TwinVisor: Hardware-isolated Confidential Virtual Machines for ARM

Dingji Li[†§‡], Zeyu Mi[†‡], Yubin Xia[†‡], Binyu Zang[†‡], Haibo Chen[†‡], Haibing Guan[◇]
[†]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*
[§]*MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University*
[‡]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*
[◇]*Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University*

Abstract

Confidential VM, which offers an isolated execution environment for cloud tenants with limited trust in the cloud provider, has recently been deployed in major clouds such as AWS and Azure. However, while ARM has become increasingly popular in cloud data centers, existing confidential VM designs mainly leverage specialized x86 hardware extensions (e.g., AMD SEV and Intel TDX) to isolate VMs upon a shared hypervisor.

This paper proposes TwinVisor, the first system that enables the hardware-enforced isolation of confidential VMs on ARM platforms. TwinVisor takes advantage of the mature ARM TrustZone to run two isolated hypervisors, one in the secure world (called *S-visor* in this paper) and the other in the normal world (called *N-visor*), to support normal VMs and confidential VMs respectively. Instead of building a new *S-visor* from scratch, our design decouples protection from resource management, and reuses most functionalities of a full-fledged *N-visor* to minimize the size of *S-visor*. We have built two prototypes of TwinVisor: one on an official ARM simulator with S-EL2 enabled to validate functional correctness and the other on an ARM development board to evaluate performance. The *S-visor* comprises 5.8K LoCs while the *N-visor* introduces 906 LoC changes to KVM. According to our evaluation, TwinVisor can run unmodified VM images as confidential VMs while incurring less than 5% performance overhead for various real-world workloads on SMP VMs.

CCS Concepts: • **Security and privacy → Virtualization and security**.

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483554

Keywords: Cloud Computing, Virtualization, Confidential Computing, ARM TrustZone

1 Introduction

Confidential computing, which leverages hardware to provide attestable isolated execution environments, is getting prevalent on cloud platforms [78]. Sensitive data is isolated in such environments, making it invisible to all (including the cloud provider) but the authorized data processing code. There are different levels of confidential computing, including application-level, container level, and virtual machine (VM) level [40, 45, 80, 82]. Among these different types, hardware-based VM-level confidential computing has recently gained traction due to its compatibility with existing IaaS (Infrastructure as a Service) clouds, minimal intrusiveness to tenants' workloads, and clear security boundary easy to be implemented and enforced.

Major cloud vendors, such as Google Cloud [17], Microsoft Azure [31] and IBM Cloud [29], are now offering confidential VMs as a service to end users based on AMD's Secure Encrypted Virtualization (SEV) [1, 2]. In addition, Intel will include Trust Domain Extensions (TDX) [24, 25] in its future CPUs and IBM published Protected Execution Facility (PEF) [54] in the latest POWER9 chips to support confidential VMs.

With the rise of cloud computing, the ARM platform has become popular in data centers [12, 22] due to its rich ecosystem and excellent price-performance ratio [52, 55, 56, 73]. As a result, it is natural for security-sensitive ARM users to concern about when and how ARM servers will support confidential VMs. ARM recently announced a new ARMv9 extension called Confidential Compute Architecture (CCA) [7] to enable confidential VMs. However, its hardware will not be available for another year or two according to ARM's roadmap [4]. Further, it is still unclear how to design and implement systems atop CCA.

On the other hand, ARM has a mature hardware extension named TrustZone [74] (since 2003), which is capable of partitioning a server into two isolated worlds (a *normal world* and a *secure world*). Therefore, TrustZone has been

widely used on mobile platforms to provide Trusted Execution Environment (TEE). Furthermore, TrustZone has recently (since ARMv8.4) introduced hardware virtualization with the *secure EL2 extension* (called S-EL2 in our paper) to efficiently running VMs in the secure world. This leads to one core question: ***is it possible to support confidential VMs on ARM servers by retrofitting mature hardware features like TrustZone with new software designs?***

One straightforward direction is to deploy critical VMs in the secure world with a dedicated hypervisor. We argue that it is inadequate to rebuild or port a full-fledged hypervisor in the secure world. This argument is based on the evolution of commercial hypervisors and TrustZone TEE-Kernels: both were born with a small Trusted Computing Base (TCB) and high-security guarantee but eventually evolved into software entities with numerous security vulnerabilities and large attack surfaces [37, 43, 61, 69, 71, 84].

In this paper, we propose TwinVisor, the first system that enables hardware-isolated confidential VMs for ARM servers. A key observation is that mature and extensively tested hypervisors, such as KVM/ARM [26, 46, 47] and Xen-ARM [38, 41], already exist in the normal world. Therefore, TwinVisor disentangles the management of confidential VMs from the protection mechanisms. Specifically, TwinVisor reuses the existing hypervisor in the normal world (called *N-visor* in this paper) to manage hardware resources and serve both secure VMs (S-VM) and normal VMs (N-VM). In terms of protection, TwinVisor creates a tiny hypervisor in the secure world (called *S-visor*) to focus on protecting S-VMs. One major benefit of TwinVisor is that the S-visor's TCB keeps small and reliable despite the contiguous evolution of the N-visor's functionalities. While enjoying the convenience brought by the N-visor, every S-VM is isolated from the entire normal world as well as other S-VMs in the secure world.

Nevertheless, existing TrustZone hardware was originally designed to run two hypervisors with ***independent*** privilege models, ***static*** resource partitions, and ***infrequent*** inter-world communications, posing three major challenges for TwinVisor. First, TrustZone's secure world is no more privileged than the normal world, making it impossible for the S-visor to transparently intercept the execution of the N-visor like the trap-and-emulate method [75]. Second, traditional TrustZone applications have fixed resource requirements, so that TrustZone adopts a static policy of resource partition. For example, TrustZone only allows designating a limited number of physical memory regions as secure memory. However, the static resource partition results in insufficient resources or low resource utilization for S-VMs, whose use of hardware resources is highly dynamic. Third, the conventional TrustZone usage model is based on the assumption that world switches are not frequent, so that a large switch overhead has little impact on overall performance [53]. But TwinVisor's design requires close collaboration between two hypervisors in both worlds, leading to frequent world switches that cause much runtime overhead for S-VMs.

We address the above challenges through three key designs [1]. To tackle the first challenge, we propose horizontal trap (H-Trap, § 4.1), a trap-and-emulate like mechanism that enables the S-visor to check the N-visor's operations with minor modifications. For the second challenge, we design split contiguous memory allocator (split CMA, § 4.2), with which the normal and secure worlds collaborate to resize secure memory regions dynamically. Lastly, we use a fast switch facility (§ 4.3) that avoids redundant operations to boost world switches.

We have implemented two prototypes of TwinVisor for functional validation and performance evaluation on two ARM platforms (an official simulator and a hardware SoC). The code size of the S-visor is about 5.8K LoCs, and we also slightly modify the Linux kernel v4.14 (906 LoCs). The performance evaluation results on microbenchmarks and various real-world applications show that TwinVisor incurs less than 5% performance overhead for SMP S-VMs. Both prototypes demonstrate that the N-visor can manage hardware resources and schedule all N-VMs and S-VMs while the S-visor protects unmodified S-VMs transparently. Besides software designs, we also discuss possible hardware improvements for future ARM architectures to support confidential VMs better (§ 8). A prototype of TwinVisor is open-sourced at https://github.com/TwinVisor.

Contributions. The contributions of the paper are:

- We design and implement TwinVisor, the first hypervisor architecture that supports confidential VMs on ARM with a small TCB.
- We introduce a set of software designs to securely and efficiently protect S-VMs based on current ARM features and propose advice to improve future hardware features.
- We evaluate not only the functional correctness of TwinVisor on an official emulator but also its runtime performance by using functionally similar hardware.
- We provide a reference design for future systems with similar architectures (e.g., ARM CCA), which may assist them in addressing similar challenges.

2 Background

This section first compares state-of-the-art solutions for confidential computing, and then introduces background knowledge about ARM TrustZone, S-EL2 extension and ARM CCA.

[1] CCA's software design may have similar challenges faced by TwinVisor since both CCA and TrustZone require a dual-hypervisor system architecture. CCA can address these challenges with TwinVisor's techniques.

	Basic Info.				Security Metrics			
Name	Arch	Domain Type	Domain Num	Software Shim	Reg Prot	Secure Mem	Mem Size	Mem Granu
Intel SGX	x86	Process	Unlimited	✗	✓	Static	128/256MB	Page
Intel Scalable SGX	x86	Process	Unlimited	✗	✓	Static	1TB	Page
AMD SEV	x86	VM	16/256	✗	✗	Dynamic	All	Page
AMD SEV-ES/SNP	x86	VM	Limited	✗	✓	Dynamic	All	Page
Intel TDX	x86	VM	Limited	✗	✓	Dynamic	All	Page
Power9 PEF	Power	VM	Unlimited	✓	✓	Static	All	Region
Komodo	ARM	Process	Unlimited	✓	✓	Dynamic	All	Region
ARM S-EL2	ARM	VM	Unlimited	✓	✗	Dynamic	All	Region
ARM CCA	ARM	VM	Unlimited	✓	✓	Dynamic	All	Page
TwinVisor	ARM	VM	Unlimited	✓	✓	Dynamic	All	Page

Table 1. A comparison of different solutions for confidential computing. In the "Secure Mem" column, "Static" means the physical range of the secure memory can only be reserved statically at boot time, while "Dynamic" means the solution can adjust the range of the secure memory dynamically at runtime.

2.1 Existing Solutions for Confidential Computing

As shown in Table 1, various solutions have been released to guard cloud tenants [1, 13, 14, 17, 23, 25].

Intel SGX [23, 40, 59, 72] and Komodo [49] focus on providing enclaves for user-mode applications. Intel SGX defends against both malicious software and hardware attacks by utilizing hardware memory encryption and integrity enforcement, but its secure memory size is restricted to 128/256MB. A scalable version of SGX [21, 35] was released recently, increasing the secure memory size to 1TB, but it weakens the integrity protection at hardware level. Komodo relies on the isolation of ARM TrustZone, which supports unlimited secure memory size but has no hardware guarantee of encryption and integrity, succumbing to physical attacks.

Different from the process-level enclave, AMD SEV-SNP [1, 2, 33], Intel TDX [25], Power PEF [54], and ARM CCA aim at enabling confidential VMs with specialized hardware extensions. SEV-SNP, TDX and CCA benefit from hardware memory encryption and integrity protection. To enforce strong confidentiality, SEV-SNP and TDX assign a unique encryption key to each different VM, resulting in the limited number of available VM instances [24]. By contrast, CCA only uses global encryption keys to support an unlimited number of VMs. SEV deploys a security co-processor with its dedicated memory in the CPU to take care of VMs, whereas TDX employs a microcode-implemented hardware shim called TDX module to monitor the interactions between the VM and the untrusted hypervisor and enforce security policies. In comparison, PEF and CCA choose a software shim, which has better flexibility and is easier to be updated and verified [49].

2.2 ARM TrustZone

As shown in Figure 1, ARM TrustZone divides a processor state into normal world and secure world, each of which has multiple exception levels (ELs): EL0 for applications, EL1 for kernels, EL2 (if exists) for hypervisors. The normal-world software, including the hypervisor, is untrusted and cannot access the secure-world resources, while the secure-world software may access all resources. The normal-world hypervisor requests a TA's service by invoking *Secure Monitor Call (SMC)*, a special hardware instruction, to trap into the secure monitor in EL3, which further forwards the control flow to TEE-Kernel in S-EL1. Returning from the TEE-Kernel to the hypervisor is a reverse procedure. The world switch is usually infrequent, so it is unlikely to affect the runtime performance [53].

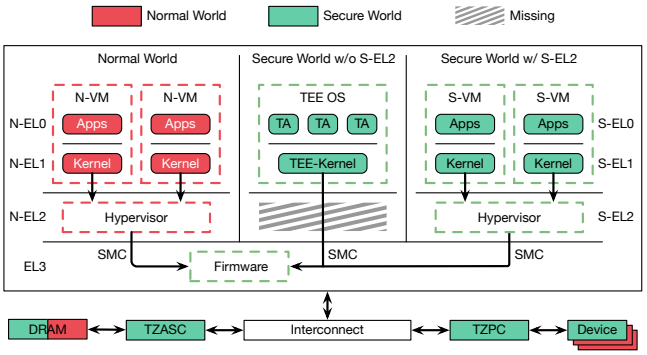

Figure 1. The architectures of ARM TrustZone with and without the S-EL2 extension. The middle secure world is the traditional secure world that misses the secure EL2 level while the right secure world owns the S-EL2 level since ARMv8.4. Please note that the middle and the right secure world cannot co-exist.

All **physical** memory pages in ARM TrustZone can be categorized into secure memory and non-secure memory [2]. The two types of memory are isolated from each other by ARM TrustZone Address Space Controller (TZASC). TZASC throws a page fault exception for each memory access if the security states of the current CPU and the physical page mismatch. The latest implementation of TZASC (TZC-400) [8] supports configuring the security attributes for up to *eight* different memory regions. For each memory region, the address range is defined by a base address register and a top address register, while its accessibility to the two worlds is described by a region attribute register. Only privileged and

[2]We use the terms "non-secure memory" and "normal memory" interchangeably in this paper.

trusted software (e.g., secure monitor, TEE-Kernel) can configure these registers.

In TrustZone, a peripheral device belongs to either secure world or normal world. The normal world cannot access a secure-world device. TrustZone also divides interrupts into two worlds. A secure interrupt has to be handled by the TEE-Kernel. The designation of interrupts can be configured by modifying Generic Interrupt Controller (GIC) registers.

2.3 Secure EL2 (S-EL2) Extension

Since ARMv8.4, the S-EL2 enables the hardware virtualization feature in secure world, as shown on the right in Figure 1. S-EL2 mirrors almost all aspects of N-EL2 in the secure world to support a hypervisor. For example, the register VSTTBR_EL2 stores the base address of a secure stage-2 page table (S2PT) in S-EL2, while its counterpart in N-EL2 is VTTBR_EL2. When an S-VM accesses a secure Intermediate Physical Address (IPA), this address will be translated through the secure S2PT pointed by VSTTBR_EL2.

2.4 ARM CCA

CCA is a planned security extension for ARMv9 that introduces Realm [5], which is another isolated world with virtualization support. Granule Protection Table (GPT) [6] is introduced as a new third-stage page table to determine the accessibility of each physical page. A processor in Realm and secure states cannot touch the memory of the other, but both states have access to non-secure memory. Besides, CCA encrypts the memory of Realm and the secure world with separate keys and enforces integrity protection by hardware. However, hardware support alone is insufficient, which further requires hypervisor-level software for VM management. Such a dual-hypervisor system may encounter the same challenges faced by this paper. Given that CCA's unavailability will last for some time and the cancellation of its predecessor Bowmore [86], existing ARM cloud tenants require a feasible confidential VM solution before CCA is fully ready.

3 Overview

3.1 Design Goals and Architecture

TwinVisor targets to open the secure world to the normal software without bloating the TCB. It provides cloud tenants multiple hardware-isolated confidential VMs (we also call them secure VMs or S-VMs in our paper), which run unmodified operating systems.

The concrete goals of TwinVisor are listed as follows:

- **G1: Security:** S-VMs are protected from illegal accesses by untrusted software. The TCB of TwinVisor should keep small so that it can be verified easily.
- **G2: Efficiency:** The performance of workloads running in S-VMs is comparable to that of VMs in a system without TwinVisor. It should also be easy to scale to multiple virtual CPUs (vCPUs) and S-VMs with small overhead.
- **G3: Minimal Modification:** TwinVisor introduces minor modifications to existing software.

Figure 2 is an overview of TwinVisor's architecture, in which two hypervisors run in two worlds. The N-visor in N-EL2 manages hardware resources for both S-VMs and N-VMs to consolidate VMs, while the S-visor in S-EL2 guards S-VMs [3]. The N-visor in N-EL2 manages hardware resources and provides services for S-VMs while the S-visor in S-EL2 guards them. When creating an S-VM, the N-visor allocates hardware resources including CPU time slices, physical memory, and I/O devices. When the N-visor decides to run an S-VM, it hands over the control flow to the S-visor, which installs environment and executes code of the target S-VM.

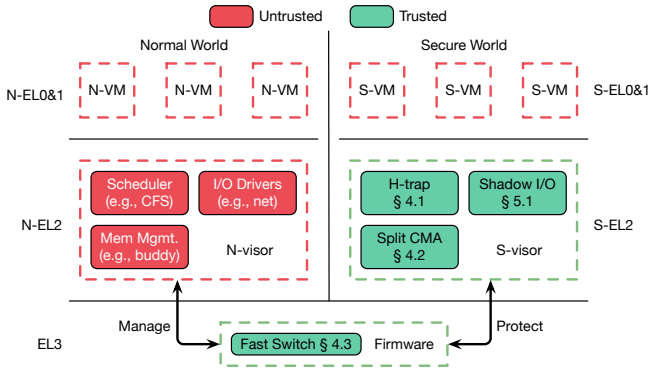

Figure 2. The overall architecture of TwinVisor. Green blocks are the TCB of TwinVisor.

For CPU time slices, a scheduler in the N-visor schedules all S-VMs and N-VMs, whereas the S-visor neither includes a scheduler nor reserves physical cores for S-VMs to keep its TCB small. If a time slice expires and a periodic timer interrupt fires when an S-VM is running, the S-VM traps into the S-visor, which then returns to the N-visor to invoke scheduling. For physical memory, the N-visor dynamically allocates normal memory for an S-VM. These memory pages are transformed into secure memory and then mapped into the S-VM's IPA space by the S-visor. For I/O devices, the N-visor manages physical devices (such as storage and network devices) and provides para-virtualization (PV) I/O devices for S-VMs, whose I/O data is transparently copied between the two worlds by the S-visor [4]. S-VMs should protect their own I/O data through encryption and integrity checking.

An S-VM is protected against any illegal accesses by untrusted software, which includes anything except the secure monitor in EL3 and the S-visor in S-EL2. Illegal accesses are malicious read and write operations targeting an S-VM's

[3] An S-VM is not a virtual TrustZone of an N-VM [53], so it can only provide services for VMs via the network.

[4] For the secure devices directly connected to the secure world, TwinVisor can support them by running a TEE-Kernel in an S-VM like the traditional I/O model of TrustZone, which is outside the scope of this work.

CPU registers, memory pages, and corresponding hypervisor data structures (e.g., S2PT). First, the S-visor carefully hides all CPU registers' values from the N-visor to prevent data leakage and arbitrary writes. Second, an S-VM's memory pages are of secure memory, and thus inaccessible to the N-visor. An illegal physical memory access will trigger a page fault waking up the secure monitor and finally notifying the S-visor. Moreover, an S-VM cannot touch other S-VMs' memory due to the IPA isolation enforced by different S2PTs. Third, the N-visor cannot access sensitive data structures like S2PTs of S-VMs, which also reside in the secure memory.

3.2 Threat Model and Assumptions

The TCB of TwinVisor includes the thin S-visor and the trusted firmware that cannot be changed arbitrarily since it requires the vendor's signature during secure boot. TwinVisor assumes that the firmware and the S-visor are loaded securely by the secure boot of TrustZone, and the implementation of ARM TrustZone and TZASC conforms to their specifications.

Attackers' capabilities: An attacker can control an N-VM to compromise the N-visor via exploiting its software vulnerabilities. Therefore, all these software running in the normal world is untrusted, including the N-visor and all N-VMs. A breached N-visor may try to read and tamper CPU registers, memory contents, I/O data of an S-VM and the S-visor. Rogue devices can issue malicious DMA to access S-VM's memory, which can be defeated by configuring SMMU page tables. TwinVisor assumes that the software in S-VMs will not voluntarily reveal their sensitive data and protects their I/O data by using encrypted message channels like SSL [28] and full disk encryption [30]. Though an S-VM is possibly controlled by a malicious tenant and sends its own secret out, TwinVisor enforces isolation to ensure that a malicious S-VM cannot access any secret data of other S-VMs and the S-visor.

Out of scope: An attacker does not have physical access to the machine so she cannot mount physical attacks via offline DRAM analysis (e.g., cold-boot attacks [83]) and intercepting communications between CPU and memory. We argue that because software attacks happen far more frequently than hardware attacks in today's clouds, it is reasonable to focus on defending against software attacks [13]. Besides, TwinVisor can benefit from such hardware supports when new hardware is available. For example, ARM CCA plans to enable memory encryption for the secure memory of TrustZone [6]. Protecting an S-VM against Denial-of-Service (DoS) is not a security objective of TwinVisor. Side-channel attacks including cache-based side-channel attacks are also out of scope. Their defense methods are orthogonal to the design of TwinVisor. For example, existing mechanisms [57, 68, 79] can be applied to TwinVisor to protect S-VMs from cache side-channel attacks, and speculation barrier instructions [3] can be used to mitigate speculative execution attacks.

Attestation: TwinVisor also assumes that there exists a hardware-backed root of trust to support remote attestation. Before sending sensitive data to S-VMs, cloud tenants ask their applications in S-VMs to attest the firmware, the S-visor and kernel images [5] through the chain of trust. Measurements of the firmware and the S-visor should be verified by the hardware vendors that deploy them. Similar attestation approaches have already been widely supported [36, 49, 76], which can be applied to TwinVisor.

4 Detailed Design

4.1 Logical Deprivileging Model

An ideal hardware capability of S-EL2 is that it resides at a higher privilege than N-EL2 so that the S-visor can trap sensitive instructions invoked by the N-visor, like the nested virtualization model [42, 65, 69, 84]. This capability allows the S-visor to transparently monitor the N-visor's behaviors without modifying it, which satisfies **G3**. But the reality is that N-EL2 and S-EL2 are essentially two **independent** privilege levels, and the N-visor can directly control hardware behaviors without triggering any traps, which makes it hard for the S-visor to know when the N-visor needs to run an S-VM and what updates are made by it. For instance, when the N-visor executes an *ERET* instruction to enter one S-VM, the very first instruction executed by the vCPU will trigger a synchronous external exception to crash the system.

A possible solution is using the PV model [41] to modify the N-visor to proactively use APIs implemented by the S-visor, which includes replacing all sensitive instructions in the N-visor with **SMC** instructions and designing special data structures in shared memory pages to synchronize information (e.g., page table updates and I/O data) with the S-visor. But this method not only causes numerous world switches, but also leads to excessive modifications to the N-visor, which violates **G2** and **G3**.

To address this issue, we propose a technique called **horizontal trap (H-Trap)**, which slightly modifies the N-visor to logically deprivilege the N-visor. A key observation is that any hypervisor or VM configurations cannot affect the S-VM's execution until the S-visor starts to run this VM, logically making the N-visor less privileged than the S-visor. Therefore, all checks on these configurations can be batched until the S-visor enters the S-VM, significantly reducing world switches. Moreover, the H-Trap does not provide any shared PV data structures for the two hypervisors to communicate with. Instead, it reuses the existing hardware interface including CPU registers and S2PTs in the normal

[5] Since major cloud providers allow users to import custom kernel images [11, 18, 32], cloud tenants can upload and verify their own trusted kernel images.

world to avoid excessively modifying the N-visor. The S-visor checks CPU registers and memory mappings in place and blocks illegal states before entering the S-VM.

TwinVisor provides a call gate that includes an *SMC* instruction to help the N-visor to proactively enter the S-visor. ERET is the only sensitive instruction that we replace in the N-visor. Today's commercial hypervisors are usually well-structured, and it is highly likely that the N-visor uses ERET to resume VMs in only a few locations. For example, we found there are only two such locations in KVM. Moreover, the replacement of ERETs with call gates is only for functional considerations (to allow the N-visor to run S-VMs). A malicious N-visor may try to run an S-VM in the normal world using an arbitrary ERET. Since the S-VM is located in the secure memory, the execution after this ERET will be intercepted by TZASC and finally reported to the S-visor. Therefore, even though there exist remaining ERETs that are not replaced with call gates, they will not pose any security threats to TwinVisor.

VM and System Registers. When handling an H-Trap, the S-visor checks what updates the N-visor makes to VM and system registers. Therefore, any register values cannot affect an S-VM's execution until the S-visor checks them and resumes this S-VM. The H-Trap mechanism combines different trapping strategies for various types of registers. First, all VM registers are shared between the two worlds, including system (in EL1) and general-purpose ones. To prevent the N-visor from reading or writing register values, the S-visor saves all VM register values into its secure memory and randomizes general-purpose register values before redirecting a VM exit to the N-visor. However, sometimes the N-visor does need to access an S-VM's register values (e.g., for emulating devices). To this end, the S-visor selectively exposes necessary register values to the N-visor. Specifically, each time a register of the S-VM is chosen for passing parameters to the N-visor. The index of the register to be exposed can be decoded from ESR_EL2 by the S-visor, while other registers do not need to be exposed to the N-visor. Second, for the normal hypervisor registers like VTCR_EL2 and HCR_EL2 that control the virtualization behaviors of an S-VM, TwinVisor still allows them to be used freely by the N-visor. The S-visor validates these registers before resuming an S-VM.

Shadow S2PT. TwinVisor guarantees that the N-visor cannot directly read or write the secure S2PTs of any S-VMs. Similar to previous work [42, 65], the S-visor uses secure memory to build a shadow S2PT, the base address of which is stored in VSTTBR_EL2 register, for each S-VM. To reuse the N-visor to handle stage-2 page faults normally, the N-visor is still able to modify the normal S2PT of each S-VM. But a normal S2PT does not affect an S-VM's memory translation, it only conveys what mapping updates the N-visor wishes to perform. The S-visor checks and synchronizes mapping updates to the shadow S2PT, which is the actual S2PT that controls the S-VM's memory translation. Specifically, when a stage-2 page fault occurs, the S-visor intercepts it, records the fault address (IPA), and forwards it to the N-visor. The N-visor then calls its page fault handler to directly modify the normal S2PT, which is checked by the S-visor before resuming the S-VM's execution. The S-visor maintains a page mapping table (PMT) for each S-VM to record which physical memory pages this S-VM owns. The PMT can be used to prevent the N-visor from maliciously mapping one physical page to multiple S-VMs, and to guarantee no memory leakage will occur.

4.2 Cooperative Management of Memory Resources

The traditional TrustZone use cases assume that all hardware resources are statically partitioned to the two worlds at boot time. So the TrustZone hardware (TZASC) is unable to support dynamically changing the security states of physical memory at page granularity. It can only manage the memory security by using at most eight memory regions. This is incompatible with the scenario in which an S-VM's memory may be dispersed. Therefore, if S-VMs' memory pages are not consecutive, the S-visor will quickly run out of memory regions.

The second problem is that the N-visor is unaware of the dynamic adjustment of memory security. If the S-visor dynamically changes some memory pages for an S-VM from non-secure to secure at runtime, the physical memory allocator unaware of this attribute change may still allocate them to other kernel components, which will encounter a hardware fault due to the access to these pages. One naive solution is to statically divide the entire physical memory space into two parts (secure and non-secure) and notify the two hypervisors of the address ranges free to use. Although this solution avoids the hardware fault, the static partition prevents dynamic physical memory adjustment and affects overall memory utilization. When the S-visor exhausts its own memory, it is unable to serve new S-VMs. On the other hand, even if the N-visor is hunger for memory, overall memory utilization remains low if the S-visor does not fully utilize its secure memory regions.

Split Contiguous Memory Allocator (Split CMA). Fortunately, we observe that today's OS kernels are usually equipped with a special type of physical memory allocators that can allocate consecutive physical pages at a large scale. They were designed to support I/O devices working on large consecutive physical memory regions (e.g., hardware video decoders). Linux CMA (Contiguous Memory Allocator) [16] is a typical example, which reserves large regions of consecutive physical memory early at boot time. The reserved memory is then returned to the buddy allocator to serve normal memory allocation requests. If CMA memory cannot satisfy an allocation request, it makes room by migrating

pages that have been allocated by the buddy allocator to other locations.

Therefore, we propose a split contiguous memory allocator (split CMA) that reuses the existing CMA to address the above two problems. The first problem can be resolved by the consecutiveness of CMA-managed memory. The split CMA tries to keep secure memory physically consecutive so that TZASC regions are sufficient to cover every secure memory page. For the second problem, the split CMA is divided into two modules (a normal end and a secure end) in two worlds, each of which cooperates with the other to dynamically switch the memory security attribute. If the S-visor and S-VMs do not occupy the reserved memory, the split CMA normal end gives them to the buddy allocator for normal memory allocation. When the S-visor asks for more memory, the normal end will recycle or migrate memory from the buddy allocator and allocate them to the secure end. The secure end will, in turn, compact and return memory to the normal end when the N-visor is hungry for memory. Therefore, this dynamic memory adjustment improves overall memory utilization.

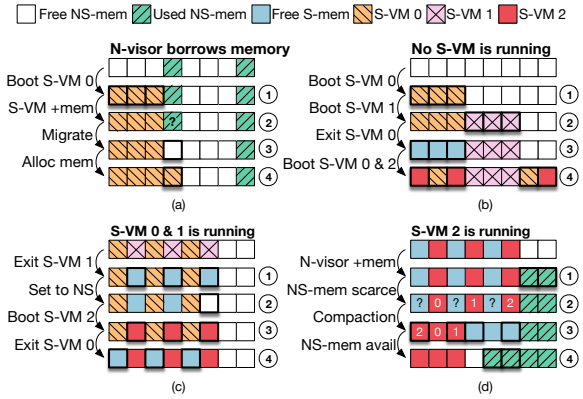

Figure 3. Four examples of the secure memory allocation in the split CMA. For simplicity, we just show eight secure memory chunks in the pool.

Memory Organization. The split CMA designs a hierarchical structure of memory organization. At the top level, the split CMA arranges all available memory in a memory pool. Though only one memory page is needed by an S-VM in each stage-2 page fault, it is unwise to allocate memory from the pool at page granularity, because the lock contention of the pool can lead to severe performance degradation in the multi-VM scenario. Therefore, the split CMA increases the allocation granularity to memory chunks. At the middle level, a memory pool is divided into fixed-size memory chunks, each of which consists of multiple pages and belongs to one S-VM exclusively. The address of every memory chunk is aligned to the chunk size. For example, the chunk size in our implementation is 8MB and its address is aligned to 8MB. At the bottom level, a memory chunk is utilized as a cache of memory pages and maintains a bitmap to record which pages are free. An S-VM obtains memory from its local cache of pages and requests a new one if the old one is used up. The split CMA marks a memory cache as inactive if exhausted, active otherwise. When booting Linux, the split CMA normal end reserves a contiguous range of physical memory for the memory pool.

According to our experience using TZASC, though it supports up to eight memory regions, only four regions are available to use for S-VMs since the other four have been occupied by the S-visor (e.g., the S-visor will reserve a region for its own secure memory). The split CMA utilizes the rest 4 regions and implements 4 memory pools so that an allocation request failing in one pool (such as temporarily busy pages) can be redirected to other pools.

Handling a Stage-2 Page Fault. When handling a stage-2 page fault, the N-visor calls its own page fault handler that has been slightly modified to use the split CMA normal end for page allocation. If there is no active memory cache for this specific S-VM, the normal end assigns a new cache with the lowest physical address in the pool to the S-VM. The N-visor then gets a page from the cache and maps it into this S-VM's IPA space by configuring the normal S2PT whose base address is stored in VTTBR_EL2 register. At this moment, the allocated physical page still belongs to the normal memory. Once the N-visor invokes the call gate to pass the control flow to the secure world, the secure end walks the normal S2PT using the recorded IPA and gets the mapped HPA value. After validating the S-VM's PMT table, the S-visor synchronizes this mapping to the corresponding shadow S2PT (the really used one) before the S-visor resumes this S-VM.

This page table walk can be boosted by just checking the page table pages that translate the fault IPA. So there are at most four pages needed to be read. The secure end finds the memory chunk the mapped HPA belongs to by masking out the lower bits and validates whether the chunk's owner VM is this S-VM. The secure end then dynamically changes the whole memory chunk to the secure memory. Any future allocation requests served by this chunk will not need to change the memory security. Afterwards, the N-visor cannot access the mapped memory page anymore since an access will be detected by TZASC, which generates a synchronous external exception to wake up the trusted firmware and notify the S-visor.

Figure 3(a) is a typical example of memory allocation. Two chunks have already been allocated to the buddy allocator in the N-visor initially. Suppose the N-visor needs to boot S-VM 0 that needs 3 memory chunks, the three memory chunks at the head of the pool are enough to boot this VM. But when S-VM 0 needs more memory at runtime, the normal end has to migrate the first normal memory chunk away and then allocates this chunk to this VM.

When an S-VM shuts down, the N-visor notifies the S-visor to release this S-VM's memory, during which the secure end clears all related pages. But the S-visor keeps these memory chunks as secure for other S-VMs and lazily returns them to the N-visor if needed. An S-VM shutdown is depicted in Figure 3(b). When S-VM 0 is closed in step-2, the secure end zeros its memory contents and keeps the released memory as secure, allowing subsequent S-VMs to reuse this memory without changing its security.

Memory Compaction. If the normal end uses up the reserved memory, it will borrow secure memory from the secure end. However, while the secure end may have sufficient free memory chunks at times, these secure memory chunks may be nonconsecutive and thus unable to be returned to the normal world. As shown in Figure 3(c), two S-VMs use a memory pool chunk by chunk. If S-VM 0 exits, the secure end is only able to return the memory chunk at the end of secure memory range. Even worse, all released memory after closing S-VM 0 in step-4 cannot be returned to the normal world despite that they are not used by any S-VMs. To address this issue, the split CMA supports compacting the nonconsecutive secure memory chunks in the pool and returns the compacted memory to the normal world. Figure 3(d) is an example of chunk migration, the secure end will try to compact the memory chunks of S-VM 2 by migrating them to the head of the pool and then replenish the non-secure memory for the N-visor.

The pages being migrated could have been mapped to an S-VM. The secure end reconfigures its shadow S2PT to mark these pages as non-present and then moves these pages' contents to new locations. If an S-VM accesses the migrated pages while the migration is in progress, the S-VM will encounter a stage-2 page fault and get trapped into the S-visor, which pauses the S-VM and resumes it when the migration is complete. This migration will have no effect on S-VMs that do not run or access the migrated pages.

4.3 Efficient World Switch

Each time the N-visor enters or exits an S-VM, a context switch between the two worlds happens, which is a frequent operation. The world switches between the two hypervisors have to involve the trusted firmware in EL3 to change the NS bit in SCR_EL3 register [6]. This long path increases the latency of VM exit handling, which has already been deemed highly related to the overhead of virtualization [70, 85]. We find that a world switch contains redundant operations like repeatedly saving and restoring the register values of a vCPU when entering and exiting EL3 and S-EL2, which incurs large overhead as shown in our evaluation (§ 7.2). To mitigate the performance issue, we design a **fast switch** facility that reduces the latency of world switch by 37.4% (see § 7.2). While previous work [66, 77, 81] used a combination of side core polling and shared memory to avoid context switches, this method wastes CPU resources. Furthermore, it is difficult to determine the appropriate number of cores dedicated to polling.

Shared Pages for General-purpose Registers. We use a shared page on each physical core to transfer vCPU general-purpose register values between two hypervisors. Before invoking the SMC instruction, the N-visor stores all vCPU register values into a shared page. The trusted firmware will not save or restore any register values into and from stacks. It just changes the NS bit and installs necessary states of the S-visor. The S-visor directly reads values from the shared page and writes these values into corresponding registers. However, using shared pages to transfer states easily leads to a TOCTTOU attack. Specifically, after the S-visor checks the register values in the shared page, a malicious N-visor can concurrently modify the shared page in another physical core. One straightforward solution is to disable access to the shared page after switching to the secure world, which effectively prevents the N-visor from modifying this page. But this solution requires frequent modifications to TZASC regions to change the security attribute of the shared page, which further increases the latency of fast switch. And the limited number of TZASC regions discourages us to spare a separate region for this page. TwinVisor defends against this attack in a check-after-load way [50] by reading register values before checking them.

Register Inheritance for System Registers. TwinVisor leverages register inheritance to further avoid redundant operations that save and restore EL1 and some EL2 registers. First, since both hypervisors work in EL2 and they do not need to use any EL1 system registers, all these registers set up by the N-visor for a guest vCPU will not be touched by the firmware and directly inherited by the S-visor, which checks these registers in place. Second, the two hypervisors have two different sets of hypervisor control registers in EL2 (e.g., VTTBR_EL2 and VSTTBR_EL2), which means these registers can also be directly passed between two worlds without being touched by the firmware.

5 Implementation

5.1 I/O Virtualization

The S-visor fully reuses the I/O mechanism and device drivers of the N-visor. In the current stage, TwinVisor takes the PV model to enable I/O supports for S-VMs. Before the S-VM is ready for PV I/O, we need to safely load the kernel into the memory of the S-VM in case of malicious kernel code modifications. To keep the TCB small and minimize modifications, we reuse the kernel loading logic of the N-visor and store an S-VM's kernel image without encryption in the normal world (separated from the encrypted disk image in which sensitive data is saved). After an S-VM starts

[6] SCR_EL3 is only accessible in EL3 and will trigger an exception if accessed in lower exception levels.

to boot, the kernel image is loaded into the memory within a fixed GPA range. Before the S-visor synchronizes a mapping into the shadow S2PT, it will check the integrity of the page if the GPA falls into the range of the kernel image.

Shadow PV I/O. TwinVisor uses shadow I/O rings and shadow DMA buffers to be transparent to S-VMs and reuse existing PV I/O code without modification. In the current implementation of Linux, both I/O rings and DMA buffers are allocated from the secure memory of S-VMs, which is inaccessible to the N-visor. Therefore, the S-visor duplicates I/O rings and DMA buffers in the normal memory for the N-visor, and synchronizes I/O requests and DMA data between two worlds for shadowing. Take the shadow I/O ring for example, when a frontend driver in an S-VM issues an I/O request, the S-visor copies the request from the S-VM (secure) to the I/O ring (non-secure). Conversely, if the backend driver in the N-visor raises an I/O completion interrupt, the S-visor synchronizes the I/O ring (non-secure) to the shadow one (secure) and redirects the interrupt to the S-VM.

However, the shadow I/O ring will bring non-negligible overhead for applications, especially network-intensive ones that involve large numbers of small network packets. Frequent network packets require low-latency I/O ring synchronization to achieve good performance. In the vanilla case, the frontend and the backend drivers use shared memory for the I/O ring so that they know the I/O handling progress of each other in real time, reducing the frequency of interrupts. But in TwinVisor, the shadow I/O ring is not shared memory between the two drivers and the synchronization brings a large time window during which the two drivers are unaware of the I/O progress. So they have to send more interrupt notifications to synchronize shadow I/O ring, leading to a larger runtime overhead. We optimize this by leveraging routine VM exits caused by WFx instructions and physical IRQs to piggyback the updates of the TX shadow I/O ring. These VM exits trap into the S-visor, which then synchronizes the shadow I/O ring in time. The piggyback technique greatly improves network-intensive applications. For example, the normalized overhead of Memcached in a 4-vCPU S-VM drops from 22.46% to 3.38%.

5.2 Prototypes for Evaluations

At the time of writing, there is still no commercially available hardware supporting ARMv8.4-A (Secure-EL2).

Functional Evaluation. We validate the design of the TwinVisor by implementing a prototype on ARM Fixed Virtual Platform (FVP) [9] with full-featured S-EL2 enabled, which is an official full-system ARM simulator and has been used by previous work [48] for testing functional correctness. This prototype helps confirm that TwinVisor will function well on future commercial hardware.

Performance Evaluation. To evaluate the performance of TwinVisor, we use a development board with a Hisilicon Kirin 990 [20]) SoC, which supports N-EL2 and Virtualization Host Extension (VHE) [27], working similarly to the hardware with S-EL2 enabled. The device manufacturer also provides the source code of the firmware in EL3 and their internal Linux. We move the S-visor from the secure world to the normal world, which means both the N-visor and the S-visor now run in N-EL2. All S-VMs thus run in N-EL1 and N-EL0. To manage S-VMs, the N-visor still gets trapped into the firmware in EL3, which then forwards the control flow to the S-visor in N-EL2. Since the S-visor is unable to control TZASC registers in N-EL2, we emulate all the TZASC-related operations by delaying for different time periods, whose lengths are equal to those of the same operations we have measured in the secure world on this device.

	Lines of Code
S-visor	5.8K
TF-A	1.9K (w/o S-EL2) / 163 (w/ S-EL2)
Linux	906
QEMU	70

Table 2. The code size of the prototype system of TwinVisor.

5.3 Implementation Complexity

We run *cloc* [15] tool to measure the code size of TwinVisor. The code size of the S-visor is around 5.8K LoCs (4.7K lines of C and 1.1K lines of assembly), which is much smaller than existing TEE kernels (e.g., Linaro TEE with 110K LoC). The small code size makes formal verification feasible, just as SeKVM (3.8K LOC) [63], CertiKOS (9K LOC) [51], and seL4 (10K LOC) [58], which would be our future work. We use KVM in Linux kernel v4.14 to be the N-visor: 90 LoCs of existing files are modified and 816 LoCs of new files are added to implement the fast switch (130 LoCs) and the split CMA (686 LoCs). For the trusted firmware, we have to add much code (1.9K LoCs) in the Trusted Firmware-A (TF-A) v1.5 on the Kirin 990 to support the S-visor in N-EL2 for performance measurement. Most of the added code is for the fast switch between the N-visor and the S-visor in the N-EL2. If the hardware supports S-EL2, we just need to add 163 LoCs (94 lines of C, 69 lines of assembly) to support the S-EL2 bootstrap and the fast switch. Finally, we add less than 70 LoCs to QEMU v4.2.0 to support the shadow PV I/O.

6 Security Analysis and Evaluation

6.1 Security Analysis

This section enumerates six properties of TwinVisor and shows how these properties protect S-VMs.

Property 1: The firmware and the S-visor are trusted during the system's lifetime. Device vendors provide the secure boot of TrustZone to guarantee the integrity of the firmware and the S-visor during bootstrapping. When the system is booted, remote attackers can come from the normal world or S-VMs. Attackers from the normal world cannot access the firmware and the S-visor since they are isolated by TrustZone and secure memory. As for malicious

S-VMs in the S-EL1, the S-visor residing in the more privileged S-EL2 leverages a separate address space to defend itself. Besides, the S-visor prevents attackers from maliciously changing hardware behaviors by the horizontal trap (§ 4.1).

Property 2: The integrity of S-VMs' kernel images is enforced by the S-visor. An S-VM's kernel image is loaded into the memory by the untrusted N-visor when the S-VM starts to boot. Based on Property 1, before a kernel page takes effect, the trusted S-visor first turns the page into secure memory and then checks its integrity. Therefore, a malicious N-visor cannot touch the kernel image after loading it into the memory, and the S-visor can ensure only the verified kernel takes effect.

Property 3: Each S-VM's CPU register states are protected by the S-visor. Based on Property 1, the trusted S-visor protects S-VMs' CPU register states in two ways. First, it saves registers before entering the N-visor and compares the saved values with new ones after returning back. As a result, the N-visor is unable to hijack the control flow of S-VMs by tampering registers such as link registers (LR, ELR) and page table base registers (TTBR). Second, it hides the general-purpose registers by randomizing them. Thus, the N-visor cannot obtain sensitive data through S-VMs' registers.

Property 4: Each S-VM's memory is isolated from other S-VMs and the normal world. Based on Property 1, isolations between S-VMs are enforced by the shadow S2PTs maintained by the trusted S-visor. The normal world has no access to S-VMs' memory and shadow S2PTs in the secure memory. A compromised N-visor may try to leak S-VMs' data by mapping their pages to a malicious S-VM. However, the S-visor keeps track of the ownership of physical pages. Before synchronizing mappings to shadow S2PTs (§ 4.1), the S-visor verifies the page ownership and ensures that no two S-VMs share a page. Besides, the S-visor can leverage ARM SMMU [10, 67] to defeat DMA attacks.

Property 5: Each S-VM's I/O data is protected by the S-visor. TwinVisor assumes that S-VMs utilize end-to-end encryption and integrity checking to protect their I/O data. If any plaintext I/O data and encryption keys stay in the S-VM's CPU registers or memory pages, they cannot be leaked to the N-visor due to Property 3 and 4. The shadow I/O mechanism (§ 5.1) further ensures that any data copied to the normal memory has already been encrypted and thus cannot reveal any sensitive information. The integrity checking performed by S-VMs ensures that malicious code or data pages constructed by attackers will not be used.

Property 6: All data and the control flow of each S-VM are protected by the S-visor. Based on Property 1, the firmware and the S-visor cannot be compromised by remote attackers. Combined with Properties 3 and 4, attackers are unable to access S-VMs' CPU register states or memory even if they compromise the N-visor or collude with a malicious S-VM. Along with Property 5, all data and the control flow of S-VMs are protected from remote attackers.

6.2 Security Evaluation against CVEs

To verify the security of TwinVisor in real-world scenarios, we analyze KVM CVEs relevant to our threat model and apply them to TwinVisor's architecture. Table 3 lists representative CVEs with security threats in the last five years. The CVEs listed primarily aim to gain complete control of, execute arbitrary code in, and disclose sensitive information from the N-visor in order to compromise the data of VMs. Initially, remote attackers can either connect to the system through VMs, or login into the N-visor as unprivileged users. By leveraging vulnerabilities (e.g., buffer overflow, use-after-free), attackers are able to breach the normal execution of the N-visor and access the data of S-VMs.

Type	CVEs
Privilege Escalation	CVE-2019-6974, CVE-2019-14821, CVE-2018-10901
Remote Code Execution	CVE-2020-3993, CVE-2018-18021
Information Disclosure	CVE-2021-22543, CVE-2020-36313, CVE-2019-7222, CVE-2017-17741

Table 3. Representative KVM CVEs in recent five years.

However, as TwinVisor inherently distrusts the N-visor, none of the above attacks can threaten S-VMs. We also simulate three attacks assuming that the N-visor has been controlled by remote attackers. First, the N-visor mapped a secure memory page of the S-visor in its own page table and tried to read the content of this page. An exception triggered by TZASC was taken to the trusted firmware and reported to the S-visor. Then, the N-visor tried to corrupt the PC register value of an S-VM. The S-visor detected the abnormal value by comparing it with the previously stored one. Next, the N-visor mapped a secure memory page belonging to an S-VM in the non-secure S2PT of another S-VM, attempting to synchronize this page into the latter's secure S2PT. As expected, the S-visor detected and rejected this attempt.

7 Performance Evaluation

7.1 Experimental Setup

We run experiments on a Kirin 990 development board with 2 Cortex-A76 big cores (2.86 GHz), 2 Cortex-A76 middle cores (2.36 GHz) and 4 Cortex-A55 small cores (1.95 GHz). All these cores are ARMv8.2. To prevent the performance instability caused by the asymmetric multi-core architecture, we only enable 4 Cortex-A55 cores in our evaluation. The development board has 8GB RAM and 256GB ROM. For network related benchmarks, we build a local area network (LAN) between the device and an x86-64 PC by USB tethering through a USB Type-C connection. The PC is equipped with a 6-core Intel i7-8700 CPU and 32GB RAM. The servers of these benchmarks are placed in on-device VMs and the clients are in a Ubuntu-18.04 VM on PC.

The firmware in EL3 is TF-A v1.5. The N-visor is a Linux kernel 4.14 that is slightly modified to support the Kirin 990. It uses QEMU v4.2.0 to manage all N-VMs and S-VMs, each of which runs Linux kernel 4.15 and contains an 8GB disk image file.

Operation	Vanilla	TwinVisor	Overhead
Hypercall	3,258	5,644	73.24%
Stage2 #PF	13,249	18,383	38.75%
Virtual IPI	8,254	13,102	58.74%

Table 4. A comparison of various architectural operations between TwinVisor and Vanilla. (Unit: cycles)

7.2 Microbenchmarks

First, we run microbenchmarks to quantify the slowdown of several frequently-used hypervisor primitives, including the round trip of hypercall, stage-2 page fault handling and virtual IPI sending. We leverage PMCCNTR_EL0 to measure CPU cycles. Table 4 shows the average costs of these operations in TwinVisor and vanilla QEMU/KVM (hereinafter called Vanilla). Note that Vanilla runs VMs in the normal world without bothering EL3.

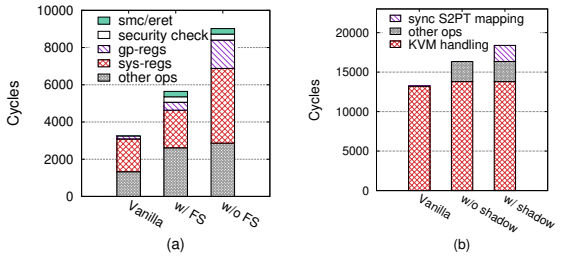

Figure 4. Breakdown comparisons between TwinVisor and Vanilla on hypercall and stage-2 page fault handling microbenchmarks. In (a), "w/ FS" and "w/o FS" on X-axis mean the fast switch is enabled and disabled, "smc/eret" represents SMC/ERET instructions, "gp-regs" and "sys-regs" mean state saving/restoring operations for general-purpose registers and EL1/EL2 registers, and "sec-check" stands for security check. In (b), "w/o shadow" and "w/ shadow" on X-axis mean shadow S2PT is disabled and enabled, "sync" stands for synchronization of shadow S2PT.

In the hypercall microbenchmark, an S-VM issues a *null* hypercall that directly returns without doing anything. The experiment is repeated 1 million times in a uniprocessor S-VM pinned to one physical core and we use the average cycle count. TwinVisor introduces about 73% performance overhead compared with Vanilla. We further break down the time costs of null hypercalls with and without *fast switch*. As shown in Figure 4(a), the hypercalls with and without fast switch costs 5,644 and 9,018 cycles respectively. The fast switch accelerates the world switch because of two major factors: the shared page technique eliminates operations that redundantly save and restore general-purpose registers of the S-VM (1,089 cycles) and the register inheritance avoids the operations of saving and restoring EL1/EL2 registers (1,998 cycles). Saving and restoring general-purpose registers cost a lot if fast switch is disabled because there are 4 memory copies on the roundtrip path of a hypercall. Each copy involves 31 registers and costs more than 62 load/store operations (e.g., registers spilled to the stack). Therefore, 4 redundant memory copies and some unnecessary savings and restorings waste more than 300 load/store operations which cost 1,089 cycles in total.

For stage-2 page fault handling, we repeat reading four bytes from an unmapped page in an S-VM for 1 million times and calculate their average cycles. A read operation will trigger a stage-2 page fault that is handled by the N-visor. We measure the time duration before reading this page and after getting the content. The overhead is about 39% that mainly comes from the synchronization of shadow S2PT (2,043 cycles) and other operations in the trusted firmware and the S-visor (2,358 cycles). We also measure the cost of handling stage-2 page faults with shadow S2PT disabled, which means that the original S2PT prepared by the N-visor is directly used for the S-VM for performance comparison. Figure 4(b) shows that TwinVisor without shadow S2PT saves 2,043 cycles because it no longer needs to validate and synchronize the modifications to shadow S2PT.

For virtual IPI sending, we measure the cycles of sending an IPI from a vCPU that will invoke an empty function on the other vCPU and wait until the function returns. The average overhead is about 59%.

Name	Description
Memcached	Memcached v1.6.7 running the memaslap benchmark v1.0 with a default 128 concurrency on the remote client to test transactions per second.
Apache	Apache Web server v2.4.34 using the ApacheBench v2.3 with 80 concurrency on the remote client to test the number of handled requests per second serving the index page.
Hackbench	Hackbench using Unix domain sockets and default 10 process groups running in 100 loops, measuring the time cost.
Untar	Untar extracting the 5.8.13 Linux kernel tarball using the standard tar utility, measuring the time cost.
Curl	Curl v7.72.0 downloading a 10MB image from the Apache Web server to the remote client, measuring the time cost.
MySQL	MySQL v5.7.32 running sysbench v0.4.12 with 2 threads concurrently on the remote client to measure the time cost by an oltp test, the size of table is 1,000,000 and the test mode is complex mode.
FileIO	Fileio test in sysbench v0.4.12 with threads equal to the number of vCPUs concurrently and 1GB file size in random read-/write mode.
Kbuild	Compilation of the Linux 5.8.13 kernel using allnoconfig for aarch64 with gcc v7.3.0.

Table 5. Descriptions of application benchmarks

7.3 Real-world Applications Performance

We further run real-world applications to evaluate the performance of TwinVisor (with shadow S2PT and fast switch enabled) compared with Vanilla. Table 5 shows the applications we use and they cover CPU-intensive, memory-intensive and I/O-intensive situations. TLS/SSL connection is enabled in Curl and MySQL benchmarks, and we find that whether to enable TLS/SSL has little effect on their performance. Though ApacheBench also supports TLS/SSL, we still disable it because the throughput of Apache drops more than 92% (e.g., TPS drops from 1109 to 86 in a UP N-VM) if

using TLS/SSL, which cannot reflect the real performance of the application. We measure the performance of S-VMs with 1, 4, 8 vCPUs and 512MB memory. The 8-vCPU stands for the CPU oversubscription scenario.

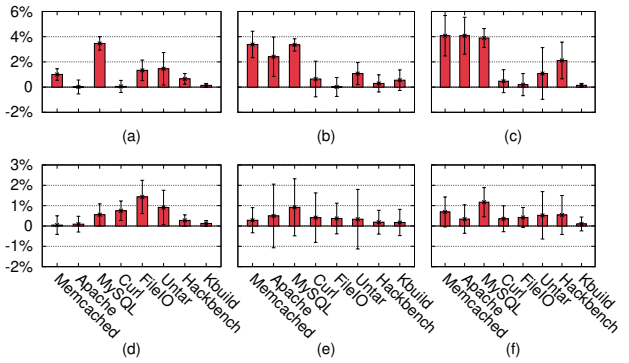

Figure 5. Normalized performance of an S-VM and an N-VM in TwinVisor. (a), (b) and (c) show a UP S-VM, a 4-vCPU S-VM and an 8-vCPU S-VM while (d), (e) and (f) show a UP, a 4-vCPU and an 8-vCPU N-VM. The Y-axis is the normalized overhead of an application compared with that running in a Vanilla VM. Note: The Y-axis ranges of (a), (b), (c) and (d), (e), (f) are different. Absolute values of [UP, 4-vCPU, 8-vCPU] S-VMs are as follow: Memcached in TPS [4897.2, 17044.2, 16853.6], Apache in RPS [1109.8, 2949.7, 2605.6], MySQL in number of events [4165.6, 5222.4, 5095.6], Curl in seconds [0.345, 0.350, 0.342], FileIO in MB/s [29.2, 52.4, 48.6], Untar in seconds [280.574, 279.555, 282.587], Hackbench in seconds [1.694, 0.754, 1.709], Kbuild in seconds [619.725, 162.978, 194.839].

As shown in Figure 5, the performance gap between TwinVisor and Vanilla is less than 5% in all benchmarks. This is because world switches and shadow operations occupy merely a small proportion of applications' effective execution time. Non-WFx exits, whose time cost directly affects applications' performance, only make up a small part of total time. As TwinVisor's overhead mainly comes from world switches during non-WFx exits, the overhead in application benchmarks is lower than the microbenchmark. Besides, the increased time cost of non-WFx exits leads to fewer WFx exits, which trades idle time for execution time in the S-VM and has just a little impact on application's overall performance. We first take Memcached as an example. The Memcached in the UP S-VM whose average TPS is 4,897 incurs 1.0% overhead compared with Vanilla. There are totally 133K VM exits during the test in TwinVisor while the counterpart in Vanilla has 162K VM exits. The difference between the number of VM exits mainly comes from WFx VM exits which happen when the vCPU is idle. According to the previous microbenchmarks, TwinVisor spends more time on world switching and thus has less idle time. As WFx VM exits take up over 70% CPU usage, the time cost by S-visor interceptions including shadow I/O operations occupies less than 2% CPU usage. The storage-intensive FileIO is another example, which is susceptible to extra memory copies due to shadow I/O operations. The FileIO in the UP S-VM averagely achieves 29.24 MB/s bandwidth and incurs a miniscule overhead of 1.33%. This is because the shadow I/O ring operations just occupy 0.21% CPU usage and the shadow DMA buffers operations merely occupy 2.81% CPU usage. If we disable the shadow I/O operations, the overhead of the FileIO benchmark drops to 0.

According to our analysis, the worst case can be an application that repeatedly invokes hypercalls to the hypervisor and then returns immediately at a high frequency. The overhead of this case should be at the same level as the microbenchmark because the time cost of non-WFx VM exits becomes its bottleneck. But it is unlikely to find such an application in the real world.

Performance Impact on N-VMs: To check whether TwinVisor will affect the performance of N-VMs, we repeat above application benchmarks in N-VMs with 1, 4, 8 vCPUs and 512MB memory. Figure 5(d), Figure 5(e) and Figure 5(f) show that the N-VM in TwinVisor achieves less than 1.5% overhead compared with Vanilla. The slowdown is caused by the code added to the N-visor, which includes vCPU identifications (belongs to S-VM or N-VM) and integrating split CMA into the N-visor.

7.4 Scalability

Scaling vCPU number: To show TwinVisor performs well enough with the growth of vCPU number, we evaluate Memcached in the S-VM with 1, 2, 4, 8 vCPUs and 512MB memory. The result is shown in Figure 6(a). In comparison to Vanilla, the overhead of TwinVisor is less than 5% no matter how many vCPUs are given. As mentioned in § 7.3, VM exits caused by WFx for I/O operations in Memcached dominate the CPU usage. In the multi-vCPU cases, we find that WFx VM exits always occupy more than 70% CPU usage and the interceptions of all VM exits occupy less than 4%.

Scaling Memory: To show TwinVisor is scalable with the growth of the size of memory, we run Memcached benchmark in a 4-vCPU S-VM with 128MB, 256MB, 512MB and 1024MB memory and assign half of the S-VM's memory to the Memcached application. The result is shown in Figure 6(b). In comparison to Vanilla, the overhead of TwinVisor keeps below 5%, because the memory accesses of an S-VM in TwinVisor have no difference from an N-VM in Vanilla after mappings are established in S2PTs. Thus, TwinVisor introduces no overhead as the size of memory grows.

Scaling S-VM number: We first run Memcached, Apache, FileIO and Kbuild in 4 UP S-VMs and run them concurrently to show TwinVisor is scalable when multiple S-VMs run a mixed workload. To avoid interference, all S-VMs are pinned to different cores (2 S-VMs pinned to 1 core in the case of 8 S-VMs) and have 256MB memory. Figure 6(c) shows that the maximum overhead of all benchmarks in the mixed workload is less than 6%. We also run the same benchmark in different numbers of UP S-VMs to show TwinVisor is scalable

when the number of S-VMs grows. As shown in Figure 6(d), Figure 6(e) and Figure 6(f), the average performance of I/O-intensive and CPU-intensive workloads achieves less than 4% overhead compared with Vanilla. The low overhead can be explained by the fact that VM exits account for merely a small percentage out of the total time cost. Take Kbuild as an example, there are 1.5M VM exits throughout the compiling according to our measurement, occupying only 2.86% CPU usage. Moreover, TwinVisor introduces no resource contention since each S-VM has its own data structures.

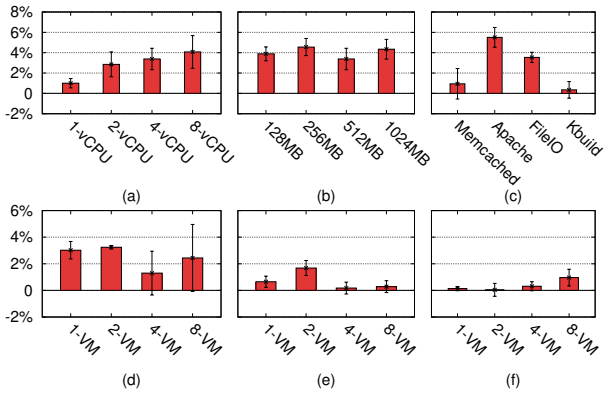

Figure 6. Comparisons of scalability between TwinVisor and Vanilla. (a) and (b) show Memcached with the increasing number (size) of vCPUs and memory in an S-VM. (c) shows Memcached, Apache, FileIO and Kbuild in 4 UP S-VMs respectively. (d), (e) and (f) show average performance of the FileIO, Hackbench and Kbuild with the increasing number of S-VMs. The X-axis is the number (size) of vCPUs/memory/S-VMs, and the Y-axis is the normalized overhead compared with Vanilla. Note: The Y-axis ranges of (a), (b), (c) and (d), (e), (f) are different. Absolute values are as follow: (a) Memcached in TPS [4897.2, 12783.8, 17044.2, 16853.6], (b) Memcached in TPS [16944.4, 17059.0, 17044.2, 17319.2], (c) mixed performance [3927.4 TPS, 960.4 RPS, 26.5 MB/s, 692.13 s], (d) FileIO in MB/s [29.2, 24.8, 16.6, 14.4], (e) Hackbench in seconds [1.694, 2.304, 3.120, 4.478], (f) Kbuild in seconds [619.752, 642.819, 766.98, 1851.796].

7.5 Overhead of Split CMA

We also evaluate the overhead of allocation and compaction operations of split CMA. For allocation operations, we first measure the time of allocating a 4KB page with an active memory cache and the average cost is 722 cycles. For a 4KB page without an active cache, the split CMA has to get a new cache before allocation. When the memory pressure of the N-visor is low, getting a memory chunk unlikely needs to migrate pages. But producing an 8MB cache (i.e., 2048 pages) still averagely costs 874K cycles. The high cost is because an allocation from CMA requires multiple steps, such as locking pages to be allocated and updating memory pages in the bitmap. However, an S-VM needs a new 8MB allocation only when it uses up the previous 2048 pages, so that the frequency of cache allocations is very low. When memory pressure is high, the split CMA must migrate pages away to make room. We leverage *stress-ng* [34] tool to stress the memory on the N-visor and then measure the latency of 8MB chunk allocation. The average cost is 25M cycles (i.e., 13K cycles per page). The same operation under high memory pressure costs 6K cycles per page in Vanilla.

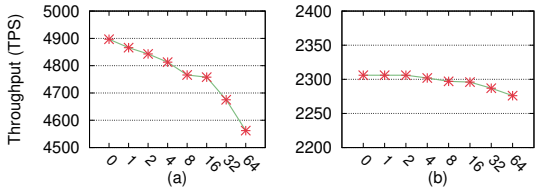

Figure 7. The impact of the number of migrated caches on Memcached using Memaslap benchmark. (a) shows the throughput of a UP S-VM with 512MB memory, while (b) shows the average throughput of 8 UP S-VMs with 256MB memory. The X-axis stands for different numbers of migrated caches in one compaction test. Note: the Y-axis starts from 4500 in (a) and 2200 in (b).

Compaction of an 8MB cache costs 24M cycles on average, including migrating pages of an S-VM and returning the free memory chunk to the N-visor. Moreover, we leverage Memcached to evaluate the performance impact of memory compactions. We first set up an S-VM with 1 vCPU and 512MB memory, allocating 450MB to the Memcached application and leaving the rest for kernel and other system services to make them runnable. We reserve nonconsecutive 512MB memory in the secure-world memory pool, which will be compacted to a consecutive region and then returned to the N-visor. We add a helper function in the N-visor to ask for a specific number of caches and trigger compactions in the secure world. The compactions are triggered at random times during the experiment. Figure 7(a) shows the performance as the number of compacted memory caches grows exponentially from 1 (8MB) to 64 (512MB). The throughput of Memcached drops by 6.84% in the worst case when all 512MB caches are migrated. To further test multiple S-VMs scenarios, we repeat the test with 8 UP S-VMs with 256MB memory. As shown in Figure 7(b), the average throughput of an S-VM only drops by 1.30% in the worst case. The overhead decreases because the overhead of compaction is amortized by multiple S-VMs.

8 Hardware Advice for Future ARM

Selective Transparent Instruction Trapping. To support the S-visor to monitor the N-visor in a transparent way, we propose to add a hypervisor register to future ARM hardware that is accessible only from S-EL2 and EL3. This register controls selective traps of sensitive instructions. Each bit in the register configures whether an instruction executed in N-EL2 should trigger a synchronized exception to S-EL2. For example, setting the bit controlling ERET to "1" means that if an ERET is executed in N-EL2, an exception is taken

to S-EL2. The proposed hardware extension places S-EL2 at a higher privilege level than N-EL2, facilitating the S-visor to supervise the N-visor's behaviors transparently. CCA can also benefit from this extension to transparently monitor the normal world's behaviors from Realm, which can avoid intrusively modifying the N-visor.

Fine-grained Secure Memory Configuration. The complex design of Split CMA can be optimized by making fine-grained changes to the security attribute of memory pages. Though CCA introduces GPT to control memory accessibility at page granularity, the third-stage translation adds multiple page table walks for a memory access if the TLB misses. Given the current TLB reach has been low in data centers [39], GPT may bring non-trivial memory access overhead. Therefore, we propose to slightly extend the TZASC with a bitmap to indicate page security to reduce the overhead of multiple GPT accesses. Each bit in the bitmap represents one physical memory page. For example, a "0" means that the page can be accessed by both worlds while an "1" allows the page to be accessed by the secure world exclusively. For each memory access, TZASC refers to the bitmap according to the page's HPA and decides whether this access is legal by the bit value. Unlike GPT that must be controlled in EL3, the bitmap can be configured by the S-visor in S-EL2 to reduce the EL3-involved overhead. The memory consumption of the bitmap is small: a bitmap of 256GB physical memory consumes only 8MB (0.003% of total). Besides, reading the bitmap results in an additional memory access that increases the memory access latency. Buffering the bitmap entries in CPU caches can boost the bitmap lookup.

Direct World Switch. We also propose that the hardware supports direct world switches between N-EL2 and S-EL2. According to our microbenchmark, the overhead of TwinVisor mainly comes from the costly world switches through EL3. If the hardware supports directly switching between N-EL2 and S-EL2, the overhead will be reduced a lot due to the elimination of processing in EL3. This extension can further reduce the overhead of fast switch. A trap/return-like mechanism is needed for the software to directly switch between two worlds, which benefits the transparent instruction trapping as well since any exception triggered in N-EL2 can be taken to S-EL2 without bothering EL3. A new S-EL2 vector base address register is also necessary, which holds the handler base address for exceptions taken to S-EL2. With this extension, CCA can accelerate world switches between Realm and the normal world as well.

9 Related Work

In addition to the confidential computing solutions introduced in § 2.1, this section describes other prior work related to TwinVisor.

ARM Virtualization. The hardware virtualization for ARM has been enabled since ARMv8.0 and KVM/ARM introduced the split-mode virtualization, which runs a lowvisor in EL2 to help manage hypervisor-related registers and forward control flows between VMs and the highvisor. HypSec [62] utilizes the split-mode architecture to defend VM privacy against malicious hypervisors. But its VMs still fail to benefit from the security features provided by TrustZone. TwinVisor leverages the hardware isolation of TrustZone to guard S-VMs from the entire normal world.

Systems based on ARM TrustZone. Before S-EL2, some systems utilize software techniques to support multiple virtual TEE-Kernel instances in the secure world simultaneously [44, 53, 60, 64]. TEEv [64] utilizes the *same privilege isolation* to run the TEE instances and TEE-visor in the same privilege level while enforcing isolation among them inside TrustZone. Though TEEv theoretically supports running full-fledged Linux, it has to add more functionalities and inevitably increase the complexity of the TEE-visor. vTZ [53] creates a virtual TrustZone instance for each normal world VM. Similar to TwinVisor, it also decouples security from management by reusing Xen hypervisor. But vTZ does not leverage secure memory to protect virtual TrustZone instances and assumes that TAs in the virtual TrustZone are invoked by normal applications infrequently. Hence, the design of vTZ would incur non-trivial overhead for a normal VM due to the numerous costly world switches.

With S-EL2, Hafnium [19] runs an independent hypervisor in the secure world to virtualize multiple simple TEE OSes. It requires modifications of existing Trusted Applications (TAs) and Client Applications (CAs) to communicate through new SMC interfaces. By contrast, TwinVisor focuses on supporting unmodified complex OSes such as Linux for universal applications.

10 Conclusion

TwinVisor is the first hardware-isolated system that utilizes ARM S-EL2 to provide confidential VMs for ARM platforms. It decouples security protection from resource management and reuses a full-fledged *N-visor* to minimize the size of *S-visor*. Performance evaluation showed that TwinVisor incurs less than 5% overhead for all applications on SMP VMs.

Acknowledgments

We sincerely thank our shepherd Raluca Ada Popa and anonymous reviewers for their insightful suggestions. Besides, we thank HiSilicon for providing a testbed for our evaluation. This work was supported in part by the National Key Research & Development Program of China (No. 2020YFB2104100), and the National Natural Science Foundation of China (No. 62002218, 61972244, U19A2060, 61925206). Zeyu Mi (yzmizeyu@sjtu.edu.cn) is the corresponding author.

References

[1] Amd secure encrypted virtualization (sev). https://developer.amd.com/sev/. Referenced September 2021.

[2] Amd sev-snp: Strengthening vm isolation with itegrity protection and more. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Referenced September 2021.

[3] Arm architecture reference manual armv8, for armv8-a architecture profile. https://developer.arm.com/architectures/architecture-security-features/threats-andcountermeasures#specbar. Referenced September 2021.

[4] Arm cca future enablement plans. https://static.linaro.org/connect/armcca/presentations/CCATechEvent-210623-MC.pdf. Referenced September 2021.

[5] Arm cca hardware architecture. https://developer.arm.com/documentation/ddi0615/latest/. Referenced September 2021.

[6] Arm cca hardware architecture. https://static.linaro.org/connect/armcca/presentations/CCATechEvent-210623-CGT-2.pdf. Referenced September 2021.

[7] Arm confidential compute architecture. https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture. Referenced September 2021.

[8] Arm corelink tzc-400 trustzone address space controller technical reference manual. https://developer.arm.com/documentation/ddi0504/c/. Referenced September 2021.

[9] Arm fixed virtual platforms. https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms. Referenced September 2021.

[10] Arm system memory management unit architecture specification, smmu architecture version 3. https://developer.arm.com/documentation/ihi0070/latest. Referenced September 2021.

[11] Aws custom image. https://docs.aws.amazon.com/vm-import/latest/userguide/vmie_prereqs.html. Referenced September 2021.

[12] Aws graviton processor. https://aws.amazon.com/ec2/graviton/. Referenced September 2021.

[13] Aws nitro enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/. Referenced September 2021.

[14] Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Referenced September 2021.

[15] cloc: Count lines of code. https://github.com/AlDanial/cloc. Referenced September 2021.

[16] A deep dive into cma. https://lwn.net/Articles/486301/. Referenced September 2021.

[17] Google cloud confidential virtual machines. https://www.wired.com/story/google-cloud-confidential-virtual-machines/. Referenced September 2021.

[18] Google custom image. https://cloud.google.com/compute/confidential-vm/docs/how-to-byoi. Referenced September 2021.

[19] Hafnium architecture. https://review.trustedfirmware.org/plugins/gitiles/hafnium/hafnium/+/HEAD/docs/Architecture.md. Referenced September 2021.

[20] Hisilicon kirin 990 5g. https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-990. Referenced September 2021.

[21] How 3rd generation intel® xeon® scalable processor platforms support 1 tb epcs. https://www.intel.com/content/www/us/en/support/articles/000059614/software/intel-security-products.html. Referenced September 2021.

[22] Huawei cloud elastic cloud server. https://www.huaweicloud.com/en-us/product/ecs.html. Referenced September 2021.

[23] Intel software guard extensions programming reference, 2014. https://software.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-developer-guide.pdf. Referenced September 2021.

[24] Intel® trust domain cpu architectural extensions. https://software.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-cpu-architectural-specification.pdf. Referenced September 2021.

[25] Intel® trust domain extensions (intel® tdx). https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html. Referenced September 2021.

[26] Kvm: Kernel-based virtual machine. https://www.linux-kvm.org/. Referenced September 2021.

[27] Learn the architecture: Aarch64 virtualization. https://developer.arm.com/documentation/102142/0100/Virtualization-Host-Extensions. Referenced September 2021.

[28] Let's encrypt stats. https://letsencrypt.org/stats/. Referenced September 2021.

[29] Leveraging amd sev in the ibm hybrid cloud. https://www.ibm.com/blogs/research/2020/11/amd-sev-ibm-hybrid-cloud/. Referenced September 2021.

[30] Marketsandmarkets: Encryption software market, global forecast to 2025. https://www.marketsandmarkets.com/Market-Reports/encryption-software-market-227254588.html. Referenced September 2021.

[31] Microsoft becomes the first major cloud provider to offer confidential virtual machines. https://mspoweruser.com/microsoft-cloud-provider-confidential-virtual-machines/. Referenced September 2021.

[32] Oracle custom image. https://docs.oracle.com/en-us/iaas/Content/Compute/Tasks/importingcustomimagelinux.htm. Referenced September 2021.

[33] Protecting vm register state with sev-es. https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf. Referenced September 2021.

[34] stress-ng: stress testing a computer system in various selectable ways. https://github.com/ColinIanKing/stress-ng. Referenced September 2021.

[35] Supporting intel® sgx on multi-socket platforms. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf. Referenced September 2021.

[36] Tpm main specification version 1.2, level 2 revision 116, 1 march 2011, trusted computing group. https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-3-Commands_v1.2_rev116_01032011.pdf. Referenced September 2021.

[37] Trust issues: Exploiting trustzone tees. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html. Referenced September 2021.

[38] Xen arm with virtualization extensions. https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions. Referenced September 2021.

[39] Achermann, R., Panwar, A., Bhattacharjee, A., Roscoe, T., and Gandhi, J. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 283–300.

[40] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016).

[41] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev. 37*, 5 (Oct. 2003), 164–177.

[42] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour, B.-A. The turtles project: Design and implementation of nested virtualization.

In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (USA, 2010), OSDI'10, USENIX Association, p. 423–436.

[43] Cerdeira, D., Santos, N., Fonseca, P., and Pinto, S. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. *2020 IEEE Symposium on Security and Privacy (SP)* (2020), 1416–1432.

[44] Cicero, G., Biondi, A., Buttazzo, G. C., and Patel, A. Reconciling security with virtualization: A dual-hypervisor design for ARM trustzone. In *IEEE International Conference on Industrial Technology, ICIT 2018, Lyon, France, February 20-22, 2018* (2018), IEEE, pp. 1628–1633.

[45] Costan, V., and Devadas, S. Intel sgx explained. *Cryptology ePrint Archive, Report 2016/086* (2016).

[46] Dall, C., Li, S.-W., and Nieh, J. Optimizing the design and implementation of the linux ARM hypervisor. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 221–233.

[47] Dall, C., and Nieh, J. Kvm/arm: The design and implementation of the linux arm hypervisor. *SIGARCH Comput. Archit. News 42*, 1 (Feb. 2014), 333–348.

[48] farkhani, R. M., Ahmadi, M., and Lu, L. Ptauth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association.

[49] Ferraiuolo, A., Baumann, A., Hawblitzel, C., and Parno, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 287–305.

[50] Ge, X., Talele, N., Payer, M., and Jaeger, T. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), IEEE, pp. 179–194.

[51] Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V., and Costanzo, D. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 653–669.

[52] Göddeke, D., Komatitsch, D., Geveler, M., Ribbrock, D., Rajovic, N., Puzovic, N., and Ramirez, A. Energy efficiency vs. performance of the numerical solution of pdes: An application study on a low-power arm-based cluster. *Journal of Computational Physics 237* (2013), 132–150.

[53] Hua, Z., Gu, J., Xia, Y., Chen, H., Zang, B., and Guan, H. vtz: Virtualizing ARM trustzone. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, Aug. 2017), USENIX Association, pp. 541–556.

[54] Hunt, G. D. H., Pai, R., Le, M. V., Jamjoom, H., Bhattiprolu, S., Boivie, R., Dufour, L., Frey, B., Kapur, M., Goldman, K. A., Grimm, R., Janakirman, J., Ludden, J. M., Mackerras, P., May, C., Palmer, E. R., Rao, B. B., Roy, L., Starke, W. A., Stuecheli, J., Valdez, E., and Voigt, W. Confidential computing for openpower. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021* (2021), A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, Eds., ACM, pp. 294–310.

[55] Jackson, A., Turner, A., Weiland, M., Johnson, N., Perks, O., and Parsons, M. Evaluating the arm ecosystem for high performance computing. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (New York, NY, USA, 2019), PASC '19, Association for Computing Machinery.

[56] Jarus, M., Varrette, S., Oleksiak, A., and Bouvry, P. Performance evaluation and energy efficiency of high-density hpc platforms based on intel, amd and arm processors. In *Energy Efficiency in Large Scale Distributed Systems* (Berlin, Heidelberg, 2013), J.-M. Pierson, G. Da Costa, and L. Dittmann, Eds., Springer Berlin Heidelberg, pp. 182–200.

[57] Kim, T., Peinado, M., and Mainar-Ruiz, G. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, Aug. 2012), USENIX Association, pp. 189–204.

[58] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 207–220.

[59] Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., and Fetzer, C. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, Association for Computing Machinery, p. 205–221.

[60] Kwon, D., Seo, J., Cho, Y., Lee, B., and Paek, Y. Pros: Light-weight privatized se cure oses in ARM trustzone. *IEEE Trans. Mob. Comput. 19*, 6 (2020), 1434–1447.

[61] Li, M., Zhang, Y., Lin, Z., and Solihin, Y. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *Proceedings of the 28th USENIX Conference on Security Symposium* (USA, 2019), SEC'19, USENIX Association, p. 1257–1272.

[62] Li, S.-W., Koh, J. S., and Nieh, J. Protecting cloud virtual machines from hypervisor and host operating system exploits. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019).

[63] Li, S.-W., Li, X., Gu, R., Nieh, J., and Hui, J. Z. Formally verified memory protection for a commodity multiprocessor hypervisor. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 3953–3970.

[64] Li, W., Xia, Y., Lu, L., Chen, H., and Zang, B. Teev: Virtualizing trusted execution environments on mobile platforms. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2019), VEE 2019, Association for Computing Machinery, p. 2–16.

[65] Lim, J. T., Dall, C., Li, S.-W., Nieh, J., and Zyngier, M. Neve: Nested virtualization extensions for arm. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 201–217.

[66] Liu, J., and Abali, B. Virtualization polling engine (VPE): using dedicated CPU cores to accelerate I/O virtualization. In *Proceedings of the 23rd international conference on Supercomputing, 2009, Yorktown Heights, NY, USA, June 8-12, 2009* (2009), M. Gschwind, A. Nicolau, V. Salapura, and J. E. Moreira, Eds., ACM, pp. 225–234.

[67] Markuze, A., Smolyar, I., Morrison, A., and Tsafrir, D. Damn: Overhead-free iommu protection for networking. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018* (2018), X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds., ACM, pp. 301–315.

[68] Mi, Z., Chen, H., Zhang, Y., Peng, S., Wang, X., and Reiter, M. K. Cpu elasticity to mitigate cross-vm runtime monitoring. *IEEE Transactions on Dependable and Secure Computing 17*, 5 (2020), 1094–1108.

[69] Mi, Z., Li, D., Chen, H., Zang, B., and Haibing, G. (mostly) exitless VM protection from untrusted hypervisor through disaggregated nested virtualization. In *29th USENIX Security Symposium (USENIX Security 20)* (Boston, MA, Aug. 2020), USENIX Association.

[70] Mi, Z., Li, D., Yang, Z., Wang, X., and Chen, H. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[71] Narayanan, V., Balasubramanian, A., Jacobsen, C., Spall, S., Bauer, S., Quigley, M., Hussain, A., Younis, A., Shen, J., Bhattacharyya, M., and Burtsev, A. Lxds: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 269–284.

[72] Orenbach, M., Lifshits, P., Minkin, M., and Silberstein, M. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017).

[73] Ou, Z., Pang, B., Deng, Y., Nurminen, J. K., Ylä-Jääski, A., and Hui, P. Energy- and cost-efficiency analysis of arm-based clusters. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (2012), pp. 115–123.

[74] Pinto, S., and Santos, N. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv. 51*, 6 (Jan. 2019).

[75] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (July 1974), 412–421.

[76] Raj, H., Saroiu, S., Wolman, A., Aigner, R., Cox, J., England, P., Fenner, C., Kinshumann, K., Loeser, J., Mattoon, D., Nystrom, M., Robinson, D., Spiger, R., Thom, S., and Wooten, D. Ftpm: A software-only implementation of a tpm chip. In *Proceedings of the 25th USENIX Conference on Security Symposium* (USA, 2016), SEC'16, USENIX Association, p. 841–856.

[77] Raj, H., and Schwan, K. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High-Performance Distributed Computing (HPDC-16 2007), 25-29 June 2007, Monterey, California, USA* (2007), C. Kesselman, J. J. Dongarra, and D. W. Walker, Eds., ACM, pp. 179–188.

[78] Rashid, F. Y. The rise of confidential computing: Big tech companies are adopting a new security model to protect data while it's in use. *IEEE Spectrum 57*, 6 (June 2020), 8–9.

[79] Shi, J., Song, X., Chen, H., and Zang, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)* (2011), pp. 194–199.

[80] Shinde, S., Tien, D. L., Tople, S., and Saxena, P. Panoply: Low-tcb linux applications with SGX enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017* (2017), The Internet Society.

[81] Spear, M. F., Shriraman, A., Hossain, H., Dwarkadas, S., and Scott, M. L. Alert-on-update: a communication aid for shared memory multiprocessors. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007* (2007), K. A. Yelick and J. M. Mellor-Crummey, Eds., ACM, pp. 132–133.

[82] Xia, Y., Liu, Y., and Chen, H. Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013* (2013), IEEE Computer Society, pp. 246–257.

[83] Yitbarek, S. F., Aga, M. T., Das, R., and Austin, T. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), pp. 313–324.

[84] Zhang, F., Chen, J., Chen, H., and Zang, B. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011).

[85] Zhang, X., Zheng, X., Wang, Z., Yang, H., Shen, Y., and Long, X. High-density multi-tenant bare-metal cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 483–495.

[86] Zhao, L., Shuang, H., Xu, S., Huang, W., Cui, R., Bettadpur, P., and Lie, D. Sok: Hardware security support for trustworthy execution. *CoRR abs/1910.04957* (2019).