hypervisor-based TEEs. Moreover, both approaches introduce a relatively large TCB, which includes both OS and RMM in Realm World.

Challenge 2 (C2): It is very challenging to achieve tamper-proof protection of the TCB when only a small portion of the TCB is running with the highest privilege (i.e., EL3). Based on **Security Insight 2** and **Security Insight 3**, only a small portion of the TCB is running at the highest privilege level, but how to ensure that the remaining parts of the TCB are tamper-proof? An intuitive idea is to deploy the TCB in Secure World or Realm World. However, this approach has two limitations. First, the TCB complexity has to increase. Taking Secure World as an example, the TCB will either run in secure monitor (before ARMv8, secure monitor and Secure World belonged to the same isolated domain) or be integrated with TOS. In either way, the TCB complexity will increase. Second, this approach may hurt compatibility. The three Worlds in ARM architecture are designed to be in parallel and have distinct software stacks, running different programs and services. Since the containers are running in the Normal World, this approach will inevitably introduce many cross-world operations.

Due to these limitations, it is preferable to deploy the TCB within Normal World. However, this strategy will result in the TCB and the host OS running at the same exception/privilege level, raising the problem of providing the TCB with tamper-proof protection. Generally speaking, there are two straightforward solutions to isolate different environments at the same exception level: (1) create separate pagetables for the TCB and the host OS, and hook pagetable operations in EL3; (2) use the same pagetable but set pagetable pages as read-only [15], [16], [17], [18] in Normal World, and manipulate every update of the pagetable in EL3. Unfortunately, both solutions introduce a significant performance overhead due to frequent pagetable and context switching (frequent refreshing of TLB leads to slower memory access).

TABLE I: Threat Model Matrix.

Malicious components	Steal/tamper with container	Steal/tamper with OS	DoS on container	DoS on OS
Self container	○	●	○	◑
Other container	●	●	◑	◑
Con-shim	○	●	○	◑
Depriv OS	●	○	◑	○

●indicates fully-considered; ○indicates not-considered; ◑indicates partially-considered (RContainer considers memory-related DoS [13], [20], e.g., memory consumption, abstract resource attacks, and does not consider others).

D. Threat Model

We assume that attackers can exploit vulnerabilities to take control of the system software in Secure World, Realm World, and Normal World. Then, they can steal or tamper with the private memory and register state of containers by hijacking the control flow, remapping memory, controlling DMA-capable devices, and launching Iago [19] or MUMA (Multi-User Multi-Application) [7] attacks. Moreover, attackers could deliberately deploy a malicious container on our platform and attempt to attack the kernel and other containers through container escalation and resource abuse. Table I presents the threat model matrix of RContainer, where ● indicates a fully-considered threat, ○ indicates a not-considered threat, and ◑ indicates a partially-considered threat.

A container deliberately leaking its sensitive data, side-channel attacks (e.g., [21], [22]), cryptography attacks, Denial-of-Service (DoS) attacks, and physical attacks are out of our scope. Although DoS attacks are also out of our consideration, RContainer can defend against memory-related DoS from malicious containers (e.g., abstract resource attacks [13], memory consumption). We also do not consider the vulnerabilities of applications within a container. Our approach can be deployed alongside existing side-channel attack defense solutions, e.g., [23], [24]. We assume that the system is benign during the boot-up [25], which allows for secure storage of keys and signatures. However, the system may be compromised during runtime. We also assume that the security of container images can be maintained through encryption [26], and the existing attestation (e.g., [27], [28]) can be applied to our system.

III. System Design

A. Architecture

To address ***challenge C1***, we introduce a novel secure container architecture called RContainer within the Normal World. RContainer explores and acknowledges the inherent connections between userspace isolation and kernel space isolation. Fig. 1 illustrates the architecture of RContainer, which comprises a mini-OS in EL1 and a secure monitor (native firmware) in EL3 as the TCB, and multiple con-shims for container isolation. The mini-OS is a compact and basic OS that provides memory management for container/con-shim memory and control flow protection. Meanwhile, service provisioning is still retained in the deprivileged OS. The mini-OS and deprivileged OS coexist at the same exception level but are isolated through a mixed-pagetable mechanism (Section III-B). Each con-shim represents a lightweight isolated instance allocated to each container in kernel space.

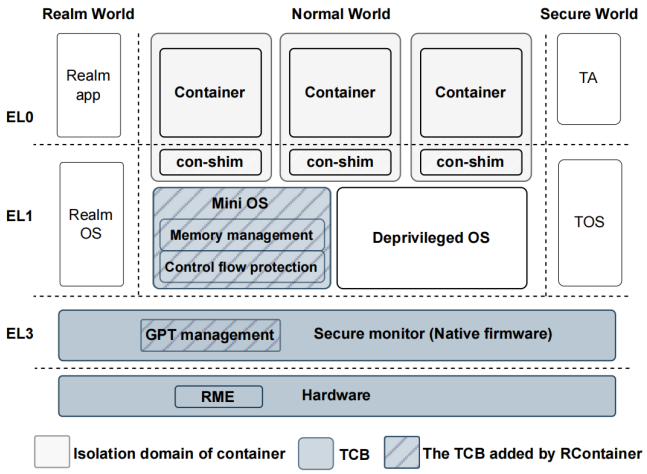
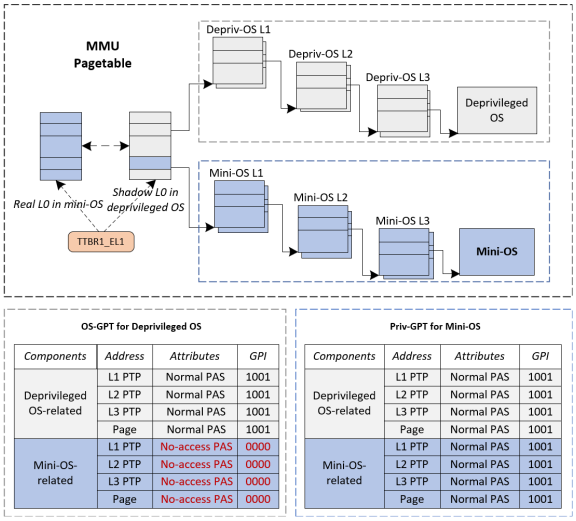


Fig. 1: RContainer Architecture.



OS-GPT for Deprivileged OS

Components	Address	Attributes	GPI
Deprivileged OS-related	L1 PTP	Normal PAS	1001
	L2 PTP	Normal PAS	1001
	L3 PTP	Normal PAS	1001
	Page	Normal PAS	1001
Mini-OS-related	L1 PTP	No-access PAS	0000
	L2 PTP	No-access PAS	0000
	L3 PTP	No-access PAS	0000
	Page	No-access PAS	0000

Priv-GPT for Mini-OS

Components	Address	Attributes	GPI
Deprivileged OS-related	L1 PTP	Normal PAS	1001
	L2 PTP	Normal PAS	1001
	L3 PTP	Normal PAS	1001
	Page	Normal PAS	1001
Mini-OS-related	L1 PTP	Normal PAS	1001
	L2 PTP	Normal PAS	1001
	L3 PTP	Normal PAS	1001
	Page	Normal PAS	1001

Fig. 2: Mixed-pagetable.

By combining con-shims, RContainer introduces a shim-style isolation paradigm (Section III-C).

B. Mini-OS

In order to address the ***challenge C2***, first, instead of running the TCB in EL3, RContainer introduces a minimal trusted mini-OS that runs in EL1 alongside the deprivileged OS. Second, the isolation between the mini-OS and the deprivileged OS is guaranteed by a mixed-pagetable protection mechanism. Regarding why tamper-proof protection of the mini-OS is achieved, it should be noted that the deprivileged OS and the mini-OS utilize the same Memory Management Unit (MMU) pagetable but different GPTs. We refer to this approach as the **mixed-pagetable**.

Specifically, the pagetable shared by the mini-OS and the deprivileged OS is maintained in the deprivileged OS, but the pagetable pages related to the mini-OS are masked and isolated from the deprivileged OS through the GPC mechanism. Two EL1-related GPTs, called Priv-GPT and OS-GPT, respectively, are maintained in the EL3 secure monitor. They record the memory security attributes of the mini-OS and the deprivileged OS, including runtime memory and the pagetable pages. Pagetable pages in EL1 possess different security attributes in the two GPTs. The Priv-GPT is the GPT of the mini-OS and has full access permission. The OS-GPT is the GPT of the deprivileged OS and can only access its own memory and pagetable pages. As depicted in Fig. 2, the pagetable pages (abbreviated as PTP in the figure) that maintain the mini-OS mapping are marked as No-access PAS in the OS-GPT. This ensures that when the deprivileged OS is running, it cannot access the mini-OS related pagetable pages. The pagetable pages that maintain the deprivileged OS mapping are marked as Normal PAS in the OS-GPT, ensuring that the deprivileged OS can update its own pagetable pages normally. Any pagetable page (except L0) can either store mini-OS-related or deprivileged OS-related mappings, but not both. It's worth noting that L0 is the only pagetable page that maps both the mini-OS and deprivileged OS, so it needs to be protected separately. We use shadow L0 to prevent deprivileged OS from tampering with L0 page. Through the mixed-pagetable isolation, RContainer places most of its security capabilities in EL1 of Normal World in a lightweight manner.

The security capabilities of mini-OS mainly include two aspects: *memory management* and *control flow protection*.

Memory Management. The mini-OS provides lightweight memory management for memory protection, which includes two security mechanisms. The first mechanism is the maintenance of GPTs at the software level. The mini-OS manages a GPT-routing table to maintain the mapping relationship between each container and its own GPT. When a container is created, the mini-OS notifies the secure monitor in EL3 to assign a specific GPT to the container and records the new GPT index in the GPT-routing table. Before the container runs, the mini-OS first locates the GPT index corresponding to the current container in the GPT-routing table and then notifies the secure monitor in EL3 to switch the GPT. The second mechanism is fast memory allocation. The mini-OS has a lightweight allocator for allocating the container memory and handling pagefaults. Furthermore, to improve the performance of allocation, the mini-OS maintains some continuous physical memory areas [29] for fast memory allocation. These physical memory areas are set to No-access PAS in the OS-GPT. The specific container memory management is described in detail in Section *IV-C*.

Control Flow Protection. The mini-OS provides control flow protection through an exception interposer. The exception interposer is utilized for exception-level switching protection. Specifically, when a container executes a system call, the control flow initially switches from the container to the mini-OS. The mini-OS then copies the parameters of the system call based on the requested information, clears the sensitive CPU registers, and finally switches to the deprivileged OS for processing. Once the processing is completed, the control flow returns to the mini-OS. Here, the mini-OS checks the processing results of the system call and, after verification, returns to the container. In addition to system calls, exceptions, and interrupts can also directly trigger exception-level switching between the container and the deprivileged OS, which may

lead to control flow hijacking. To prevent such attacks, when an exception or interrupt occurs, RContainer forces the control flow to first enter the EL3 monitor for GPT switching (from Shim-GPT/OS-GPT to Priv-GPT) and then transfers the control flow to the mini-OS. The mini-OS distributes the exception or interrupts to the specific handler in the deprivileged OS for processing with EL3 GPT switching (from Priv-GPT to OS-GPT). The mini-OS maintains a copy of the exception vector table, which is protected from modification by the deprivileged OS. After each exception handling, the mini-OS checks the results and returns to the container. All entrypoints of handlers are hard-coded.

C. Shim-style Isolation

Due to the complex dependencies, isolation is essential not only in userspace but also in kernel space. Malicious containers can exploit kernel vulnerabilities or shared resources (e.g., global variables [13]), endangering other containers and the kernel. Some works use dedicated kernels for individual containers (e.g.,[5], [30]), but this may sacrifice lightweight advantages and introduce TCB complexity. In RContainer, we adopt a shim-style isolation paradigm for kernel space, guided by two key observations: (1) While most container-related attacks originate in the control plane (i.e.,hijacking control flow, defective code, etc), they ultimately impact the data plane (i.e., global variables, kernel data structures, etc). (2) The data plane is accessed more frequently and requires stronger isolation for containers.

Building upon the first observation, the shim-style isolation approach involves two core concepts: First, containers are instantiated within the kernel's data plane. Second, Containers are monitored (by mini-OS) from the control plane of the kernel. Specifically, in the data plane, RContainer identifies three critical types of kernel data that directly influence container security: (1) Kernel boundary points: These serve as switching points between container userspace and kernel space, such as system call entry/exit points; (2) Container-specific private data structures: These unique structures store container context information (e.g., `task_struct`) and are specific to each container; (3) Shared global variables: These variables are allocated to multiple containers simultaneously and can be controlled by containers (e.g., `nr_files`). To instantiate the data plane, RContainer establishes an isolated environment known as 'con-shim' (short for container shim) for each container at EL1. Each con-shim represents a lightweight isolation domain within kernel space, encompassing the aforementioned data types, shared memory areas, and its own stacks.

Based on our second observation, each con-shim should be exclusively associated with its own container and should not be trusted by other containers. Therefore, all con-shims are out of our TCB. To restrict memory access [31], RContainer maintains a separate GPT (Shim-GPT) for each con-shim/container at EL3. Table II illustrates the memory access attributes within each Shim-GPT. Each con-shim/container is limited to accessing only its own memory, preventing access to others' memory (including other con-shims, containers, deprivileged OS, mini-OS, and secure monitor). As a result, each container's direct impact on the kernel data plane is confined to its respective con-shim. For the control plane, the bidirectional control flow between the container and the deprivileged OS is monitored by mini-OS (as described in Section III-B).

TABLE II: Memory Access Attributes in Each Shim-GPT.

Object	Permission	GPI encode in GPT
Self container	Access permitted in Shim-GPT	1001 (Normal PAS)
Self con-shim	Access permitted in Shim-GPT	1001 (Normal PAS)
Other container	No access in Shim-GPT	0000 (No-access PAS)
Other con-shim	No access in Shim-GPT	0000 (No-access PAS)
mini-OS	No access in Shim-GPT	0000 (No-access PAS)
Deprivileged OS	No access in Shim-GPT	0000 (No-access PAS)
Secure monitor	No access in Shim-GPT	1010 (Root PAS)

IV. Container Lifecycle Protection

A. Boot Integrity

1) System bootup: Secure Boot [25] is a common system secure bootup method that uses chain-verification of electronic signatures to verify the reliability of crucial images in the system, subsequently loading and running these verified images. In the ARM architecture, the secure monitor in EL3 serves as the security firmware responsible for booting the Bootloader, Linux kernel, Trusted OS, RMM, and other important images. In RContainer, the mini-OS and deprivileged OS are linked into one binary file and located in a unified address space. To reduce complexity, we employ a lazy loading approach for secure bootup. Specifically, at boot time, the deprivileged OS is loaded and measured by the secure monitor in EL3 and continues booting up itself as usual until it starts launching the mini-OS. The deprivileged OS in the early stage of system bootup is considered secure, and at this time, network, serial ports, and other services have not yet been initialized, so attackers are unable to leverage any remote attacks. When the mini-OS is launched, it first notifies the secure monitor in EL3 to initialize the OS-GPT (for deprivileged OS) and Priv-GPT (for mini-OS) and sets the physical memory areas of the mini-OS to No-access PAS in the OS-GPT. This ensures that when the mini-OS is started, the deprivileged OS will no longer be able to access the mini-OS's memory. After the initialization of the mini-OS is completed, the mini-OS notifies the secure monitor to set the current GPT to OS-GPT and transfers control to the deprivileged OS for subsequent tasks.

2) Container instance initialization: The entire process of container initialization comprises two steps: con-shim initialization and container initialization. When the deprivileged OS intends to start a container, the control flow is transferred to the mini-OS to create a con-shim by way of switching the GPT to Priv-GPT. Initially, the mini-OS allocates memory for the con-shim from the internal memory pool and records the system call stack, shared memory, and private data areas used by the container in the con-shim. Subsequently, the mini-OS assigns a new identifier and notifies the secure monitor to create a new Shim-GPT for the container and its con-shim based on this identifier. Finally, the secure monitor sets the container's memory, the process's pagetable pages, and con-shim memory as No-access PAS in the OS-GPT and Normal PAS in the newly created Shim-GPT. In this way, the deprivileged OS will no longer be able to access these container-related memory areas. Following the initialization of the con-shim, RContainer then initializes the container. This

process involves loading the container image and executing the container's init process from the deprivileged OS. For a single container image, its confidentiality and integrity can be ensured through remote authentication [27] or secure storage [26]. The executable files in the container image are encrypted through a `rc_image_binary_crypto` script. Although the deprivileged OS can load these binaries, it cannot run them without the mini-OS using their private keys to decrypt them. The details of execution are introduced in the following.

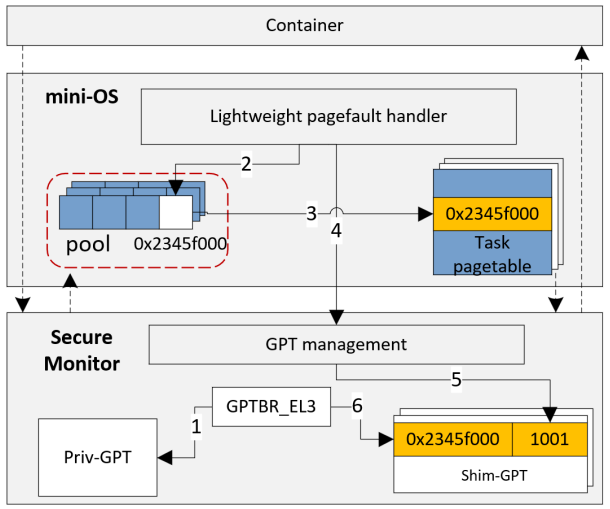
B. Task

1) Task creation: The execution of tasks within a container encompasses both their creation and termination. Task creation within a container can occur under three circumstances. The first scenario involves the creation of a process, where the newly formed task possesses its own copy of the caller's address space. In this case, once the deprivileged OS has run a child task and the task returns from the deprivileged OS, the mini-OS verifies and records the addresses of the new task structure and new pagetable in the container's con-shim. It then restricts the deprivileged OS's access to the task's CPU state and memory by instructing the secure monitor to set the relevant areas in the OS-GPT to No-access PAS. The second scenario involves the creation of a thread, implying that the newly formed task shares the same address space as its caller. In this case, after returning from the deprivileged OS, the mini-OS records the address of the new task structure and the process identifier to which the new thread-level task is associated. The new task's pagetable is the same as its process, so the mini-OS does not need to record the pagetable address again. The third scenario involves the creation of a process that replaces its caller's address space. This situation bears similarity to the first scenario, but with a key difference: it necessitates the loading of binaries and the disassociation of the old address space from the container. This includes erasing these memory areas and setting the corresponding region in the Shim-GPT to No-access PAS. Finally, the mini-OS confirms the accuracy of the task, pagetable, Shim-GPT, and CPU state. If these checks are passed, the mini-OS signals the secure monitor to switch to the container's Shim-GPT and resumes task execution in the container.

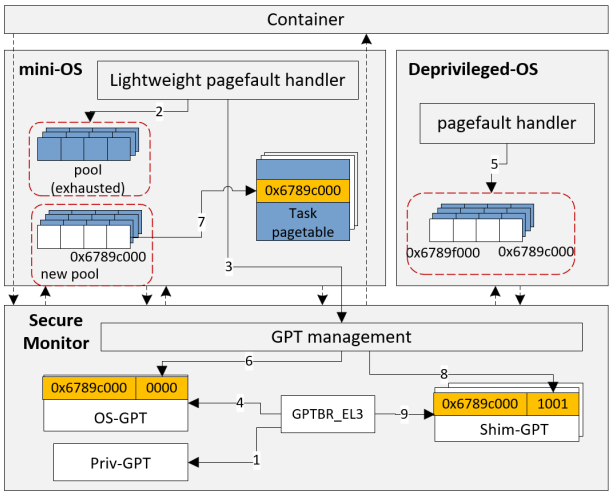
2) Task termination: In the event of task termination, the mini-OS eliminates the task structure information in the container's con-shim and purges the corresponding physical memory. If there are no remaining tasks within this address space, the mini-OS disassociates them from the container, mirroring the process outlined in the second scenario. This ensures a clean and efficient termination of tasks for containers.

C. Memory

1) Memory allocation: Typically, when a container or con-shim requires memory allocation, the control flow is transferred to the deprivileged OS for this task. Once the allocation is complete, the EL3 monitor adjusts both the Shim-GPT and OS-GPT properties of the newly allocated physical pages, returning the pages to the container or con-shim. However, this process can lead to frequent exception switching and significant performance overhead. To optimize this process, RContainer introduces the concept of 'reservation'. Specifically, at the onset of system bootup, RContainer allocates



(a) Fast pagefault workflow



(b) Slow pagefault workflow

Fig. 3: RContainer Pagefault Workflow.

several contiguous physical memory areas managed by the mini-OS. All physical pages within these memory areas are initially marked as No-access PAS in the OS-GPT and No-access in all Shim-GPTs. When a physical page is assigned to a specific container or con-shim, the secure monitor in EL3 marks the access properties of that page as Normal PAS in the container's Shim-GPT. The mini-OS maintains metadata information of these allocated pages, including page access permission, the container/con-shim to which the page belongs, and the number of times the page has been mapped. Additionally, the mini-OS features a lightweight pagefault handler to swiftly construct container-related pagetables.

2) Pagefault handling: Fig. 3a shows the fast pagefault workflow of RContainer. When a process within a container triggers a pagefault event, the control flow is redirected to the mini-OS by switching the current GPT to Priv-GPT in the secure monitor. The mini-OS then selects a physical page from the internal memory pool based on the pagefault address and populates this page into the process's pagetable. Subsequently, the mini-OS notifies the secure monitor to modify the Shim-

GPT where the process resides, marking the newly assigned page as Normal PAS in this Shim-GPT. Upon completion of these steps, the control flow is returned to the container by switching the current GPT to the container's Shim-GPT in the secure monitor. In addition, if the free pages in the memory pool are exhausted, RContainer employs the slow pagefault workflow (shown in Fig. 3b). Specifically, if the mini-OS detects a lack of free pages while handling pagefault events, it switches to the deprivileged OS to acquire new memory areas by way of switching the current GPT to OS-GPT in the secure monitor. The deprivileged OS allocates a contiguous physical memory area and passes the start address and size to the secure monitor. The secure monitor then designates these new pages as No-access in the OS-GPT. To safeguard against the deprivileged OS intentionally assigning incorrect or threatening addresses to the container, such as Iago attacks [19] and ROP attacks [32], the mini-OS checks that this memory area has not been mapped. Furthermore, RContainer focus on known Iago attacks, such as `mmap`, mini-OS maintains lists of processes' mapping range and manages pagetable updates. For virtual address (VA), mini-OS checks whether the VA belongs to valid mapping. And for physical address, RContainer then clears all physical pages in this memory area and populates one page into the process's pagetable. Finally, the mini-OS notifies the secure monitor to set the page as Normal PAS in the container's Shim-GPT and returns to the container.

3) Shared memory: Containers typically require parameter passing when executing system calls, often utilizing the application's memory buffers to transfer data between the container and the deprivileged OS. To prevent the deprivileged OS from directly accessing the container's memory, RContainer establishes a separate shared memory area for data transfer in con-shim, which is designated as Normal PAS in the OS-GPT. During the interposition of a system call exception, the mini-OS examines the arguments and transfers this data from the application's memory to the shared memory buffer in con-shim. Upon returning to the container, the mini-OS checks the return values and transfers this data back to the application's memory buffer. Additionally, copy-on-write presents another scenario that necessitates memory copying between containers and the deprivileged OS. In this case, the mini-OS verifies whether the source page belongs to the container and whether the destination page belongs to the deprivileged OS. If these checks are passed, the mini-OS notifies the secure monitor to set the destination page in the container's Shim-GPT to Normal PAS. This process ensures secure and efficient data transfer between the container and the OS.

D. Input/Output

1) Filesystem: A process within a container frequently accesses the file system via the I/O functions of the deprivileged OS. To safeguard the files within containers, RContainer employs two strategies. For files located inside a container, RContainer relies on the application's inherent encryption and decryption mechanisms to ensure file security. For files that need to be loaded and executed by the deprivileged OS, RContainer utilizes a "scan-encrypt" method to pre-process images. Specifically, RContainer offers a `rc_image_binary_crypto` script that scans the image, identifies all executable files, encrypts them using a public key, hashes these encrypted files, and repackages the container image. The private key, which pairs with the public key, is stored in the host's secure storage and can only be accessed by the mini-OS or the secure monitor.

During the pre-processing phase, all ELF files within a container image are encrypted, with the exception of the ELF headers. The ELF headers need to be parsed first by the deprivileged OS to determine the loading of the entire ELF file. When an executable binary file needs to be loaded and executed through an `exec` system call, as mentioned in the third case in Section IV-B, the mini-OS initially prompts the secure monitor to set the loaded binary-related pages in the process to No-access PAS in the OS-GPT and Normal PAS in the container's Shim-GPT. Subsequently, the mini-OS validates the hash values for integrity checks and decrypts the encrypted file. In this manner, when the deprivileged OS begins running this task, it will no longer have access to the binary's memory. Additionally, some binaries may be dynamically linked, such as libraries. These binaries need to be loaded by a loader at runtime, which is part of the container image. The loader typically mmaps these libraries through an FD-related system call. The mini-OS records all FDs of the libraries in the container's con-shim during container creation, checks the target encrypted library during loading, and finally decrypts and mmaps it like regular binary files.

2) Network: RContainer does not offer additional system-level protection for network I/O. The protection primarily relies on secure network transmission protocols, such as HTTPS [33], SSL/TLS [34].

E. Inter-process Communication

When a process initiates inter-process communication (IPC) through a system call, there may be two situations: shared memory and message passing. For the first situation, the handling method is similar to section IV-C3, where a process may use the `shmget` and `shmat` system calls to request the deprivileged OS allocation or mapping of shared memory. The mini-OS selects a page from the shared memory pages within the container's con-shim, ensuring that this physical page can only be mapped to the relevant virtual memory areas. For the second situation, a process may utilize a pipe, socket, or message to invoke IPC. RContainer employs encryption to safeguard IPC within a container. Under these circumstances, a specific communication channel is established for data transmission, associated with an IPC-related file descriptor (FDs). The mini-OS interposes on these system calls and encrypts these channels using the key employed during container initialization, based on its FD.

V. Implementation

A. FVP Prototype

We have implemented one prototype of RContainer on an ARM64v9.4-A Fixed Virtual Platform (FVP) [11], which supports ARMv9-A instruction set with RME hardware extension, GPT-supported ATF secure firmware, and ARM64 Linux kernel. Our system runs on Linux 6.2-rc2 with Trusted Firmware-A v2.8.0, using Docker container at the user-level with version 1.5. RContainer provides several interfaces which are shown in Table III.

TABLE III: RContainer Interfaces.

RContainer Call	Description
rc_create_shim	Create new con-shim for a container
rc_destroy_shim	Destroy con-shim of a container
rc_create_container	Create a new container
rc_destroy_container	Destroy a container
rc_malloc_mm	Allocate memory for container/con-shim
rc_set_pte	Update PTE of a process/thread in container
rc_copy_page	Copy page to a container
rc_set_vma	Update vma of a process/thread in container
rc_set_iopte	Update IO PTE of IO device
rc_ipc_in	Handle ipc within a container
rc_ipc_out	Handle ipc between containers
rc_task_clone	Run a new process/thread in a container
rc_task_exec	Run program in a new address space in a container
rc_task_exit	Exit a process/thread in a container
rc_switch_to_depriv	Switch contexts to Deprivileged OS
rc_switch_to_miniOS	Switch contexts to mini-OS

The TCB of RContainer consists of ATF secure firmware in EL3 and the mini-OS in EL1. The ATF secure firmware is the native TCB in the ARM architecture, running in Root mode. RContainer adds three SMC calls in the ATF: SMC_GPT_SET_GPTBR_EL3 to switch GPT, SMC_GPT_SET_GPI_BATCH to batch set the GPIs of consecutive physical pages in the target GPT, and SMC_GPT_READ_ANY_PHY_ADDR to obtain the GPI of a physical page in the target GPT. In addition, in order to reduce the performance overhead during GPT creation, RContainer has set a Shim-GPT template, in which the mini-OS and deprivileged OS memory regions have been set to No-access PAS. For the mini-OS, RContainer defines a new section for the mini-OS in kernel's linker vmlinux.lds.S, and forces this section to be loaded at a fixed address, which will use the independent PTE in the L0 pagetable page. As we all know, as long as the PTEs of the mini-OS and the deprivileged OS are separated in the L0 pagetable page, their related L1, L2, and L3 pagetable pages can automatically separate. Furthermore, we leverage a shadow L0 solution, which means that a real L0 (No-access in OS-GPT) is loaded into ttbr1_EL1 for the mini-OS, and a shadow L0 is loaded into ttbr1_EL1 for the deprivileged OS. These two L0 pages have the same mapping content, so there is no need to flush the MMU TLB during control flow switching. After the kernel pagetable is built, RContainer sets the mini-OS's section and related pagetable pages (L1, L2, L3) to No-access PAS in deprivileged OS's GPT. L0 pagetable page is the only shared pagetable page between the mini-OS and deprivileged OS, and based on our observation experience, once the kernel pagetable is built, L0 is rarely modified. Therefore, RContainer copies the L0 in a new physical page for mini-OS and sets the page to No-access in OS-GPT. All physical pages in the Normal World are set to Normal PAS in the mini-OS's Priv-GPT.

The mini-OS interposes on all system calls, interrupts, and exceptions. RContainer modifies all entry points in the exception vector table, invoking rc_switch_to_depriv before the processing of the exception, and calling rc_switch_to_miniOS after the exception has been handled. When an exception occurs while a container is running, the control flow transfers to the exception vector table and first calls rc_switch_to_depriv. During this process, the mini-OS checks the corresponding parameters and, if necessary, uses the shared memory in the container's con-shim to pass the parameters. It then switches to the deprivileged OS with OS-GPT. After the exception processing is completed, rc_switch_to_miniOS is invoked to validate the results and switch to the container's Shim-GPT. In this manner, even if the deprivileged OS bypasses these two calls by modifying the mapping or executing the instructions, it still cannot access the container's memory due to the GPT limitation. Additionally, RContainer enforces the disabling of interrupts during the running of the mini-OS.

RContainer uses GPT as the fundamental mechanism for shim-style memory isolation. We make minor modifications to Linux's memory allocator. Specifically, RContainer introduces a new flag to indicate that the currently allocated memory is unique to con-shim and notify the mini-OS to modify the GPT. Additionally, for specific global variables [13], RContainer allocates the fixed number within each con-shim and maintains a summary table in the mini-OS to track and monitor these variable allocations. In a multi-core scenario, each core maintains its own GPT pagetable for the currently running process, necessitating the consideration of synchronization and bypass issues. Similar to [10], RContainer utilizes a spin lock in ATF for GPT multi-core synchronization, thereby preventing attacks that might use multi-cores to bypass GPT protection. Additionally, RContainer invalidates GPT-related TLB entries when switching or modifying the GPT, and disables the shared TLB for containers by setting the CnP bit in the TTBR to 0. Furthermore, All other processes (unrelated to containers) have a universal GPT that shields memory about mini-OS and containers, and is used by cores by default. When scheduling occurs, mini-OS validates whether the target task context belongs to containers/con-shim or deprivileged OS, and notifies the secure monitor to switch the valid GPTs.

B. Hardware Prototype

Due to FVP's lack of cycle-timing accuracy [11], we implement another prototype of RContainer on a Firefly-RK3399 ARMv8 SoC development board [12], with Linux-firefly-4.4.149 as hostOS kernel. Furthermore, we simulate the overhead of the GPT by flushing the TLB, modifying the specified registers, and counting cycles for GPT management. Specifically, we add the same three functions as in FVP to ATF and maintain a GPT-multi structure for GPT management. Then, for making subsequent performance tests closer to the real situation, we use SCTRL_EL3 register for GPT-related register (GPCCR_EL3 and GPTBR_EL3) access simulation, and use tlbi for GPT-related flush. The implementation of mini-OS is the same as the one in FVP. For IPC-related implementation in Section IV-E, RContainer can protect IPC in the same way as it protects shared memory (Section IV-C3), or it can utilize encryption to safeguard IPC. To simplify deployment, we consider IPC encryption as an optional feature. Furthermore, we choose Advanced Encryption Standard (AES) with Electronic Code Book (ECB) mode as encryption method and use the native AES library in Linux to encrypt and decrypt IPC channels. We use eMMC RPMB for storing private key.

VI. Security Evaluation

A. Mini-OS as a Reference Monitor

The mini-OS is actually a reference monitor used to interfere with the interaction between containers and the deprivileged OS. Therefore, RContainer should ensure that the mini-OS is tamper-proofed and non-bypassable during its lifetime.

Tamper-proofed. During system bootup, RContainer employs a secure boot to ensure the integrity of the mini-OS. Post-boot, the secure monitor establishes Priv-GPT and OS-GPT for the mini-OS and deprivileged OS, respectively, setting the mini-OS related memory to No-access PAS in the OS-GPT. Even if the deprivileged OS tampers with the mapping of the mini-OS by modifying L0, when switching to the mini-OS, the secure monitor checks the mini-OS related PTE in L0 and detects the attacks immediately. In addition, the deprivileged OS may leverage a peripheral to perform DMA attacks. RContainer defends against these attacks by SMMU-enforced GPT [35], which is described in Section VI-B3).

Non-bypassable. The deprivileged OS may attempt to bypass the mini-OS in two ways: unmapping the switch to the mini-OS and utilizing TLB. In the first scenario, only the mini-OS has the capability to notify the secure monitor to switch to Shim-GPTs. Therefore, even if the deprivileged OS manages to bypass the mini-OS, it remains unable to access any containers' memory due to the limitations imposed by the OS-GPT. For example, the deprivileged OS may attempt to change the page table root (TTBR) or even disable paging to bypass RContainer. However, in RContainer, access to physical memory is controlled by `GPTBR_EL3`. Therefore, modifying `TTBRx_EL1` does not change access permissions. In the second scenario, RContainer invalidates the GPT-related TLB and disables global sharing. Furthermore, a malicious deprivileged OS or con-shim may bypass the memory isolation of mini-OS through GPT TLB. RContainer prevents this attack by flushing the GPT TLB entries of the mini-OS when switching out of the mini-OS. On the contrary, since the mini-OS is in our TCB, there is no need to flush the GPT TLB when the system switches from con-shim or deprivileged OS to mini-OS. As a result, there is no feasible method for the deprivileged OS to access the memory of the containers.

B. Defend against the Deprivileged OS

1) MUMA attacks: Compared to a single application, a container may contain multiple applications, which makes it possible for the deprivileged OS to leverage MUMA attacks to compromise the container's security [7].

Signal-related attacks. The deprivileged OS may forge random signals and inject them into the container, introducing race conditions. Additionally, the deprivileged OS may corrupt the register context during signal handling, causing the container to inadvertently leak secrets. RContainer defends against these attacks by leveraging the shim-style isolation. Typically, the deprivileged OS may inject a stack frame into the user stack for signal handling, resulting in control-flow switching upon function return. In RContainer, the shim-GPT safeguards the user stack, making it inaccessible to the deprivileged OS. Any access from the deprivileged OS will trigger a GPT fault, prompting the mini-OS to verify the operation.

Futex-related attacks. The deprivileged OS may tamper with futex locks to disrupt the synchronization mechanism within the container. In RContainer, the mini-OS records futex locks and their associated processes. Before control flow returns to a process within the container, the mini-OS checks if the process is waiting on any futex locks. If the process is waiting on a futex lock variable, its execution is ignored unless it is explicitly woken up.

IPC-related attacks. The deprivileged OS may compromise the security of a container during IPC by tampering with shared memory or communication channels. RContainer defends against such threats by monitoring shared memory and encrypting the channels, as described in Section IV-E.

2) Iago attacks: The deprivileged OS may attempt to attack the container by maliciously altering system call return values. In RContainer, the mini-OS provides control flow protection between the container and the deprivileged OS, ensuring verification of system call returns prior to container entry. The existing Iago attacks [19] primarily involve system calls related to memory (e.g., `mmap`). RContainer ensures the address returned does not overlap any existing memory regions (e.g., `mmap` in Section IV-C2). Compared to existing solutions [10], [7], RContainer integrates its defenses into Normal EL1, avoiding excessive code inflation in the highest privilege EL3 and frequent switches between different Worlds.

3) DMA attacks: The deprivileged OS may try to access the container or mini-OS memory by exploiting a peripheral (e.g., GPU) to perform DMA attacks. RContainer defends against these attacks by using SMMU-enforced GPT [35]. Specifically, RContainer employs a global SMMU-GPT (termed IOGPT) for the deprivileged OS, with all memory attributes set to No-access by default, blocking OS and peripherals from accessing arbitrary memory via DMA. When the deprivileged OS allocates memory for DMA, the mini-OS records the VA-to-PA mapping and sets the corresponding page in IOGPT to Normal-access, ensuring only explicitly declared I/O memory (e.g., data encrypted by the container user) is accessible.

C. Defend against Malicious Containers

RContainer has also considered attacks from malicious containers. An attacker may leverage a malicious container to attack the deprivileged OS or other containers by exploiting vulnerabilities or abstract resources. RContainer ensures that even if the container escapes to the deprivileged OS, it cannot access the memory of other containers by creating an isolated con-shim for each container. In addition, each con-shim limits the stack, shared memory range, and global sensitive data of the container, and works in conjunction with the mini-OS to prevent partial DoS from the container, such as resource abuse.

D. Evaluation of Practical Attacks

We use the FVP prototype for security evaluation and have evaluated 30 CVEs shown in Table IV. These CVEs include vulnerabilities originating from the deprivileged OS and containers (or container engines). Furthermore, based on the harmful consequences of being exploited, these CVEs are classified into four categories: privilege escalation, memory corruption, information leakage, and denial-of-service.

TABLE IV: Case Study.

CVE-*	Description[1]
2024-21626	Internal file descriptor leak in runc
2022-23222	Pointer arithmetic availability via *_OR_NULL pointer
2021-32606	User-after-free in isotp_setsockopt in net/can/isotp.c
2021-28972	User-tolerable buffer overflow during dev name entry
2020-14386	Kernel memory corruption due to arithmetic flaw
2020-8835	Out-of-bound access due to unrestricted register bound
2019-14271	Code injection occurs when the nsswitch loads a library
2019-10144	Do not isolate containers' processes when 'rkt enter'
2019-5736	Mishandling of file descriptor in /proc/self/exe
2018-18955	Improper handling of nested user namespace in write
2018-15664	Improper archive operations on a frozen filesystem
2018-15514	Unverify the validity of the deserialized .NET objects
2017-1000112	Memory corruption from UFO/non-UFO path switch
2017-7308	Improperly validation of certain block-size data
2016-9962	Improper execution to file-descriptors
2016-7117	Use-after-free in __sys_recvmmsg in net/socket.c
2016-5195	Race condition in mm/gup.c for handling CoW
2016-3697	Improper treats a numeric UID as username
2016-1582	Improper rights when switching container privilege
2016-1581	Improper permissions for ZFS.img when loop setup
2016-1576	Improper restricted mount namespace
2015-3630	Use weak permission for /proc/ operation
2015-3629	Unverified symlink when respawning a container
2015-3627	Open unverified file descriptor before chroot
2015-1335	Improper directory traversal operation in lxc-start
2014-9357	Improper handling of untrusted archive extraction
2014-6407	Symlink and hardlink when pulling docker images
2013-6441	Use read-write permissions when mounting /sbin/init
2010-4258	Improper handling of KERNEL_DS get_fs value
2010-2959	Integer overflow to function pointer overwrite

[1]The causes, exploitation, and defense simulation of these CVEs are detailed in Appendix Table XI.

Privilege escalation. Based on our observations of vulnerability exploitation, the most common attack path for privilege escalation is from the container to the host user-mode and then to the host root-mode. In this attack path, the attacker may exploit vulnerabilities in the container engine (e.g., CVE-2024-21626) or in the kernel (e.g., CVE-2022-023222) to gain higher privileges. Privilege escalation also serves as an intermediate step toward compromising confidentiality or integrity. In traditional systems, the root user typically possesses the highest level of privilege and can perform any operation. In our evaluation, we found that more than half of these CVEs result in privilege escalation, enabling the attacker to successfully perform root-user operations such as kernel module `insmod` and arbitrary code execution. While RContainer does not prevent attackers from gaining privileges, it effectively mitigates further attacks, acting as a defense against malicious hosts in our threat model. In RContainer, the host OS is out of our TCB and operates with restricted permissions. Therefore, even if an attacker obtains root-user privileges, they cannot compromise the security of other containers. For instance, consider CVE-2019-5736, which allows a malicious container to overwrite host runc binaries and subsequently obtain host root privileges through `/proc/self`. In our emulation of a deep attack, we first deployed a normal container, used `docker run exec` to write our `/bash/bin` to the container, and then tried to access the kernel's pagetable pages. As expected, since the container's Shim-GPT has no permission to access the kernel's pagetable pages, this operation triggered a GPT fault.

Memory corruption and information leakage. The primary objective of many attacks is to compromise the integrity and confidentiality of memory. Memory corruption is often caused by memory overflow (e.g., CVE-2021-28972) and improper parameter handling (e.g., CVE-2010-4258). Information leakage, on the other hand, can be attributed to improper permission settings (e.g., CVE-2015-3630) and weak boundaries (e.g., CVE-2020-8835). These vulnerabilities enable attackers to write to or read from memory that does not belong to the compromised components. In RContainer, both the host OS and each container have their own independent memory regions with strong physical boundary isolation, ensuring non-interference. Any attempt to perform out-of-bounds read or write operations results in a GPT fault. To enhance the detection of such faults, we have modified the GPT fault handler to generate warnings that identify the malicious components and target addresses. For instance, let's consider CVE-2020-8835. This vulnerability arises from the incorrect calculation of value ranges in the `BPF_REG_STATE` register by the BPF verifier during the verification of the `BPF_JMP32` instruction, leading to out-of-bound access. An attacker can leverage this vulnerability to locate the `init_pid_ns`, obtain the `task_struct` of the target container using `find_task_by_pid_ns`, and access the memory of any container. In our emulation of a deep attack, we attempted to modify the pagetable pages based on `mm_struct` pointer in the `task_struct` when the attacker had gained access to the target container's `task_struct`. As expected, this operation triggered a GPT fault.

Denial-of-service. DoS attacks are typically aimed at disrupting the availability and stability of containers or host platforms. In our evaluation, the causes of such attacks include various factors, such as memory overflow (CVE-2020-14386), improper parameter handling (CVE-2018-18955), and out-of-bound access (CVE-2017-1000112). The fundamental reason why most of these attacks can succeed is that the system lacks strong boundary isolation. In addition to the aforementioned causes, resource abuse is also an important way for malicious users or containers to perform DoS attacks. The main reason why these attacks can succeed is mostly because the system lacks restrictions on the resources or permissions that containers can use. In RContainer, through shim-style isolation, we have limited the scope of each container in kernel-layer and to some extent alleviated memory-related DoS attacks (e.g.,[13]) caused by compromised containers. The DoS attacks caused by the compromised host are out of our consideration.

The causes, exploitation, and defense simulation of 30 CVEs are detailed in Appendix Table XI. Through the security analysis and evaluation, we have found that regardless of the type of vulnerability, the necessary conditions for its successful exploitation are the problems of excessive permissions and weak-boundary isolation in the native system design. In RContainer, we have restricted some permissions of the host OS and containers to ensure that they cannot perform non-essential operations without monitoring. Furthermore, through a hardware-enforced strong isolation mechanism, RContainer effectively suppresses unexpected behavior of compromised containers and host OS.

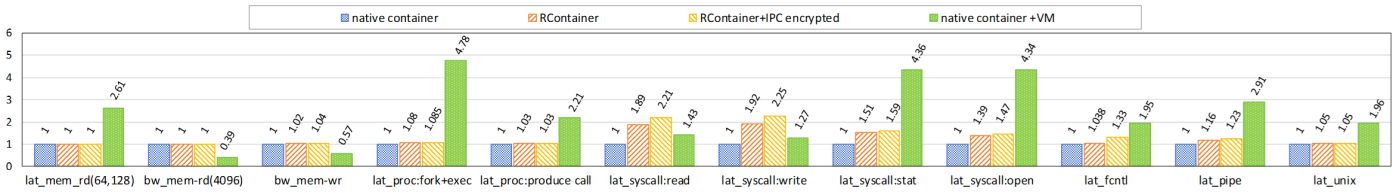


Fig. 4: Lmbench Performance. (Lower is better)

VII. Performance Evaluation

This section evaluates the overall performance and scalability of RContainer by answering the following questions:

Q1: What is the implementation complexity of RContainer? (§VII-B)

Q2: How many system calls does RContainer support? (§VII-C)

Q3: How do microbenchmarks perform under RContainer? (§VII-D)

Q4: How do real-world applications perform under RContainer? (§VII-E)

Q5: What is the lifecycle cost of containers under RContainer? (§VII-F)

Q6: How does RContainer perform when running multiple containers? (§VII-G)

A. Methodology

Given the absence of publicly accessible servers or development boards that support RME, we use the hardware prototype for performance evaluation. Specifically, we use Firefly-RK3399 ARMv8-A board featuring a 2-core Cortex-A72 64-bit and 4-core Cortex-A53 64-bit, clocked at 1.8GHz, with 4GB RAM, running on Linux-firefly-4.4.149, Trusted Firmware-A-1.3, and Docker 25.0.0-beta.1. Our network configuration utilizes an Intel I219 and a TL-SG2008D switch. Based on these setups, we conducted performance evaluation on the Firefly-RK3399. On this basis, we first evaluated the complexity and system call coverage. Then, we ran microbenchmarks and application workloads, as detailed in Table V, across four distinct configurations: (1) Native Docker (serving as the baseline); (2) RContainer; (3) RContainer with encrypted IPC; and (4) Native Docker within KVM-VM (container running in guest VM with ubuntu-server-18.04, Linux 5.4.0, docker 25.0.0, 2 vcpus, and 512MB). Lastly, we evaluated the lifecycle cost and the concurrent overhead.

B. Status Quo and Complexity

To answer the first question (**Q1**), we utilized CLOC [46] to evaluate the complexity of RContainer. The results show that RContainer has introduced a total of 2,647 lines of code, including only 130 lines of C in ATF (running at EL3), 2,272 lines of C and 112 lines of assembly in Linux, and 133 lines of Python in a user-level python-script for binary encryption within images. In addition, we have chosen AES-ECB as the IPC encryption algorithm and introduced optional 2,308 lines of C. Table VI shows the breakdown of RContainer TCB. In addition, RContainer uses approximately 200 SLoC patch in Linux for con-shim, which are out of TCB. Compared to Shelter [10], RContainer provides a more scalable and stable design. Firstly, Shelter's existing security features do not cover all potential Iago attacks or other threats. To defend against more attacks, the newly added code in EL3 by Shelter would rapidly increase. In contrast, RContainer centralizes most security functions in EL1. Even integrating new security features in the future, RContainer will not significantly increase EL3's TCB. Secondly, security functions generally belong to runtime code, as most attacks occur at runtime. Our analysis of ATF-1.3 Code, shown in Table VII, indicates that only 2.7% (≈ 11K SLoC) of code corresponds to runtime support. Focusing strictly on runtime security, Shelter increases EL3 runtime code by nearly 18%, while RContainer only by about 1.2%.

TABLE V: Benchmarks and Configuration.

Name	Configuration
Lmbench	lmbench v3.0-a9-1 [36] micro benchmark
Kernel build	Linux longterm: 4.19.309 with defconfig, `make -j2`
Hackbench	Hackbench in Linux test project-20240129 [37] with 25 groups, 40 pairs of sender and receiver, and 100 datasize
Apache	Apache server v2.4.58 [38] handling 100 concurrent requests from remote Apache Bench v2.3 [39], serving from 4KB to 512KB pages with 1GB data per-page
Memcached	Memcached v1.6.22 [40] using memtier benchmark v2.0.0 [41] with default configuration, serving from 16Byte to 512Byte data
MySQL	MySQL v8.2.0 [42] handling requests from sysbench benchmarks v1.0.2 [43], running workload using six types of operations with 5 tables, 1,000 counts per-table, 10 threads, and lasting for 30 seconds
Nginx	Nginx v1.25.3 [44] handling requests from remote ApacheBench v2.3 [39], serving from 4KB to 512KB pages with 1GB data per-page, 500 requests, and 100 concurrency
Netperf	Netperf 2.7 [45] testing `TCP_STREAM`, `TCP_CRR`, `TCP_RR`, `UDP_STREAM`, `UDP_RR` five cases, with lasting for 60 seconds

TABLE VI: The Breakdown of RContainer TCB.

Component	Functions	Language	SLoC
Mini-OS	Con-shim management	C	373
	Memory mapping protection	C	511
	Boot protection	C	200
	Context switch protection	C and Assembly	350
	Syscall Interposition	C and Assembly	500
	IPC encryption (optional)	C	2,308
	Others	C	450
Image scan-encrypt	Image parse	Python	35
	Image encrypt	Python	98
Secure monitor	GPT management	C	130
Total			4,955

TABLE VII: The Breakdown of Native EL3 Code.

Function	SLoC
Platform bootup	218,909 (51.92%)
TrustZone support	17,460 (4.14%)
Realm World support	17,408 (4.13%)
Normal runtime support	11,387 (2.70%)
multi-Platforms/drivers	156,457 (37.11%)
Total	421,621

C. System Call Coverage

To answer the second question (**Q2**), we utilized the Linux Test Project (LTP)-v20240129 [37] system call test suite in both native container and RContainer to test our system call coverage. LTP consists of a total of 1,409 test cases. During the testing process, 970 test cases succeed, 280 test cases were skipped, and 159 test cases failed in both the native container and RContainer. Except for some architecture-related cases, due to our usage of kernel-firefly-4.4.194, some new features are not supported like `io_uring`. Overall, the system call coverage of RContainer is relatively the same as that of the native container.

D. Microbenchmarks

To answer the third question (**Q3**), we utilized Lmbench and the Performance Monitor Unit (PMU) in four configurations mentioned in Section VII-A.

1) Lmbench: We ran Lmbench using a total of 12 use cases of five types, including memory-related, process-related, file system-related, and IPC-related. These tests are run 10,000 times to get the average values. Fig. 4 shows the results. We normalized the results so that a value of 1.0 means the same performance as the native container. Specifically, for memory-related cases, such as `lat_mem_rd`, `bw_mem_wr`, there is nearly no overhead since RContainer hardly intercepts pagetable translation. For process-related cases, such as `fork+exec`, `produce call`, RContainer introduces approximately 10% overhead. This is because it is necessary to validate these parent-child processes' task structures and their address spaces, and further modify the GPT tables. For file system-related tests, such as `read` and `write`, the measurements show the highest overhead, about twice as much as native Docker. This is because the deprivileged OS cannot directly access containers' memory. Consequently, RContainer needs to copy system call parameters and data back and forth, thereby incurring overhead. For IPC-related cases, such as `lat_pipe` and `lat_fcntl`, the overhead of RContainer with IPC encryption shows about 20%, which comes mainly from encryption/decryption.

2) PMU: Further, we also counted GPT-related functions through PMU. As shown in Table VIII, although initialization introduces the most time because of the need to prepare environment, including multi-GPT and mini-OS, it is performed only once during system runtime. In addition, setting GPI overhead is related to the size of the physical page as well as GPT granularity. Our configuration is a 4K physical page in 4K-level GPT and the result is about 530μs. Context switch between deprivileged OS and mini-OS is about 492μs. Creation of con-shim is about 530μs. Compared to creation, termination takes more time (about 637μs) because it requires cleaning all con-shim-related memory to ensure that there is no residual information.

TABLE VIII: GPT-related Microbenchmarks.

Function	Description	Times (μs)
Initialization	Initialize the multi con-shim environment	45,607
OS switches	Switch between OS and mini-OS	492
Creation	Create a con-shim instance	530
Termination	Destroy a con-shim instance	637
Set GPI (4KB)	Set physical pages' GPIs in target GPT	530

TABLE IX: Kernbench Performance.

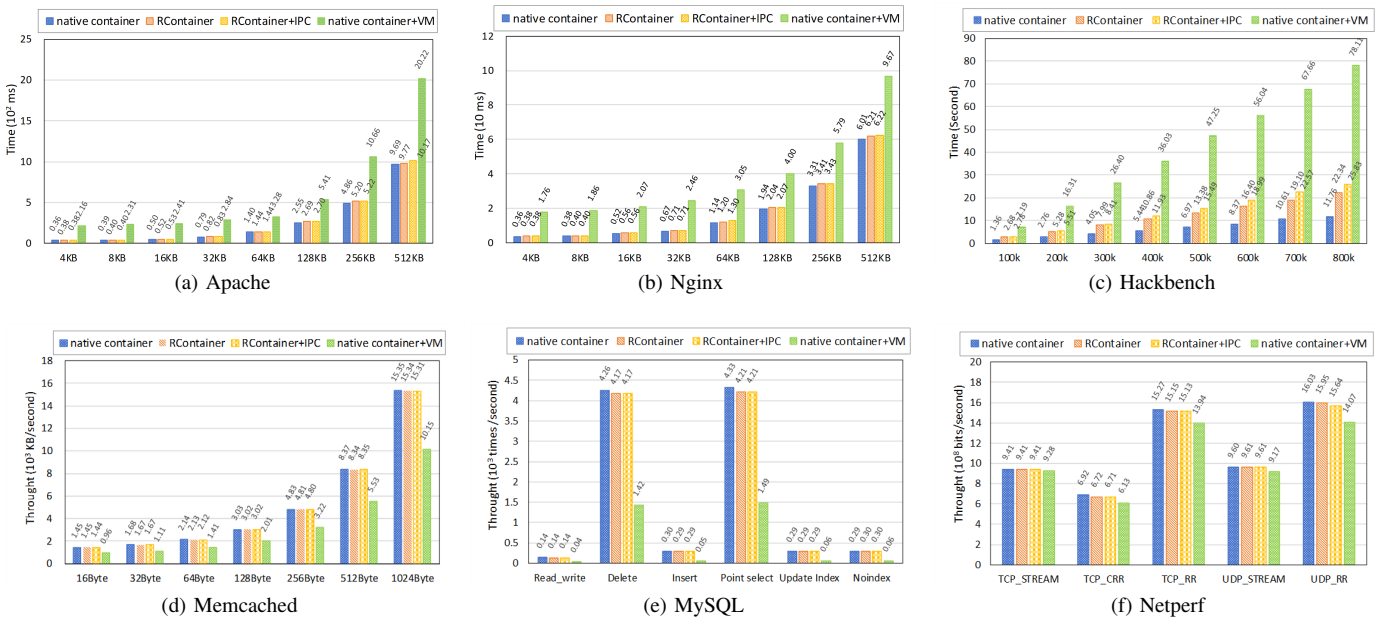
Native container	RContainer	RContainer+IPC	Container+VM
3248.38s	3393.12s	3463.1s	8060.29s

E. Application Workloads

To answer the fourth question (**Q4**), we employed kernel compilation using linux-4.19.309 with defconfig and six application workloads, tailored for real-world scenarios, in four configurations mentioned in Section VII-A.

1) Kernel build: We have conducted kernel compilation using linux-4.19.309 with defconfig. We tested ten rounds and took the average value. The results are shown in Table IX. Overall, the performance overhead of RContainer and RContainer with IPC-encrypted for kernel compilation is approximately 4.5% and 6.6%, which is much better than virtualization.

2) Real-world applications: We have utilized Apache, Memcached, MySQL, Nginx, Netperf, and Hackbench to evaluate RContainer's real-world application workload. Each workload underwent ten rounds of testing, adhering to the configuration outlined in Table V. The average runtime for each benchmark is depicted in Fig. 5. The results indicate that the overhead of RContainer on real-world application workloads is relatively lower compared to microbenchmarks. Specifically, for Apache (Fig. 5a) and Nginx (Fig. 5b), we tested from 4KB to 512KB with 1GB datasize per-page and demonstrated the performance impact by measuring the time taken for the requests. The overhead of RContainer compared to native Docker is from 4%˜7%, with an average value of 5%, whereas for virtualization, it exceeds 50%. For Memcached (Fig. 5d), we tested request data sizes ranging from 16Byte to 1024Byte with 2,000,000 ops and demonstrated the performance impact through throughput measurements. From the results, it can be seen that as the data size continues to increase, the throughput increases almost proportionally. The performance overhead of RContainer remains stable at 0.3%. For MySQL (Fig. 5e), we selected six commonly used operations with 10 threads to test throughput. The result shows that the overhead of RContainer is from 0.2%˜0.3%. Even in IPC-encrypted mode, the overhead of RContainer is only 0.7%, which is much better than virtualization (about 34%). For Netperf (Fig. 5f), to avoid excessive variance caused by the previous connection not being released, we paused for 60 seconds after each test. The results show that the overhead of RContainer is less than 3%.



(a) Apache (b) Nginx (c) Hackbench

(d) Memcached (e) MySQL (f) Netperf

Fig. 5: Performance of Apache, Nginx, Hackbench, Memcached, MySQL, and Netperf. (For the first three workloads, lower is better; for the last three workloads, higher is better)

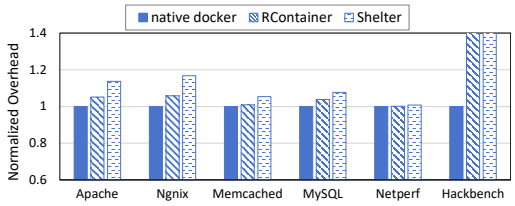


Fig. 6: Performance Comparison between RContainer and Shelter. (Lower is better)

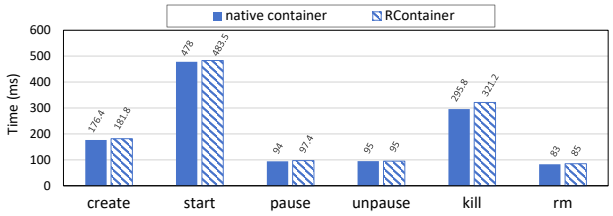


Fig. 7: Lifecycle Cost in RContainer Compared to the Native Docker. (Lower is better)

The results of Hackbench (Fig. 5c) are the worst performance (nearly increased 1x) in all benchmarks. There may be two main reasons. The first reason is that Hackbench creates a large number of threads (even numbers) to serve as senders and receivers of messages. This process involves frequent thread creation, leading to frequent memory-attribute checks and GPT modifications, resulting in an amount of context-switching. The second reason is that the communication between senders and receivers in Hackbench is done through IPC (mainly pipe and local socket), which means that each message transfer involves encryption and decryption operations, resulting in increased overhead. However, the overhead of RContainer is still reduced by nearly 70% compared to virtualization.

3) Performance Comparison: Furthermore, we compared the overhead of real-world applications between RContainer and Shelter. Since Shelter is not designed for container scenarios, we did our best to simulate its functionalities and port it to our RK3399 development board. We ran six application containers (shown in Appendix Table X). Fig 6 shows the normalized overhead results. Overall, for large-scale applications (e.g., Apache, Nginx, Memcached, MySQL), RContainer exhibits a smaller performance overhead (about 5.7% on average) than Shelter. According to Shelter's open-source project, as all security features are leveraged in EL3, we need to add some context analysis of EL1 into EL3 to bridge the semantic gap between different exception levels, increasing the overhead. Additionally, both RContainer and Shelter show the worst performance on Hackbench, likely due to frequent process creation and IPC (as mentioned in Section VII-E2).

F. Container Lifecycle Cost

To answer the fifth question (**Q5**), we used busybox:1.36.1-glibc image [47] to evaluate the lifecycle cost of RContainer. Specifically, we measured the execution time of `docker create`, `docker start`, `docker pause`, `docker unpause`, `docker kill`, and `docker rm`, testing each command 10 rounds and taking the average. Fig. 7 shows the results. Overall, the time overhead for most commands under RContainer is less than 5%, with the overhead for `docker create` being 3.06%, `docker start` 1.17%, `docker pause` 3.62%, `docker unpause` 0.21%, and `docker rm` 2.41%. The `docker kill` command shows the highest overhead, approximately 8.5%. This is because, during the command execution, RContainer needs to clear the

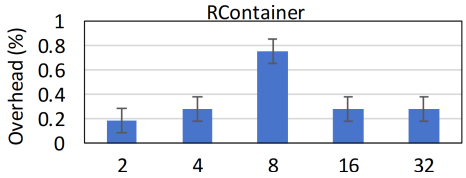


Fig. 8: Kbuild Overhead for Different Number of Containers.

container-related memory and modify the permissions of the corresponding pages in GPTs (both shim-GPT and OS-GPT), leading to increased overhead.

G. Performance of Multiple Containers

Finally, to answer the scalability question (**Q6**), we demonstrated how RContainer performs by running kernbench under different numbers of containers. We ran ten rounds and took the average values both on native docker and RContainer. Fig. 8 shows the overhead of concurrently running kernel build (Linux-4.19.309) with `allnoconfig` on different numbers of containers. Due to the limitations of our development board, when the number of concurrent containers exceeds 32, neither native docker nor RContainer can run normally (out-of-memory). In general, In RContainer, the overhead on multiple containers introduced by RContainer is negligible because the operations of each container are limited to its own regions (con-shim and Shim-GPT), which are not shared by others. The case of 8 containers shows the worst performance, considering the overall results, it may be caused by runtime variation.

Performance issue. Due to the current inability to obtain a real machine of ARMv9-A, the performance evaluation of RContainer is mainly simulated on the ARMv8-A machine. Although we have tried to simulate GPT-related operations as much as possible, there is still a high probability of performance deviation from a real ARMv9-A machine in the future. Nevertheless, we believe the performance evaluation is informative with our simulation work.

VIII. Related Work

OS-based mechanism. Container technology relies heavily on Linux Security Features (LSFs) or Linux Security Modules (LSMs) [48], [49], such as Namespaces, CGroups, Capabilities, and Seccomp. Some researchers have further expanded these technologies to enhance isolation among containers. ContainerDrone [50] uses CGroups and a MemGuard kernel module to protect containers from CPU and memory DoS attacks. Sun et al [51] present a novel approach to enable LSMs called security namespace, allowing each container to have its own security profiles. These methods can enhance the security of containers but involve a fragile OS as the TCB. Therefore, they cannot defend against threats from the untrusted OS.

Hypervisor-based mechanism. In order to defend against an untrusted OS, researchers have deployed virtualization technology to protect containers [52], [2], [1], [53], [54], [55], [4], [3]. Inktag [2] splits an application into sensitive and non-sensitive parts by using two nested page tables in the virtualization, and enforces memory isolation based on para-virtualization. Hosseinzadeh et al [55] first uses vTPM to measure the integrity of the container. X-Container [4] aims to enhance the security of containers in the cloud. It provides an entire library OS for each container and uses Xen hypervisor to achieve isolation. Compared to RContainer, X-Container has a larger TCB (including Xen hypervisor and library OS) and requires modifications to containers. BlackBox [3] involves a container security monitor to provide fine-grained protection. By extending the virtualization layer on ARM, it creates an independent memory area for each container. Both RContainer and Blackbox provide fine-grained container memory isolation, but RContainer uses hardware-based security isolation and introduces smaller TCB. gVisor runs a userspace kernel for each container. Compared to these solutions, RContainer employs the concept of "shim-style isolation" + "miniOS". It isolates the data plane not the whole kernel-plane, ensuring a lightweight solution with a small TCB.

Hardware TEE-based mechanism. Various hardware-based approaches [56], [7], [30], [57], [58], [59], [60] have been explored to enhance container security, such as Intel SGX [61], MPK, ARM TrustZone [6]. SCONE [30] presents Secure Linux Container with Intel SGX to protect containers from privileged software. By providing a customized thread and new C library with SGX enclave, it achieves low-overhead system call support. CubicleOS [56] provides a specialized library OS to run inside the container, and isolates applications by MPK. TZ-Container [7] leverages a container shield in TrustZone, which interferes with pagetable updates and control flow, to provide an isolated execution environment for each container. TrustShadow [8] introduces a tiny system with TrustZone to support sensitive application operations. Unlike these approaches, RContainer does not need additional OS libraries, which makes our TCB as small as possible and does not limit the system calls that containers can use. SHELTER [10] uses CCA [14] to provide a userspace isolation for application security. Unlike SHELTER, RContainer uses a secure EL1 component, called a mini-OS, instead of EL3 to undertake most of the security functions. This mini-OS is used as a security monitor between the container and the untrusted OS, which can integrate more defense mechanisms (e.g., Iago and MUMA protection) without expanding EL3 code. So the isolation achieved by RContainer is as strong as Shelter and the amount of newly added code running with EL3 privileges is reduced by one order of magnitude. In addition, Shelter focuses on isolation in userspace; in contrast, RContainer addresses the isolation challenges which include not only how to isolate the containers from each other, but also how to isolate the mini-OS from the deprivileged OS and how to isolate kernel space.

IX. Conclusion

In this paper, we provide RContainer, a new secure container architecture on the ARM platform, to protect containers from untrusted OS and to enforce strong isolation among containers. RContainer extracts a trusted tiny mini-OS, which is at the same exception level as the deprivileged OS but has higher privileges, responsible for interfering with the interaction between the container and the deprivileged OS. Furthermore, RContainer introduces a container shim for each container to achieve strong isolation among containers in kernel space. We implemented two prototypes of RContainer and evaluated its security and performance. The results show that RContainer is effective and efficient.

ACKNOWLEDGMENT

This work was partially supported by the Program of Key Laboratory of Network Assessment Technology, the Chinese Academy of Sciences, Program of Beijing Key Laboratory of Network Security and Protection Technology, National Key Research and Development Program of China (No.2021YFB2910109) and National Natural Science Foundation of China (No.62202465). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any finding agencies.

REFERENCES

[1] 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, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*. ACM, 2016, pp. 277–290, doi:10.1145/2872362.2872372.

[2] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. ACM, 2013, pp. 265–278, doi:10.1145/2451116.2451146.

[3] A. V. Hof and J. Nieh, "Blackbox: A container security monitor for protecting containers on untrusted operating systems," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 2022, pp. 683–700, url:osdi22/presentation/vant-hof.

[4] Z. Shen, Z. Sun, G. Sela, E. Bagdasaryan, C. Delimitrou, R. van Renesse, and H. Weatherspoon, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 2019, pp. 121–135, doi:10.1145/3297858.3304016.

[5] Google, "gvisor," https://github.com/google/gvisor, 2023.

[6] A. Ltd, "Arm security technology - building a secure system using trustzone technology," https://www.arm.com/zh-TW/technologies/trustzone-for-cortex-a/tee-reference-documentation, 2009.

[7] Z. Hua, Y. Yu, J. Gu, Y. Xia, H. Chen, and B. Zang, "Tz-container: protecting container from untrusted OS with ARM trustzone," *Sci. China Inf. Sci.*, vol. 64, no. 9, 2021, doi:10.1007/S11432-019-2707-6.

[8] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with ARM trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'17, Niagara Falls, NY, USA, June 19-23, 2017*. ACM, 2017, pp. 488–501, doi:10.1145/3081333.3081349.

[9] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1416–1432, doi:10.1109/SP40000.2020.00061.

[10] Y. Zhang, Y. Hu, Z. Ning, F. Zhang, X. Luo, H. Huang, S. Yan, and Z. He, "SHELTER: extending arm CCA with isolation in user space," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2023, pp. 6257–6274, url:usenixsecurity23/presentation/zhang-yiming.

[11] ARM, "Arm ecosystem fvps," https://developer.arm.com/downloads/-/arm-ecosystem-fvps, 2023.

[12] Firefly, "Rk3399 development board," https://www.t-firefly.com/product/rk3399.html, 2020.

[13] N. Yang, W. Shen, J. Li, Y. Yang, K. Lu, J. Xiao, T. Zhou, C. Qin, W. Yu, J. Ma, and K. Ren, "Demons in the shared kernel: Abstract resource attacks against os-level virtualization," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, 2021, pp. 764–778, doi:10.1145/3460120.3484744.

[14] "Arm cca," https://www.arm.com/ja/architecture/security-features/arm-confidential-compute-architecture, 2023.

[15] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. S. 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, ASPLOS 2015*. ACM, 2015, doi:10.1145/2694344.2694386.

[16] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li, "Deconstructing xen," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017, url:ndss-2017-programme/deconstructing-xen.

[17] Y. Wu, Y. Liu, R. Liu, H. Chen, B. Zang, and H. Guan, "Comprehensive VM protection against untrusted hypervisor through retrofitted AMD memory encryption," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, 2018, pp. 441–453, doi:10.1109/HPCA.2018.00045.

[18] Q. Zhou, X. Jia, S. Zhang, N. Jiang, J. Chen, and W. Zhang, "Secfortress: Securing hypervisor using cross-layer isolation," in *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*, 2022, pp. 212–222, doi:10.1109/IPDPS53621.2022.00029.

[19] S. Checkoway and H. Shacham, "Iago attacks: why the system call API is a bad untrusted RPC interface," in *ASPLOS*, 2013, doi:10.1145/2451116.2451145.

[20] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's escape: Breaking the resource rein of linux control groups," *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, doi:CorpusID:202590117.

[21] M. Lipp, M. Schwarz, and D. Gruss, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018, url:usenixsecurity18/presentation/lipp.

[22] P. Kocher, J. Horn, A. Fogh, , and D. Genkin, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019, doi:10.1109/SP.2019.00002.

[23] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: closing controlled channels with self-paging enclaves," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 2020, pp. 7:1–7:16, doi:10.1145/3342195.3387541.

[24] "Kernel page-table isolation," https://en.wikipedia.org/wiki/Ker\nel$_$page-table$_$isolation, 2018.

[25] "Tboot," https://sourceforge.net/projects/tboot/, 2021.

[26] I. Amazon Web Services, "Aws nitro enclaves user guide," https://docs.aws.amazon.com/enclaves/latest/user/building-eif.html, 2023.

[27] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 287–305, doi:10.1145/3132747.3132782.

[28] "Trusted platform module librarry," https://www.iso.org/standard/66510.html, 2016.

[29] "Contiguous memory allocator," https://lwn.net/Articles/396657/, 2010.

[30] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, "SCONE: secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016, pp. 689–703, url:osdi16/technical-sessions/presentation/arnautov.

[31] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[32] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012, doi:10.1145/2133375.2133377.

[33] "Ssl," https://www.ssl.com/, 2021.

[34] "Ssl," https://www.ssl.com/, 1999.

[35] "Arm system memory management unit architecture specification-smmu architecture version 3," https://developer.arm.com/documentation/ihi0070/latest/, 2023.

[36] "Lmbench," https://sourceforge.net/projects/lmbench/, 2023.

[37] "Linux test project," https://github.com/linux-test-project/ltp/releases/tag/20240129, 2024.

[38] "httpd," https://hub.docker.com/layers/library/httpd/latest/images, 2023.

[39] "Apache benchmarks," https://httpd.apache.org/, 2022.

[40] "Memcached," https://hub.docker.com/_/memcached, 2023.

[41] "Memtier benchmarks," https://github.com/RedisLabs/memtier_benchmark, 2023.

[42] "Mysql," https://hub.docker.com/_/mysql, 2023.

[43] "Sysbench," https://github.com/akopytov/sysbench, 2023.

[44] "Nginx," https://hub.docker.com/_/nginx, 2023.

[45] "Netperf," https://hewlettpackard.github.io/netperf/, 2024.

[46] "Github - aldanial/cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages." https://github.com/AlDanial/cloc, 2024, (Accessed on 04/15/2024).

[47] "busybox:1.36.1-glibc," https://hub.docker.com/_/busybox/, 2024.

[48] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*. USENIX, 2002, pp. 17–31, url:publications/library/proceedings/sec02/wright.html.

[49] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris, "Docker-sec: A fully automated container security enhancement mechanism," in *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 2018, pp. 1561–1564, doi:10.1109/ICDCS.2018.00169.

[50] J. Chen, Z. Feng, J. Wen, B. Liu, and L. Sha, "A container-based dos attack-resilient control framework for real-time UAV systems," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*. IEEE, 2019, pp. 1222–1227, doi:10.23919/DATE.2019.8714888.

[51] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: Making linux security frameworks available to containers," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 2018, pp. 1423–1439, doi:usenixsecurity18/presentation/sun.

[52] 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, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. ACM, 2011, pp. 375–388, doi:10.1145/2046707.2046752.

[53] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *31st IEEE Symposium on Security and Privacy, SP 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 143–158, doi:10.1109/SP.2010.17.

[54] B. Danev, R. J. Masti, G. Karame, and S. Capkun, "Enabling secure vm-vtpm migration in private clouds," in *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*. ACM, 2011, pp. 187–196, doi:10.1145/2076732.2076759.

[55] S. Hosseinzadeh, S. Laurén, and V. Leppänen, "Security in container-based virtualization through vtpm," in *Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC 2016, Shanghai, China, December 6-9, 2016*. ACM, 2016, pp. 214–219, doi:10.1145/2996890.3009903.

[56] V. A. Sartakov, L. Vilanova, and P. R. Pietzuch, "Cubicleos: a library OS with software componentisation for practical isolation," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 2021, pp. 546–558, doi:10.1145/3445814.3446731.

[57] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel SGX," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 2020, pp. 955–970, doi:10.1145/3373376.3378469.

[58] A. Baumann, M. Peinado, and G. C. Hunt, "Shielding applications from an untrusted cloud with haven," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. USENIX Association, 2014, pp. 267–283, url:osdi14/technical-sessions/presentation/baumann.

[59] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016, pp. 533–549, url:osdi16/technical-sessions/presentation/hunt.

[60] C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, 2017, pp. 645–658, url:atc17/technical-sessions/presentation/tsai.

[61] I. Corporation, "Intel software guard extensions programming reference," https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf, 2014.

APPENDIX

TABLE X: Application Workloads for Comparison.

Name	Configuration
Apache	Apache server v2.4.58 [38] handling 100 concurrent requests from remote Apache Bench v2.3 [39], serving 4KB with default `index.html`
Nginx	Nginx v1.25.3 [44] handling requests from remote ApacheBench v2.3 [39], serving 4KB with default `index.html`, 500 requests, and 100 concurrency
Memcached	Memcached v1.6.22 [40] using memtier benchmark v2.0.0 [41] with default configuration, serving 16Byte
MySQL	MySQL v8.2.0 [42] handling requests from sysbench benchmarks v1.0.2 [43], running workload using `otp_read_write` with 5 tables, 1,000 counts per-table, 10 threads, and lasting 30 seconds
Netperf	Netperf 2.7 [45] testing `TCP_STREAM` with lasting 60 seconds
Hackbench	Hackbench in Linux test project-20240129 [37] with 25 groups, 40 pairs of sender and receiver, and 100 datasize

TABLE XI: Detailed Case Study*.

CVE	Detail of the Cause, Exploitation, and Defense Simulation	D	RC
2010-2959	This vulnerability is due to the Linux CAN subsystem (/net/can/bcm.c) not increasing restrictions when processing nframes about TX_SETUP and RX_SETUP, resulting in an integer overflow that allows attackers to gain root privileges and execute arbitrary code. An attacker can leverage this vulnerability to gain the root privilege and overwrite the kernel's memory. In our emulation of a deep attack, we deployed a privileged container with `-- privileged=true` to access the kernel's bcd_op and other containers' pagetable pages (PTPs). As expected, it triggered a GPT fault.	No	Yes
2010-4258	This vulnerability is due to Linux passing the user controlled child_tidptr directly to p->clear_child_tid during copy_process, resulting in put_user writing 0 to any userspace during do_exit. An attacker can leverage this vulnerability to write null pointers to any kernel address through calling set_fs() to set the process upper address as KERNEL_DS and leverage DoS attacks. In our emulation of a deep attack, we deployed a privileged container containing a malicious program that writes a null pointer to a pre-set kernel address. As expected, it triggered a GPT fault.	No	Yes
2013-6441	This vulnerability is due to LXC lxc-sshd template mistakenly opening access permissions in/var/lib/lxc/*/config when mounting /sbin/init, allowing attackers to obtain root through the init script. Attackers can leverage this vulnerability to gain the root privilege and then execute arbitrary code. In our emulation of a deep attack, we deployed a privileged container and accessed other containers' PTPs. As expected, it triggered a GPT fault.	No	Yes
2014-6407	This vulnerability is since when executing the `docker pull` to pull an image, a malicious image can download arbitrary files to any path on the host through symlink or hardlink, leading to remote code execution or privilege escalation. Attackers can leverage this vulnerability to extract malicious files to the host OS and gain the root privilege. In our emulation of a deep attack, we deployed a privileged container and attempted to execute our code to access the kernel's PTPs. As expected, it triggered a GPT fault.	No	Yes
2014-9357	This vulnerability is due to Docker allowing the xz binary command to run in root mode to decompress files when image construction, resulting in malicious images or files being able to obtain root privileges on the host through the `xz` command. An attacker can leverage this vulnerability to extract malicious files to the host OS and gain the root privilege. In our emulation of a deep attack, we deployed a privileged container and attempted to execute our code to access the kernel's pagetable pages. As expected, this operation triggered a GPT fault.	No	Yes
2015-1335	This vulnerability is due to LXC's failure to check for potential bad entries when mounting fstree during container startup, resulting in malicious containers being able to change the mounting target through symbolic links to bypass AppArmor policies and gain higher privileges.An attacker can leverage this vulnerability to bypass the AppArmor policies and mount the target directory to gain the root privilege. In our emulation of a deep attack, we deployed privileged container mounted in the /root/bin directory and attempted to execute `/bin/bash` to access other containers' pagetable pages. As expected, this operation triggered a GPT fault.	No	Yes
2015-3627	This vulnerability is due to libcontainer and Docker allowing any file descriptor to be passed to the pid-1 process before performing chroot, allowing malicious images to carry out directory traversal attacks by creating symbolic links, thereby overwriting host files and gaining higher privileges. An attacker can leverage this vulnerability to overwrite the host files and execute arbitrary code. In our emulation of a deep attack, we deployed a privileged container within a malicious symlink to `/bin/bash` and attempted to access the kernel's PTPs. As expected, it triggered a GPT fault.	No	Yes
2015-3629	This vulnerability is due to improper link parsing by Libcontainer before performing chroot operations during respawning a container, which allows malicious images to carry out directory traversal attacks and elevate permissions by constructing `symlink.s` An attacker can leverage this vulnerability to overwrite the host files and execute arbitrary code. In our emulation of a deep attack, we deployed a privileged container and attempted to access the kernel's pagetable pages. As expected, this operation triggered a GPT fault.	No	Yes
2015-3630	This vulnerability is due to Docker allowing containers to write to /proc/asound, /proc/timer_stats, /proc/latency_stats, and /proc/fs, allowing malicious containers to modify the system's global configuration and obtain sensitive information. Attackers can leverage this vulnerability to change the global configuration and obtain sensitive information. In our emulation of a deep attack, we deployed a privileged container with permission to write to /proc/* and attempted to access other containers' image files. As expected, all images are encrypted.	No	Yes
2016-1576	This vulnerability is due to the lack of strict restrictions on setuid and other attributes when mounting overayFS, which allows attackers to mount an overayFS filesystem on the FUSE and execute arbitrary custom setuid programs.Attackers can leverage this vulnerability to mount an overlayFS filesystem on FUSE to gain higher privileges. In our emulation of a deep attack, we mounted a overlayFS filesystem with setting setuid as 's'(root permission) and attempted to execute our code to access containers' PTPs. As expected, it triggered a GPT fault.	No	Yes
2016-1581	This vulnerability is caused by setting the /var/lib/lxd/zfs.img permission attributes to 644 (readable by anyone) during `lxd init`, and setting the /var/lib/lxd/containers/<container>. zfs and rootfs to 755, allowing anyone to read the contents of any container through the container's name.An attacker can leverage this vulnerability to get the target container's rootfs and obtain the sensitive messages. In our emulation of a deep attack, we deployed a container as the target by configuring its rootfs as 755 and then attempted to access the executable files inside the container from the host OS. As expected, although these files can be accessed, sensitive content cannot be obtained due to encryption.	No	Yes
2016-1582	This vulnerability is due to the fact that when a container switches from non-privileged mode to privileged mode, its directory has a 755 attribute instead of 700 and is still owned by the container uid/gid, causing host users to traverse the directory to find the root setuid path.An attacker can leverage this vulnerability to get the target container's rootfs and obtain the sensitive messages. In our emulation of a deep attack, we deployed a container as the target by configuring its rootfs as 755 and then attempted to access the executable files inside the container from the host OS. As expected, although these files can be accessed, sensitive content cannot be obtained due to encryption.	No	Yes
2016-3697	This vulnerability is due to /libcontainer/user/user.go in runc, which processes the `--user` function by treating the numeric UID as the username, resulting in privilege escalation by modifying /etc/passwd and constructing a specific user id. Attackers can leverage this vulnerability to gain the root and execute arbitrary code. In our emulation, we deployed a privileged container with host root id (the same number as in /etc/passed) as the parameter for `--user` and attempted to access the kernel's PTPs with root permission in the container. As expected, it triggered a GPT fault.	No	Yes
2016-5195	This vulnerability arises from a condition race issue within the __get_user_pages() during Copy-on-Write. When executing the write(), there's a lack of atomic consistency protection for the sequence involving the follow _page_mask() and faultin_page(), allowing the sequence to be interrupted by madvise (MADV_DONTNEED), resulting in the kernel's dirty-writeback writing the page-cache page back to a high-privilege file.An attacker can leverage this vulnerability to gain the host root privilege and execute arbitrary code. In our emulation of a deep attack, we added a user with privileges by modifying the /etc/passwd, deployed a malicious privileged container with this user information, and then attempted to access the kernel's pagetable pages in the container. As expected, this operation triggered a GPT fault.	No	Yes
2016-7117	This vulnerability is due to the fact that when processing the exit path of recvmmsg(), the struct sock pointer may be released anywhere after calling fput_light(). However, when datagrams != 0 &&err != -EAGAIN, the struct sock pointer is used again at sock->sk->sk_err=-err, resulting in a user-after-free and privilege escalation.An attacker can leverage this vulnerability to gain the root privilege and execute arbitrary code. In our emulation of a deep attack, we deployed a malicious privileged container as an attacker who has already obtained root privileges and attempted to execute our code to access other containers' pagetable pages. As expected, this operation triggered a GPT fault.	No	Yes
2016-9962	This vulnerability is due to an improper dumpable state configuration of the runc init process, allowing the pid-1 process of a container to ptrace additional processes started with `runc exec` and obtain the file descriptors of these new processes in the container, resulting in container escalation or runc state modification.An attacker can gain access to host machine files by executing malicious code in the interval between the runc init function and the `execve` function. In our emulation of a deep attack, we deployed a malicious container, accessed host machine files by exploiting relative paths "../../.." through open file descriptors during the runc init process for gaining root privilege, and attempted to execute our code to access the kernel's pagetable pages. As expected, although the container gained the root permission, this operation still triggered a GPT fault.	No	Yes
2017-7308	This vulnerability is due to packet_set_ring() copying int type req_u->req3.tp_sizeof_priv to unsigned short p1->blk_sizeof_priv through init_prb_bdqc() when calculating block size within the kernel code net/packet/af_packet.c, resulting in a hep overflow and out-of-write.An attacker can leverage this vulnerability to gain higher privileges and execute arbitrary code by using malicious input to manipulate the kernel's memory. In our emulation of a deep attack, we deployed a malicious container with allocating the kernel's circular buffer and packet_sock object together by leveraging the packet_sock structure and memory blocks within the ring buffer, overwrited the packet_sock->xmit function pointer to execute commit_creds(prepare_kernel_cred(0)) in container, and then attempted to access the kernel's PTPs. As expected, it triggered a GPT fault.	No	Yes

2017-1000112	This vulnerability is due to UFO allowing for the filling of skb larger than MTU, causing the value of maxfraglen-skb->len to become negative during non-UFP execution, causing skbs to be reallocated and skb_prev->len-maxfraglen to exceed MTU, resulting out-of-bounds write during the skb_copy_and_csum_bits() execution.An attacker can leverage this vulnerability by using skb_copy and_csum_bits() to overwrite the skb, resulting in memory corruption or DoS in the kernel address. In our emulation of a deep attack, we deployed a malicious container, sent a buffer with the MSG_MORE flag set via the UFO path, constructed an ROP chain by transitioning to the non-UFO path to send a buffer of length 1, and attempted to write into the kernel's address space. As expected, this operation triggered a GPT fault.	No	Yes
2018-15514	This vulnerability arises from the HandleRequestAsync() function in Docker for Windows, which deserializes incoming requests via the .\pipe \dockerBackend pipe, introducing potential .NET object deserialization vulnerabilities, resulting in executing arbitrary code on the host OS with administrative permissions.An attacker can leverage this vulnerability to gain administrator privileges on the host OS and execute arbitrary code. In our emulation of a deep attack, we created a test user account within the docker-users group, gained administrator privileges by using the BinaryFormatter class as the formatter, loading malicious code, and deploying it via the TypeConfuseDelegate gadget chain, and finally executed our malicious code to access other containers' pagetable pages. As expected, this operation triggered a GPT fault.	No	Yes
2018-15664	This vulnerability is because during the execution of the `docker cp` command, the docker client does not immediately use the conversion path, but instead sends it to the get/container/ID/archive interface of the daemon. This gap allows attackers to exploit the execution interval from FollowSymlinkInScope() to addTarFile() to insert a malicious symlink, thereby gaining access to arbitrary files on the host.An attacker can leverage this vulnerability to gain root access to arbitrary files on the host or other containers, achieved by substituting the copy path with another symbolic link between parsing the file during the copy process and executing the copy action. In our emulation of a deep attack, we deployed a container and initiated a continuous process that swapped the '/safe_path' file with a symbolic link to a malicious binary to access the other containers' pagetable pages. We then continuously executed the `docker cp` command to simulate the scenario of copying files from the container to the host machine. As expected, although the content of '/safe_path' has been modified, the execution of this binary still triggered a GPT fault.	No	Yes
2018-18955	This vulnerability arises from the mishandling of nested user namespaces in the map_write() function when the number of UID or GID is more than 5, resulting in unrestricted kernel to namespace mapping, which allows attackers to gain root privileges by creating a new user namespace using a non-privileged user.An attacker can leverage this vulnerability to gain unauthorized access by injecting malicious code with incorrect UID values and gain the root privilege. In our emulation of a deep attack, we deployed a privileged container that has the permission to execute `/bin/bash` and attempted to access other containers' pagetable pages. As expected, this operation triggered a GPT fault.	No	Yes
2019-5736	This vulnerability is due to Docker's mishandling of file descriptor /proc/self/exec, which allows attackers to overwrite the runc binary by executing the docker `exec` command, enabling arbitrary code execution on the host OS. Attackers can leverage this vulnerability to overwrite the `/bin/bash` in the host OS, and then execute their code with root privilege. In our emulation of a deep attack, we first deployed a normal container, used `docker run exec` to write /bash/bin to the container, and then tried to access the kernel's PTPs. As expected, it triggered a GPT fault.	No	Yes
2019-10144	This vulnerability arises from the improper isolation of processes within containers executed through the `rkt enter` command. It allows malicious processes to have all capabilities when the applications are running within the container, and then to gain unauthorized access to host resources. Attackers can leverage this vulnerability to gain unauthorized root access to the host system by exploiting the `rkt enter` to execute a binary. In our emulation of a deep attack, we deployed a malicious container and used the `rkt enter` command to execute a malicious binary to access the kernel's PTPs within a container with elevated privileges, bypassing the cgroup isolation. As expected, this operation triggered a GPT fault.	No	Yes
2019-14271	This vulnerability stems from docker-tar mistakenly loading the libnss dynamic library from the container's filesystem instead of the host's filesystem when executing the `docker cp` command. This allows malicious containers to bypass the isolation between the container and the host system by constructing the libnss_files.so libraryAn attacker can leverage this vulnerability to execute arbitrary code with root privileges on the host system by crafting a Docker container that includes a malicious NSS library. In our emulation of a deep attack, we added a function to the source code of libnss_files.so.2 for accessing other containers' pagetable page and defined the execution function, the newly compiled libnss_files.so.2 was sent to the container, triggering malicious code. As expected, this operation triggered a GPT fault.	No	Yes
2020-8835	This vulnerability arises from the incorrect calculation of value ranges in the BPF_REG_STATE register by the BPF verifier during the verification of the BPF_JMP32 instruction, leading to out-of-bounds access.An attacker can leverage this vulnerability to locate the init_pid_ns struct, obtain the task_struct structure of the target container using find_task_by_pid_ns, and ultimately access the memory of any container. In our emulation of a deep attack, we attempted to modify the pagetable pages based on mm_struct pointer in the task_struct when the attacker had gained access to the target container's task_struct. As expected, this operation triggered a GPT fault in RContainer.	No	Yes
2020-14386	This vulnerability is caused by the tpacket_rcv function, which converts and assigns unsigned int type tp_reserve and unsigned short type netoff when calling skd_network_offset to calculate the header offset, resulting in integer overflow Attackers can leverage this vulnerability to control the netoff variable in tp_reserve(), thereby further controlling the macoff variable and achieving out-of-bounds kernel heap write when calling virtio_net_hdr_from_skb to allocate a ring buffer. In our emulation of a deep attack, we deployed a privileged container to write to the kernel's memory. As expected, this action triggered a GPT fault.	No	Yes
2021-28972	This vulnerability is due to memory overflow caused by the incorrect handling of the variable '\0' by add_slot_store and remove_slot_store in RPA PCI Hotplug driver when the user writes drc_name data from userspace to the kernel, allowing the user to control drc_name to overwrite the kernel stack and heap.An attacker can leverage this vulnerability to write arbitrary data to the kernel stack, and corrupt the kernel memory. In our emulation of a deep attack, we modified the add_slot_store() to incorrectly handle drc_name, and then deployed a malicious container to construct a malicious device name to achieve kernel memory corruption. As excepted, this action triggered a GPT fault.	No	Yes
2021-32606	This vulnerability arises from the system's ability to modify the socket option to CAN_ISOTP_SF_BROADCAST through isotp_setsockopt after socket binding. As a result, it becomes impossible to unregister the CAN receiver, bypassing isotp_release and leading to potential UAF (Use-After-Free) issues when messages are sent to the previous socket by other sockets.An attacker can leverage this vulnerability to overwrite the function pointer by spraying struct isotp_sock, and then achieve arbitrary kernel execution. In our emulation of a deep attack, we deployed a privileged container that has a root shell, and then tried to execute code to traverse pagetable pages of other containers. As expected, it triggered a GPT fault.	No	Yes
2022-23222	This vulnerability arises from the incomplete inclusion of *OR_NULL pointer types by the eBPF verifier when disabling arithmetic addition and subtraction operations for certain pointer types in adjust_ptr_min_max_vals(). This allows attackers to obtain PTR_TO_MEM_OR_NULL type pointers through BPF_FUNC_ringbuf_reserve(), further leaking the kernel address.An attacker can leverage this vulnerability to overwrite stack pointers with crafted data by utilizing the bpf_skb_load_bytes_ * series of functions, and further obtain leaked address information. In our emulation of a deep attack, we deployed a privileged container which has obtained permission to access host filesystem like /sys/fs/bpf, and further accessed the target process cred using ebpf program. As expected, this operation triggered a GPT fault in RContainer.	No	Yes
2024-21626	This vulnerability is caused by runc not closing the file descriptor of /sys/fs/cgroup on time before calling setcwd(2). This allows malicious containers to access the host filesystem by setting the working directory to the symbolic link /proc/self/fd/7 during `docker run` or `docker exec`, and further overwrite binary files by tracing back to `/bin/bash` through relative paths. One possible method of exploiting this attack involves the utilization of /proc/self/fd/7/../../../bin/bash as a means for the attacker to overwrite binary files, such as `/bin/bash`, by employing it as a binary parameter for "process.args". This manipulation allows the attacker to execute arbitrary code on the host. In our emulation of a deep attack, we deployed a privileged container, where we wrote a bash script to attempt to traverse and access the pagetables of other containers. As expected, this operation triggered a GPT fault in RContainer.	No	Yes

* **RC** means RContainer; **D** means native Docker
* **Yes** means that the CVE can be mitigated by the system; **No** means that the CVE cannot be mitigated by the system.