Scalable Memory Protection in the PENGLAI Enclave

Erhu Feng[1†‡], *Xu Lu*[1†‡], *Dong Du*[†‡], *Bicheng Yang*[†‡], *Xueqiang Jiang*[†‡], *Yubin Xia*[†§‡], *Binyu Zang*[†§‡], *Haibo Chen*[†§‡]

[†]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

[§]*Shanghai AI Laboratory*

[‡]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

Abstract

Secure hardware enclaves have been widely used for protecting security-critical applications in the cloud. However, existing enclave designs fail to meet the requirements of scalability demanded by new scenarios like serverless computing, mainly due to the limitations in their secure memory protection mechanisms, including static allocation, restricted capacity and high-cost initialization. In this paper, we propose a software-hardware co-design to support dynamic, fine-grained, large-scale secure memory as well as fast-initialization. We first introduce two new hardware primitives: 1) *Guarded Page Table* (GPT), which protects page table pages to support page-level secure memory isolation; 2) *Mountable Merkle Tree* (MMT), which supports scalable integrity protection for secure memory. Upon these two primitives, our system can scale to thousands of concurrent enclaves with high resource utilization and eliminate the high-cost initialization of secure memory using fork-style enclave creation without weakening the security guarantees.

We have implemented a prototype of our design based on PENGLAI [24], an open-sourced enclave system for RISC-V. The experimental results show that PENGLAI can support 1,000s enclave instances running concurrently and scale up to 512GB secure memory with both encryption and integrity protection. The overhead of GPT is 5% for memory-intensive workloads (e.g., Redis) and negligible for CPU-intensive workloads (e.g., RV8 and Coremarks). PENGLAI also reduces the latency of secure memory initialization by three orders of magnitude and gains 3.6x speedup for real-world applications (e.g., MapReduce).

1 Introduction

There has been a surge of interest in using enclaves like Intel SGX [77], AMD SEV [12] and ARM TrustZone [58] to host security-critical applications in cloud with minimal reliance on the trust of cloud providers [28, 29, 37, 40, 46, 59, 84, 87, 96]. Meanwhile, microservice [80] and serverless computing [1, 3–5] have become emerging paradigms of cloud, which use *single-purpose service* or *function* as a basic computation unit and achieve high scalability. Since the frameworks of both are also managed by the cloud providers, it is natural to use enclaves to protect serverless-like and microservice-like applications in the cloud [36, 93].

However, existing enclave systems cannot well fit some inherent characteristics of these cloud applications, including resilient memory allocation [86], high resource utilization [43, 76], auto-scaling [43] and ephemeral execution time [34, 56, 64], mainly due to three scalability limitations of their secure memory protection mechanisms:

Limitation-1. Non-scalable memory partition/isolation: Most existing enclave systems use static or almost-static partition for region-based memory isolation, like fixed-sized PRM (Processor Reserved Memory) in SGX [15], limited secure world memory regions in ARM TrustZone [2][2], 16 protected memory regions in Keystone [13, 68], etc. It is hard to dynamically adjust the boundaries of partitions and scale to a large amount of secure memory. Region-based isolation also violates the scalability requirement of fined-grained memory management in the cloud.

Limitation-2. Non-scalable memory integrity protection: Using a traditional Merkle hash tree (or its variants) to protect memory integrity is hard to scale. For example, Intel SGX only supports 128/256MB EPC (Enclave Page Cache)[3]. Although SGX has no restriction on the number of enclaves, running thousands of enclaves may either cause little available EPC for each instance or frequent EPC swapping, which fails to meet the scalability requirement of serverless computing.

Limitation-3. Non-scalable secure memory initialization: High-cost secure memory initialization causes long startup latency for enclaves, which significantly affects the performance of auto-scaling. For example, SGX needs seconds to create an enclave [40, 69] by adding memory to EPC and measuring the contents (by EADD and EEXTEND instructions) for every page. On the contrary, serverless functions usually have a very short life cycle (<1s) [50, 88].

In this paper, we propose scalable secure memory protection mechanisms for enclaves with three metrics: (1) size and granularity of secure memory, (2) number of enclaves, (3)

[1]The two authors contributed equally to this work and should be considered co-first authors.

[2]The number depends on specific implementations, and is typically 8.

[3]The latest SGX platforms (Ice Lake Server) support larger EPC sizes (e.g., 1TB) [22], but they do so by giving up on integrity protection.

startup latency of enclaves. We first introduce two new architectural primitives, *Guarded Page Table* (GPT) and *Mountable Merkle Tree* (MMT): GPT protects page table pages and enables memory isolation with page-level granularity, and MMT is a new abstraction to achieve on-demand and scalable memory encryption and integrity protection. Leveraging these two primitives, a lightweight *secure monitor* running in the most privileged mode is in charge of enclave management and maintaining security guarantees. To mitigate the high overhead of enclave creation due to costly secure memory initialization, we propose a new type of enclave, called *shadow enclave*, to support fork-style fast enclave creation.

We have implemented a prototype of our design based on PENGLAI [24], an open-source RISC-V enclave system using a secure monitor to manage all the enclaves. We extend the secure monitor to support scalable secure memory protection with two hardware primitives, as well as fast enclave creation. The evaluation results show that PENGLAI can host 1,000s of concurrent enclave instances and support secure memory up to 512GB. PENGLAI incurs negligible overhead for CPU-intensive benchmarks (e.g., RV8 and Coremarks), and incurs 5% overhead for memory-intensive benchmarks (e.g., Redis). For startup latency, PENGLAI leverages the shadow enclave to boost enclave creation by three orders of magnitude (with 16MB enclave memory). We also evaluate PENGLAI with real-world scalable applications. The results show that PENGLAI can significantly reduce the execution time of MapReduce (3.6x speedup with shadow fork) and achieve near-native performance for a serverless application. We have implemented all the architectural features of PENGLAI on RISC-V platform, including an FPGA board, QEMU and the Gem5 simulator, and implemented the software monitor with 6,399 LoCs, including enclave/hardware management and encryption libraries. The hardware costs are also minor, i.e., 0.81% (without MMT or memory encryption) in LUT and 0.73% in FF on a Xilinx VC707 FPGA board.

PENGLAI is open-sourced at https://github.com/Penglai-Enclave.

2 Motivation

This section analyzes the scalability and security of prior enclave systems through several metrics, as shown in Table 1.

2.1 State-of-the-art Enclaves

Intel SGX [57, 77] can protect both confidentiality and integrity of enclave memory, but it has a restriction on secure memory size, i.e., 128/256MB. Also, the secure memory must reside in a contiguous region (PRM) reserved by the CPU in advance. Recently, Intel has released a scalable version of SGX [22], which extends the secure memory size to TB level but weakens the memory integrity protection, and still requires a static reserved secure memory region. AMD SEV [12, 27, 61] protects virtual machines (VMs) without memory size restriction. However, the number of secure VMs is restricted to 16 (509 in EPYC Generation 2 [14]). Intel TDX [16] is also designed to isolate secure VMs from other software, including the hypervisor. However, TDX only provides basic integrity protection and cannot defend against hardware-based memory replay attacks. The number of secure VMs is limited by hardware to 64 private keys in MKTME (Multi-Key Total Memory Encryption). ARM TrustZone-based enclaves, e.g., Komodo [52] and Sanctuary [35], have no restriction on enclave number or memory size. However, the secure memory can only reside in a few fixed memory regions and has no encryption or integrity guarantees.

Keystone [68] implements enclave memory isolation by leveraging the PMP (Physical Memory Protection) mechanism of RISC-V [100], which includes a set of paired registers to indicate physical memory regions as well as their access permissions. Thus, the number of memory regions in Keystone is restricted by the number of PMP registers (up to 16). In order to defend against physical attacks, Keystone leverages on-chip computing, which is costly due to the restricted on-chip RAM [68]. CURE [21] adopts enclave ID-based access control for customizable enclaves. It utilizes a hardware arbiter to record contiguous physical memory regions of enclaves, which can only support 13 enclaves. Similarly, the enclave number of Sanctum [45] is also restricted by the number of isolated DRAM regions. TIMBER-V [102] extends the RISC-V ISA to run unlimited number of enclaves, but it incurs non-trivial overhead (25.2% on average) and does not consider memory integrity protection.

2.2 State-of-the-art Fall Short

Fine-grained memory isolation. Prior art [27,91,92] achieves fine-grained and flexible memory isolation by introducing additional metadata like bitmap [27, 92] or tags [102] to identify whether a page belongs to an enclave and check each memory access. However, due to the capacity restriction of in-SoC RAM, most of the metadata has to be stored in the main memory, which requires an extra memory load when metadata is out of SoC and incurs a high performance penalty in TIMBER-V [102].

Large-scale memory integrity protection. Several schemes [38, 39, 85, 91, 92] have been proposed to provide integrity protection for more memory. For example, VAULT [92] extends SGX and optimizes the integrity tree node structure to increase the node's fan-out, which can protect larger memory region given the same tree depth. However, these schemes need to take up extra memory space even when no enclaves are running (e.g., 14.1% in VAULT without MAC optimization), and the size of protected memory (e.g., 64GB in VAULT) is still insufficient for cloud applications.

Boosting startup latency. Some researchers add a new software layer to manage enclaves, like the secure OS in ARM TrustZone [26] and the libOS in Occlum [89], which can

Systems		Scalability Metrics				Security Metrics			
Name	Arch	Enclave number	Mem size	Fast startup	Mem granu.	No PT channel	No Cache channel	Mem enc.	Mem inte.
SGX [57, 77]	Intel	Unrestricted	128/256MB	✗	Region	✗	✗	✓	✓
Scalable SGX [22]	Intel	Unrestricted	All	✗	Region	✗	✗	✓	✗
TDX [16]	Intel	64	All	✗	Page	✓	✗	✓	Partial
SEV [12, 61]	AMD	16/509	All	✗	Page	✗	✗	✓	✗
SEV-ES [61]	AMD	16/509	All	✗	Page	✗	✗	✓	✗
SEV-SNP [27]	AMD	16/509	All	✗	Page	✗	✗	✓	✗
Trustzone [26]	ARM	Unrestricted	All	✓	Region	✓	✗	✗	✗
Komodo [52]	ARM	Unrestricted	All	✓	Region	✓	✗	✗	✗
Sanctuary [35]	ARM	Unrestricted	All	✗	Region	✓	✓	✗	✗
Sanctum [45]	RISC-V	DRAM regions	All	✗	Region	✓	✓	✗	✗
TIMBER-V [102]	RISC-V	Unrestricted	All	✗	Page	✓	✗	✗	✗
Keystone [13, 68]	RISC-V	PMPs	All	✗	Region	✓	✓	On-chip	On-chip
CURE [21]	RISC-V	13	All	✗	Region	✓	✓	✗	✗
PENGLAI	**RISC-V**	**Unrestricted**	**All**	✓	**Page**	✓	✓	✓	✓

Table 1: A comparison on enclave systems. *Mem granu.* means the granularity of secure and non-secure memory. Region means secure memory can only reside in a few contiguous memory regions. *Mem enc.* means memory encryption. *Mem inte.* means memory integrity protection. *Unrestricted* means the number of enclaves is unrestricted, but when secure memory is exhausted, the performance will decline. *Partial* means the physical memory replay attack is out of scope. *16/509* means EPYC Generation 2 (Rome) processors [14] can support 509 keys for SEV VM. PENGLAI is the only system that can achieve both high-security and scalability.

create a new enclave with less overhead. However, these systems do not consider the process of attestation during enclave creation. Clemmys [93] leverages the dynamic memory management of SGX2 as well as batching for EPC augmentation to improve the startup latency of enclaves. However, creating an enclave still takes hundreds of milliseconds, and it needs to add redundant pages into EPC when creating the same enclave multiple times.

3 System Overview

We first present our design goals of memory protection.

- **G1: Scalability.** The design shall not have restrictions on (1) the number of concurrent enclave instances, and (2) the size of secure memory of enclaves. It shall also consider the characteristics of scalable applications, e.g., fine-grained memory management and auto-scaling.
- **G2: Performance.** The design shall not incur high performance overhead.
- **G3: Security.** The design shall achieve the previous two goals with security guarantees. It should consider privileged software attacks, off-chip hardware attacks (e.g., hardware-based memory replay attacks), cache-based side-channel attacks, etc.

3.1 Architecture

As shown in Figure 1, PENGLAI is a software-hardware co-design enclave system. We introduce a small software component called *secure monitor*, which runs in the most privileged mode (e.g., machine mode in RISC-V) and several new hardware extensions to provide the enclave abstractions. Each enclave runs in the user space and is isolated from an untrusted host and other enclaves.

Secure monitor. The secure monitor runs in the most privileged level and separates OS and userspace software into two worlds: one for the OS and normal applications, the other

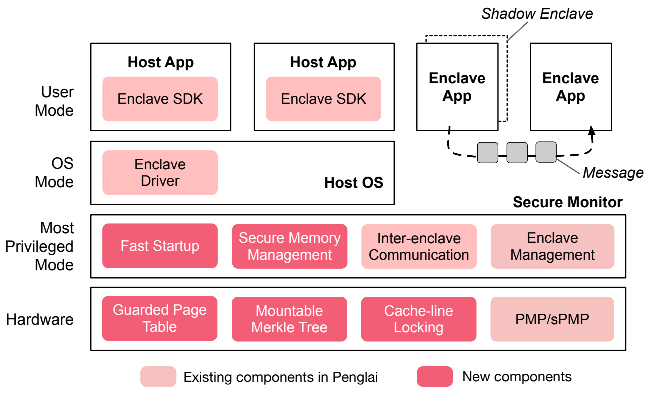


Figure 1: Overview of PENGLAI architecture. PENGLAI is composed of the software monitor, driver, SDK and hardware extensions. The red components are additional to realize the scalable memory protection.

for enclaves. The secure monitor manages all enclaves and provides APIs for users to deploy enclaves. To achieve scalability, we add two components in the secure monitor: one for fine-grained and large-scale secure memory management, and the other for enclave fast startup. Furthermore, to minimize the size of the secure monitor, we separate resource protection from management [51]: the secure monitor only configures privileged hardware resources (e.g., GPT and MMT configurations), and the managements of other hardware are done by the untrusted host OS.

During system boot, the secure monitor is loaded and verified by the boot ROM (aka. secure boot). It then takes control of the system and protects itself with hardware-supported memory isolation (e.g., RISC-V PMP). It also leverages encryption and Mountable Merkle Tree to protect itself from physical memory attacks (details in 4.2).

Hardware primitives. We propose new hardware primitives to assist the secure monitor and achieve scalable memory protection. We briefly introduce their purposes here. Guarded Page Table (GPT) is the basis of fine-grained memory isolation (§4.1). Mountable Merkle Tree (MMT) is a new physical memory protection abstraction to achieve scalable memory integrity and encryption protection (§4.2). Cache line locking is a cache partition extension to defend against cache-based side-channel attacks (§4.4).

3.2 Threat Model

The TCB of our system only contains the CPU and the secure monitor. Other hardware (e.g., off-chip DRAM and peripherals) and software (e.g., the host OS) are untrusted and could be compromised by an attacker.

We consider three classes of attacks in our threat model:

- **Privileged software attacks:** An attacker may have full control of the untrusted OS and applications and launch adversarial enclaves.
- **Physical attacks:** An attacker may intercept and tamper with any messages between CPU and other hardware (e.g., DRAM), and issue attacks from any off-chip hardware.
- **Side-channel attacks:** An attacker may learn information by observing access patterns during enclaves' execution. Our system aims to solve the controlled channel attacks [81] and cache-based side-channel attacks [74, 82, 107, 109, 110]. Other side-channel attacks, like the ones based on TLB or speculative execution, are out of the scope. The potential defense mechanisms of these attacks are orthogonal to our design.

Our system does not consider DoS attacks performed by untrusted software or hardware.

4 Design

This section focuses on how the secure monitor and the hardware extensions achieve the design goals. We discuss other security issues in §7.

4.1 Fine-grained Flexible Memory Isolation

For fine-grained and flexible memory isolation, the secure monitor maintains an ownership bitmap to record the status of each physical page: *secure* for monitor and enclaves, *non-secure* for untrusted OS and applications, and *TreeNode* for SubTrees (details in §4.2). It achieves 4KB page granularity to isolate memory between enclaves and host. To allocate secure pages, the secure monitor needs to update the ownership of them in the bitmap. The ownership bitmap is protected by hardware (e.g., RISC-V PMP). Any memory access to the ownership bitmap issued by OS or user-level programs will trap into the monitor for a security check.

Several similar approaches [27, 91, 92, 102] also use the ownership bitmap to achieve fine-grained memory isolation. However, these approaches check the page ownership during the memory access, which needs double memory access for a single address. Prior work demonstrates that the double

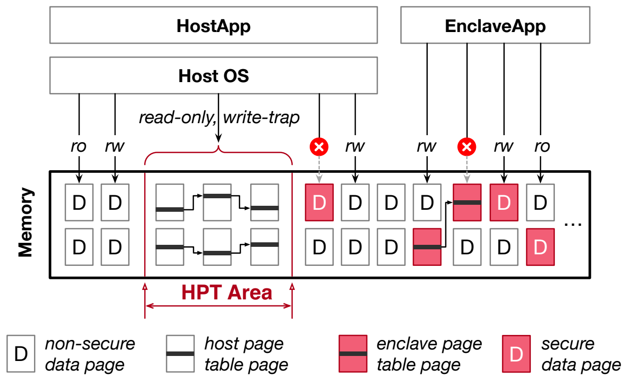


Figure 2: The design of Guarded Page Table with Host Page Table Area (HPT Area). PENGLAI maintains two kinds of page tables: the host page table is used by untrusted software, and the enclave page table is used by enclave.

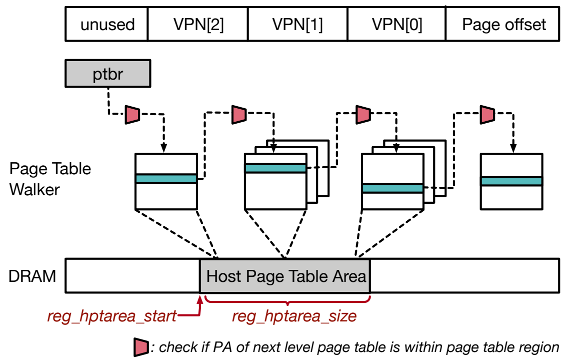


Figure 3: The MMU extension of Guarded Page Table. Extensions are marked as red. MMU will check the location of each page table. VPN means virtual page number.

memory access may introduce the 25.2% average runtime overhead [102].

We propose a new hardware primitive, Guarded Page Table, to shift the ownership checking overhead to the mapping phase, which is based on an observation that mapping operations are far less frequent than memory accesses.

Guarded Page Table. Guarded Page Table implements and optimizes ownership checking with bitmap. The design is based on our insight that if there is no page table of untrusted software containing any mapping to secure pages, then the checking during memory access can be avoided. To this end, we put all host page tables (Figure 2) in a protected memory region: Host Page Table Area (HPT Area), and trap any modification in HPT Area to ensure that *no secure page will be mapped by any page tables of untrusted software*.

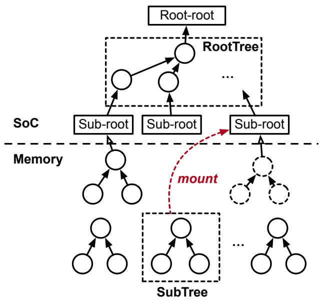
HPT Area is indicated by two new registers, *reg_hptarea_start* (start physical address of HPT Area) and *reg_hptarea_size* (size of HPT Area), as shown in Figure 3. To guarantee all the host page tables are located in HPT Area, we extend page table walker (PTW). When a TLB miss occurs, the PTW will walk into each level of page table page (PT page) according to the virtual address to get a corresponding physical address. Our extension to PTW will check whether each PT page is located in HPT Area, as shown in Figure 3. If the address of any PT page is out of HPT Area and the software currently running is not an enclave, the CPU will raise an exception to the monitor for further check. Furthermore, a CPU mode status register, *reg_ms*, is introduced to indicate whether the current CPU is running an enclave. Therefore, we can enforce the untrusted OS to only use pages in HPT Area as Guarded Page Table.

Protecting Host Page Table Area (HPT Area). Similar to the ownership bitmap, HPT Area is also protected by the secure monitor with hardware support. When the OS updates address mappings, the request will be redirected to the secure monitor, which ensures the new page table entry does not point to a secure page. Also, we need to prevent the OS from bypassing such checking via stale TLB entries or disabling page table. Firstly, the secure monitor will flush the corresponding TLB entries during enclave switching and page ownership changing. Directly writing TLB entries is out of scope as there is no such instruction in the prevailing ISA (e.g., X86, RISC-V). Secondly, the hardware will raise an exception if address translation is disabled while the *reg_hptarea_start* and *reg_-hptarea_size* registers are non-zero (these two registers can only be modified by monitor). HPT Area is also protected from hardware attacks such as the PThammer attack [113] via hardware integrity protection (details in §4.2).

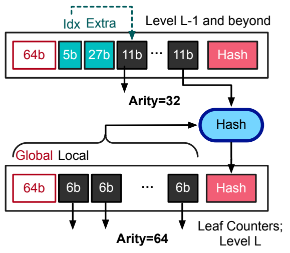
Memory isolation among enclaves. As all enclaves are running in user mode, PENGLAI utilizes an enclave page table for memory isolation among enclaves. Enclave page tables are marked as secure memory and separated from HPT Area. The secure monitor maintains all the enclave page tables, and each enclave can map its own secure memory as well as non-secure memory shared with the OS.

Huge page support. To support huge pages, we further partition the HPT Area into several sub-areas and assign different levels of page table entries to the corresponding sub-area, including one sub-area for PMDs (huge page entry) and one for PTEs. The extended PTW will check whether each level of page table entry lies in the corresponding sub-area during page table walk. In this way, the secure monitor can distinguish a huge page table entry via its address and perform different security checks.

Summary. We summarize the benefits of the ownership-based design with Guarded Page Table. First, the hardware modification is minor, and the hardware maintains no in-memory metadata like SGX EPCM. Second, it achieves fine granularity since any physical memory page can be used as secure or non-secure. The only contiguous range, Host Page Table Area region, has little impact on scalability as the page table pages are much less than data pages. Last, the design introduces no overhead of checking during memory access. The only costs come from the page table mapping operations. Compared with other page table-based isolation, e.g., shadow PT [94], the mapping overhead is minor.



(a) Forest structure.



(b) Counter layout.

Figure 4: Mountable merkle tree. (a) The top panel is the mounted SubTree root and RootTree. The bottom panel is the hash forest constituted by multi-SubTrees. (b) The arity of SubTree is 32, 32 and 64, while 32, 32, 32 for RootTree. The "Idx" points to a local counter, which can use the "Extra" to avoid frequently global counter updating.

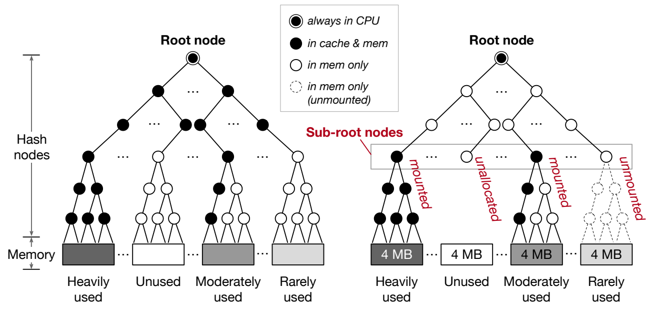
4.2 Scalable Memory Integrity Protection

We propose a new physical memory protection **abstraction**: *Mountable Merkle Tree (MMT)*. MMT promises a stable tree depth that will not increase along with total memory size and minimizes the memory space overheads by storing integrity metadata (tree nodes) on demand. Furthermore, the secure monitor can manage MMT to achieve large-scale, fine-grained memory integrity protection. In our prototype, MMT can support up to 512GB of secure memory.

Challenge. It is very challenging to achieve scalable integrity protection, as shown in Figure 5 (a). First, protecting integrity for large-scale secure memory turns to a deep integrity tree, which requires additional bandwidth to load tree nodes. Second, to boost memory integrity checking, a wise memory integrity engine may cache topmost tree nodes in the CPU cache. However, it only increases the amount of secure memory linearly. Third, the integrity engine needs to pre-allocate extra memory to store all the tree nodes, even if there is no secure memory being used. Lastly, the state-of-the-art memory protection schemes can only protect a fixed range of memory, and software cannot manage secure memory at all. These coarse-grained and fixed memory protection schemes have scalability issues which cannot be solved by just adding hardware resources (e.g., enlarging SoC storage).

MMT introduces a mountable SubTree structure for integrity tree scheme and can reduce both on-die and in-memory storage overhead. Also, MMT allows the software to take part in memory protection management and allocate secure memory with integrity protection on demand.

MMT forest organization. MMT introduces a new concept, *hash forest*, which is composed of a set of SubTrees, as shown in Figure 4 (a). The SubTree is the mountable and manageable unit in *hash forest* and can protect a physical memory region (4MB/3-level in PENGLAI) alone. To save on-die space, MMT stores all SubTrees in a specific memory region, *MMT meta-*



(a) Static Merkle Tree. (b) Mountable Merkle Tree.

Figure 5: Comparison between static Merkle tree and MMT.

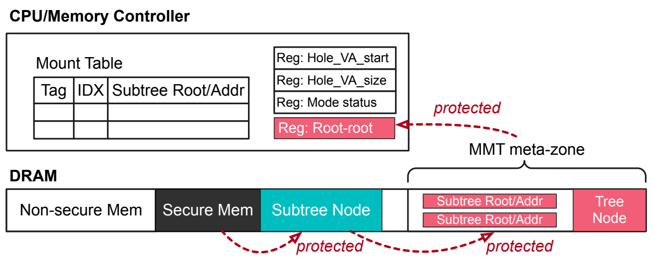


Figure 6: Hardware extension for MMT. The flow shows the integrity protection relation: secure memory is protected by SubTree nodes, while SubTree nodes are protected by MMT meta-zone.

zone. MMT utilizes *RootTree* to protect the integrity of all SubTree roots. The ancestor of RootTree (*Root-root* in the figure) is fixed in SoC to prevent attacks. The Root-root, RootTree, and SubTrees, together form *hash forest*.

SubTree Allocation. Besides normal memory allocation, the software (secure monitor) can also allocate SubTree and secure memory with two new interfaces: *ALLOC_SECURE_-MEM*, *REVOKE_SECURE_MEM*. Each SubTree protects a range of secure memory. Unlike traditional memory protection schemes, the CPU can only protect a fixed range of physical memory, and the whole checking procedure is transparent for software. MMT allows the privileged software (e.g., the secure monitor) to allocate secure memory at runtime, and both secure memory and SubTree can reside in anywhere of physical memory, not a fixed range.

Mounting. Like TLB accelerating virtual address translation, MMT accelerates integrity checking by mounting SubTrees into SoC, which avoids the memory access to retrieve SubTree root. MMT extends the memory controller to support mounting operations. As shown in Figure 6, the mounted SubTree root is stored in the *Mount table*, which records counter and address. However, the storage space for *Mount table* in SoC is restricted, e.g., 32 subtrees simultaneously. When space is exhausted, MMT unmounts an inactive SubTree root out of SoC and stores it in the MMT meta-zone, which is isolated with host memory (e.g., PMP-protected). Meanwhile, MMT needs to update the value of Root-root if MMT meta-zone is changed, which ensures the integrity of all inactive SubTree roots.

Bootstrap. Figure 6 demonstrates the hardware extension and memory layout of MMT. Besides the non-secure memory, there are three protected regions, *secure memory* (enclaves and monitors), *SubTree nodes* and *MMT meta-zone*. The MMT meta-zone is the only fixed region configured by bootrom and protected by hardware (e.g., RISC-V PMP). MMT meta-zone contains the SubTree root entries (address and counter) and RootTree nodes (Root-root is reserved in SoC). It incurs minor memory costs — about 2MB, which can support up to 512GB of integrity-protected memory. The integrity protection relation of these three regions is shown in the figure.

During system boot, the CPU bootrom will configure the MMT meta-zone range in physical memory, initialize all SubTree root entries, construct RootTree nodes and allocate the first subtree to protect the memory of the secure monitor. After this, the secure monitor takes control of subsequent booting stages.

Software management. The secure monitor is the only privileged software that can manage secure memory and record memory status (secure, normal and SubTree node) in the bitmap. As the secure monitor cannot directly allocate memory, the host kernel will allocate free memory (used as secure memory and SubTree node later) and transfer it to the secure monitor. The secure monitor configures secure memory and its corresponding SubTree in this memory. After allocation, the corresponding SubTree root is filled in the *MMT meta-zone* (i.e., *ALLOC_SECURE_MEM*). The monitor also ensures that secure memory is zero-filled and SubTree nodes are in the initialized state. So the integrity check will not fail at the first access. If the monitor needs to reclaim secure memory, it can change the memory status, clear the SubTree root in the *MMT meta-zone* (i.e., *REVOKE_SECURE_MEM*), and return memory back to the host kernel.

MMT tree structure. MMT extends the counter-based integrity tree node [55, 85, 92] to *hybrid-counter*. As shown in Figure 4 (b), each tree node is composed of $global\_counter(64b)$, $extra\_idx(5b)$, $extra\_counter(27b)$, $local\_counter(32 \times 11b)$ and $hash(64b)$ (however, leaf node only contains global, local counters and hash). The hash in the tree node is calculated with all other metadata (448b) in the same node and hybrid-counter in its parent node. The *extra counter* in a hybrid-counter remains zero unless *extra_idx* refers to itself. This design is based on an observation that there is almost one active counter in a single tree node. In our implementation, both SubTree and RootTree are 3-level, and each RootTree leaf node can contain 4 SubTree roots.

Integrity checking. For each secure memory access, MMT will first check whether the corresponding SubTree root is in SoC. If it is, MMT checks the integrity with the SubTree. Otherwise, MMT uses *mounting* mechanism to mount the target SubTree into SoC. As for the integrity checking procedure, the MMT engine will compare the hash stored in the tree node with the hash it computes level by level, until the hash is stored in SoC. A write request will increase hybrid-counter in the tree node, while a read request will not. If a local counter is exhausted or *extra_idx* is changed, MMT will re-hash the relevant tree nodes.

Integrity enabling. PENGLAI will always enable integrity protection when executing in the monitor. Nevertheless, as an enclave can access both secure and non-secure memory, we cannot just impose integrity protection on all its pages.

PENGLAI introduces a pair of *hole registers* (*reg_hole_va_start* and *reg_hole_va_size*) to configure the integrity checking. The effects are shown on the right. Hardware shall perform integrity checking when *reg_ms* is *ENCL_MODE* and disable it when *reg_ms* is *NON_ENCL_MODE*. Hole registers indicate a dedicated memory region (hole region) in the virtual address that handles exceptional cases in the memory protection strategy, e.g., integrity checking is enabled for the memory located in the hole region when *reg_ms* is *NON_ENCL_MODE*, and disable when *reg_ms* is *ENCL_MODE*. The hole region can be used for sharing untrusted memory between enclaves and OS. Besides, it cannot be used by host and enclave for other purposes. To avoid disabling the integrity protection by the malicious kernel, PENGLAI only permits the secure monitor to configure the hole registers when switching between enclave and host.

	MS=ENCL	MS=NON_ENCL
Hole mem.	Disable	Enable
Non-Hole mem.	Enable	Disable

Secure memory granularity. As integrity enabling is controlled by the combination of CPU mode (*reg_ms*) and virtual memory address (*hole register*), the SubTree is not the granularity of secure memory. A SubTree can contain both secure memory and non-secure memory, and only secure memory needs integrity and encryption protection. When considering Guarded Page Table for memory isolation, the combined granularity of secure memory is still 4KB size.

Security analysis. As shown in Figure 6, the integrity of secure memory is guaranteed by SubTree root. Each SubTree root has a backup in the MMT meta-zone, which is protected by Root-root. As Root-root resides in SoC, physical attackers cannot compromise the integrity check. As for software attackers, only the secure monitor can configure secure memory, and any other privileged software cannot tamper memory status and disable integrity protection.

Summary. We summarize the benefits of MMT against prior art, as shown in Figure 5. (1) Save both on-die and in-memory storage. Prior art reserves intermediate tree nodes for all secure memory, and the topmost level tree nodes may be stored in SoC to boost the integrity check [15]. It can only protect a small region of contiguous memory due to the high storage overhead in memory and SoC. However, MMT merely reserves the hot set of SubTree roots and Root-root in the SoC. What's more, SubTree nodes can be lazily allocated with corresponding secure memory (zero memory overhead if there is no secure memory). (2) Improve flexibility in secure memory management. With the help of allocating and mounting operations for SubTrees, MMT can support fine-grained secure memory. The software can manage secure memory, dynamically change memory status, and allocate secure memory on-demand. (3) Boost the integrity check. MMT can provide a fast path (a mounted SubTree) to boost the integrity check with fixed tree depth and save memory bandwidth.

4.3 Secure Memory initialization with Shadow Fork

Auto-scaling and fast startup are key features for cloud computing but still missing in the enclave due to high-cost enclave memory initialization. Prior systems need to create a new enclave instance from scratch even with the same codebase, which consumes more memory with redundant content and incurs high startup latency. PENGLAI follows the idea of recently proposed init-less startup [50] that leverages *fork* to skip the initialization costs, but faces two challenges: 1) memory sharing is not secure in enclave systems, and 2) attestation costs still remain even with fork. PENGLAI proposes *Shadow Fork* as well as *Shadow Enclave* to overcome them both.

Fork with the shadow enclave. Shadow Fork is based on a special kind of enclave (not runnable), *shadow enclave*, which is a clean template used to boost startup by forking a new instance. Shadow enclave is the only entity that can be forked and only contains code and data segments. During fork, PENGLAI monitor will share the read-execute code and read-only segments, copy other writable parts, initialize the stack of a new instance based on *Shadow Enclave*. As the major costs of startup come from enclave memory initialization (hash measurement), memory copying on writable data is acceptable. After fork, the created enclave can dynamically allocate memory from untrusted OS as heap or mmap region.

Lightweight attestation. Mitigating the costs of attestation during startup is based on an observation: calculating the measurement of memory takes up the majority of time in attestation (e.g., >90%), as shown in Figure 11 (b). To mitigate this overhead, the monitor will calculate the measurement for a *shadow enclave* in advance (creation phase). A user can leverage *enclave_fork* with a manifest containing the sealed enclave measurement and the user's public key (similar to SGX [23]). Later, the monitor will unseal the enclave measurement (using the user's public key) and check it with the *shadow enclave*'s measurement. If the measurement is matched, the monitor will fork a new instance based on the *shadow enclave*. Otherwise, the monitor will deny the request. Therefore, we can mitigate the attestation costs during the boot critical path.

4.4 On-demand Cache Line Locking

PENGLAI proposes an on-demand cache line locking mechanism to defend against cache-based side-channel attacks by

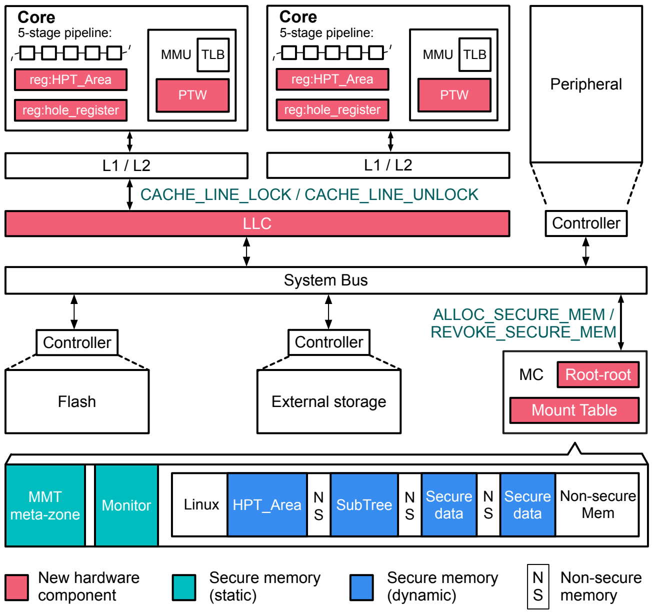


Figure 7: The hardware modification and physical memory layout of PENGLAI.

cache partition. Although cache partition designs are well explored [9, 73, 99, 112], they usually incur non-trivial performance costs [66, 83, 106], and cache partitions also restrict the number of the enclave [45]. PENGLAI optimizes the performance with a new abstraction, *section-based protection*, that can enable on-demand side-channel protection when an enclave enters a security-sensitive section (e.g., encryption) and disable the protection when it leaves. As the sections are short in a program [105, 108], an enclave can run with the best performance (no cache locking) most time.

Security-sensitive sections are decided by enclaves. When an enclave needs cache isolation protection, it issues requests to monitor. Two privileged instructions: *CACHE_LINE_LOCK* and *CACHE_LINE_UNLOCK*, assist monitor in managing cache line locking status. Specifically, the cache line locking mechanism specifies the cache lines to each CPU core. The CPU core can only evict its cache lines during cache miss, and other cores cannot evict these cache lines anymore.

Scalability discussion. Compared with the Intel's CAT [9, 73], PLCache [99] or cache coloring [112], on-demand cache line locking is more scalable. The cache partitions limit the number of protected enclaves in the prior art. However, as for on-demand cache line locking, the cache line is not assigned to an enclave, but a CPU core. In other words, if there are more cache ways than CPU cores, the cache line locking mechanism can support unlimited enclaves. Monitor holds the cache locking status in each enclave's context. If an enclave is scheduled out, the monitor will release the cache lines locked by the current core.

5 Implementation

Figure 7 shows the overall architecture of the PENGLAI. As for hardware extensions, we modify both in-core (Rocket Core) and off-core (Memory Controller) hardware resources to support Guarded Page Table and Mountable Merkle Tree. As for software extensions, we implement a tiny and secure monitor, an extended Linux kernel supporting the HPT Area allocator, an enclave driver and some requisite libraries for running enclaves.

5.1 Hardware Implementation

In-core extensions. We implement the Guarded Page Table extension in FPGA, based on the open-sourced RocketChip RISC-V core [10]. Overall, we add several new registers (e.g., *reg_ms*, *reg_hptarea_start[0..3]*, *reg_hptarea_size*, *reg_hole_va_start* and *reg_hole_va_size*, etc.), which are implemented as CSRs (Control and Status Registers) and can only be accessed by monitor via *csrr* (CSR read) and *csrw* (CSR write) instructions. *Reg_hptarea_start[0...3]* and *reg_hptarea_size* registers partition the physical memory range of HPT Area into several sub-areas, and guarantee that any write access cannot directly modify the content in this area, unless issued by M mode routines. *Reg_ms*, *reg_hole_va_start* and *reg_hole_va_size* registers act as the control switch for memory integrity and encryption checks. Besides these new CSRs, we also extend the MMU module to check the validity of memory access during page table walking. A modified page table walker can guarantee that any PTE entry must be located in the HPT Area (precisely, corresponding sub-area), but the target physical address **will not** reside in the HPT Area.

We only simulate the cache line locking mechanism on the L1 cache, as there is no LLC in our FPGA board.

Off-core extension. We extend the memory controller (MC) to integrate the MMT engine. MMT engine supports three new commands: *ALLOC_SECURE_MEM*, *REVOKE_SECURE_MEM* and *INIT_MMT_METAZONE*; two extended components: Mount Table, Root-root, and the logic of memory encryption and integrity checks. We also extend memory access with the secure/non-secure flag. If it is the secure memory access, the MMT engine will perform integrity and encryption checks. Otherwise, it accesses physical memory and reads/writes the data directly.

The secure monitor allocates/reclaims secure or SubTree node memory with new commands. When the MMT engine receives an *ALLOC_SECURE_MEM* command, it will parse the secure memory address and initialize the SubTree root in the MMT meta-zone. The software must ensure that the memory must be zero-filled before it changes into the secure or SubTree node memory. The *INIT_MMT_METAZONE* command initializes the MMT meta-zone in the booting phase.

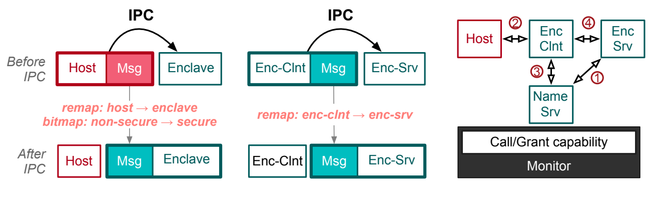
Mount Table and Root-root reside in SoC. The fabric of Mount Table is similar to cache — several sets with n-way Mount Table entries. Each Mount Table entry consists of a tag, index, SubTree root and LRU bit. Mount Table also uses the LRU strategy (clock-like algorithm) to mount/unmount the subtree roots. If SubTree root does not exist in the Mount Table, the MMT engine will choose an inactive SubTree and unmount it into the MMT meta-zone. Before mounting the requested SubTree, the MMT engine will first check the integrity of the SubTree root in the MMT meta-zone, and update the Root-root if necessary. Root-root cannot be evicted out of SoC, as it is the root trust for integrity protection.

5.2 Software Implementation

Monitor. We implement the secure monitor on both OpenSBI [18] and Berkeley Boot Loader (BBL) [17] in the machine mode in RISC-V. The secure monitor includes enclave management, hardware extensions management, memory checking as well as encryption library, which adds 6,399 LoC. We follow Sanctum [45] to implement the secure boot using the tamper-proof software approach (bootloader as the root of trust). As Figure 7 shows, just after the machine is turning on, the bootloader will first derive the attestation key and initialize the MMT engine. The MMT meta-zone will also be initialized in this phase. After all these early configurations, the bootloader will load and calculate the measurement of the secure monitor. The MMT engine performs integrity and encryption protection for monitor's memory as well. After this, the secure monitor takes control and loads the Linux kernel as its payload.

The monitor provides both host-side and enclave-side interfaces for enclave management and runtime supporting. As for the host-side interfaces, they consist of *create_enclave*, *run_enclave*, *attest_enclave*, etc. As for enclave-side interfaces, they are mainly used as ocall functions (e.g., *enclave_mmap*, *enclave_sys_write*) and inter-enclave calls (e.g., *enclave_call*, *asyn_enclave_call*). Besides the enclave management, the monitor also takes control of configuring Guarded Page Table and Mountable Merkle Tree (e.g., *reg_hptarea_start*, *reg_hptarea_size*), setting page status bitmap and allocating secure memory. Monitor guarantees that all these secure-sensitive configurations are correct and will not be compromised by an attacker.

Linux kernel. We extend the Linux kernel (version: 4.4.0/5.10.2) to support PENGLAI. There are two major modifications: (1) HPT Area allocator, (2) hijacking each PTE settings. Firstly, after memory management is initialized, the kernel will allocate a contiguous physical memory as HPT Area and copy *init_pt* into it. A dedicated allocator will manage all pages in the HPT Area, and is responsible for each page table allocation. To distinguish huge page entries and 4KB page entries, the HPT Area is divided into three sub-areas: PGD, PMD and PTE sub-areas (MMU checks each page table entry's location according to these sub-areas). HPT Area allocator must assign the entries of page tables in the corresponding sub-areas as well. Currently, we have reserved enough memory as HPT Area, which can map all pages at



(a) Host-Enclave IPC (b) Enclave-Enclave IPC (c) Communication Link

Figure 8: Ownership transfer-based communication. Figure (a) and (b) show the message passing in PENGLAI for both host-enclave communication and enclave-enclave communication. Figure (c) presents the communication link established in PENGLAI. The number represents the order of the links.

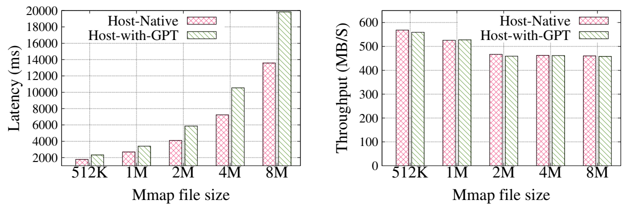
once. Normally, the reserved memory for HPT Area will not affect memory utilization, as page table pages are increased along with used pages. Dynamically adjusting the size of the HPT Area is in our future work.

Secondly, the kernel will redirect the setting operations of each page table entry to the secure monitor. Secure monitor can distinguish 2MB/4KB page entries (according to the sub-areas) and perform the security check of the target address (host cannot map any secure memory).

Server enclave. Despite the enclave and shadow enclave, PENGLAI also implements another type of enclave called *server enclave* to achieve enclave chain, which is common in serverless scenarios. A server enclave does not have running context (e.g., time slice, ocall handler) but inherits it from other enclaves. Hence, the server enclave cannot run alone. When creating a server enclave, it needs to be assigned a unique name as its identification. Other enclaves can acquire the handle of this server enclave with its unique server name. Besides, the server enclave can also perform partial functionalities of OS. For example, we can run a file system server in a server enclave to handle all FS-related requests. Separating the OS functionalities from the untrusted OS to the trusted enclave server can mitigate the risk of Iago attack [41] issued by untrusted privilege.

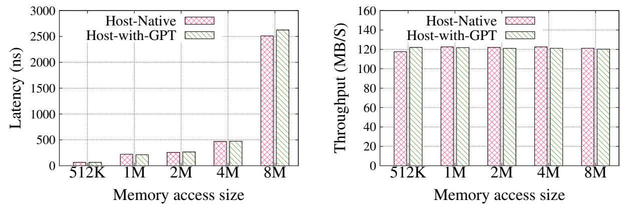
IPC. PENGLAI supports IPC between the enclave and server enclave (E-E), host and enclave (H-E), which is based on two mechanisms: shared memory and relay page [49]. Shared memory is the basic communication method. The secure monitor can map shared memory to both host and enclave, or enclave and server enclave. Relay page is a novel communication mechanism, as shown in Figure 8, and the secure monitor ensures that a page can be mapped for only one owner simultaneously. This mechanism can reduce security issues like TOCTTOU (Time-of-check-to-time-of-use) between E-E and H-E, and achieve zero-copy communication. PENGLAI can also support both synchronous and asynchronous IPC between enclaves. As for synchronous IPC, the caller enclave will wait for the callee enclave to return. As for asynchronous IPC, the caller enclave will return immediately, and arguments will be passed to the callee enclave when it starts to run.

SDK. PENGLAI provides an SDK (i.e., kernel driver, host-



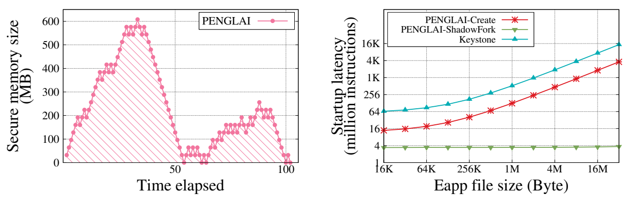
(a) Mmap latency. **(b) Mmap bandwidth.**

Figure 9: GPT performance on mmap. Test the latency and bandwidth of mmap operations, *Host-Native* represents the native linux kernel without GPT extension, *Host-with-GPT* represents the modified linux kernel with GPT extension.



(a) Memory latency. **(b) Memory bandwidth.**

Figure 10: GPT performance on memory access. Test the latency and bandwidth of memory access operations, *Host-Native* represents the native linux kernel without GPT extension, *Host-with-GPT* represents the modified linux kernel with GPT extension.



(a) Scalability of secure memory. **(b) Startup latency.**

Figure 11: Scalability and startup. Figure (a) shows the variation of secure memory size when creating 100 enclaves concurrently. Figure (b) compares the startup latency with different enclave file sizes.

side library and enclave-side library) to help users manage and develop enclave applications. The driver enables the untrusted host kernel (Linux) to interact with the secure monitor, and manage enclaves via interfaces provided by the monitor. The host side library abstracts ioctl interfaces from the driver and provides APIs to manage the enclave (e.g., create, run and attest enclave). The enclave side library is combined with modified Musl LibC (turns system calls to ocall or redirects to server enclaves) and Eapp library (e.g., IPC interfaces). Hence, PENGLAI can support unmodified POSIX applications. Besides, PENGLAI also integrates some useful libraries into enclave SDK, such as wolfssl [20] and PSA storage API, which are frequently used.

Formal verification. Currently, we are working on formal verification of PENGLAI. The approach is based on symbolic execution and bounded model checking via the state-of-the-art framework, Serval [79]. We have verified the code of the communication module. Verification on others is in progress.

6 Evaluation

6.1 Methodology

We implement PENGLAI based on the open-sourced RISC-V [100] implementation: SiFive Freedom U500 [10] on the Xilinx VC707 FPGA board. We present several microbenchmarks that evaluate the scalability metrics (i.e., startup latency, GPT overhead and the number of concurrent enclaves). We select four benchmarks, SPECCPU, Redis, RV8 and Coremark, to evaluate the overall performance of PENGLAI. We also compare PENGLAI with Keystone (as state of the art) and Linux (as the ideal performance). To evaluate the performance of MMT, we implement the MMT on the GEM5 [32] (RISC-V), and port the state-of-the-art integrity protection schemes: SGX integrity approach (SIT) and VAULT [92] on the GEM5. With the same emulation environment, we can make a fair comparison of these different approaches.

6.2 Microbenchmarks

Guarded Page Table performance. We first evaluate Guarded Page Table on LMbench [7] to gain the overhead for memory-related operations. LMbench can calculate the latency and bandwidth of memory mapping and memory access in the UNIX system. As shown in Figure 9 and Figure 10, Guarded Page Table will introduce 26%∼46% overhead for mapping latency, as each page mapping operation needs to be checked by the secure monitor. However, Guarded Page Table will not sacrifice the bandwidth of the mapping operations. As for memory access, Guarded Page Table will not affect memory access performance (both latency and throughput). The experimental results demonstrate that Guarded Page Table only incurs extra overhead in page mapping (e.g., mmap, munmap, mprotect), and will not impact the performance of memory access. As page mapping operations are infrequent, Guarded Page Table only introduces minor overhead for the whole system in most cases. Indeed, the monitor only inspects the page status with its bitmap during page mapping, so the extra overhead is minor compared with other page table-based isolation techniques [48, 54] (see §6.3).

Scalability of secure memory. To evaluate the scalability and flexibility of the PENGLAI enclave, we construct a test case to randomly create and run enclaves on the FPGA board, and calculate the total secure memory size in the monitor. As shown in Figure 11 (a), secure memory size could be very small when the system does not run any enclaves (2MB and less) and can scale to 600MB (1GB total memory in FPGA) when there are lots of concurrently running enclaves. This is a significant breakthrough over traditional fixed partitioned secure memory design, which usually needs to make a tradeoff between host memory size and enclave memory size.

Furthermore, with the same configuration, PENGLAI can achieve up to 1,000 concurrently running enclaves on the

FPGA board (1GB memory). It is possible to boot more enclave instances when the device has more resources. Hence, Guarded Page Table-based secure memory management can achieve scalability and flexibility.

Startup latency. We evaluate the startup latency of enclaves in PENGLAI and Keystone using the different sizes of enclaves. We compare two approaches of PENGLAI: normal startup (enclave create) and fast startup (shadow fork). The baseline is a traditional booting solution that loads an eapp file into the memory, prepares the resources, and verifies the measurement. PENGLAI shadow fork leverages shadow enclave to spawn a new instance, which can boost the procedure. To further compare with Keystone, we configure Keystone to use the minimum size of their eyrie runtime. We run both systems on Qemu and use the number of executed instructions (icount enabled) to represent the performance. The result is shown in Figure 11 (b). Compared with the baseline, shadow fork can achieve 4x–989x speedup (from 16KB to 32MB size). Keystone is orders of magnitude slower than PENGLAI with shadow fork, as it needs to calculate enclave measurement and prepare a new PMP region as well as load runtime in the supervisor mode.

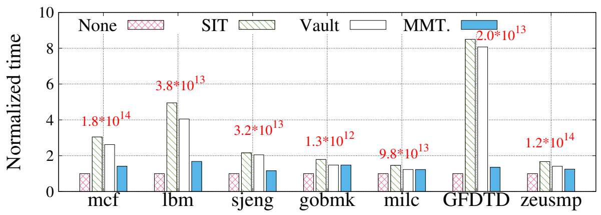
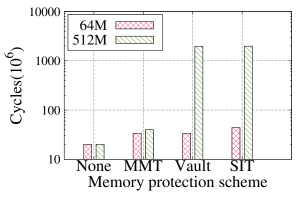


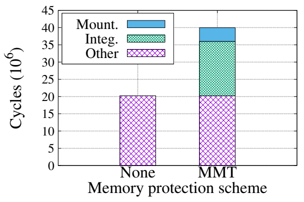
Figure 12: SPECCPU with different integrity protection schemes. *None* represents the ideal performance (w/o integrity and encryption protection). The memory costs of gobmk and milc are less than 128MB, so it will not trigger the swap mechanism in SGX and VAULT. The red numbers are the concrete execution time (cycles) of the "None".

Memory integrity and encryption. We evaluate the performance of memory integrity and encryption using SPECCPU benchmark and compare PENGLAI with VAULT and SGX. We retain the same SoC resource for all implementations (SoC storage in each integrity scheme can only protect 128MB memory). We want to demonstrate that with the same hardware resource, MMT can achieve the best performance as well as minimum memory overhead. The major reason to choose SPECCPU benchmark is that most of the related work like BMT [85], SIT [55] and VAULT [92] also use this benchmark to measure runtime overhead.

We implement all of the four systems on GEM5. As shown in Figure 12, the performance of PENGLAI is much better than the two baselines. Take milc as an example. The runtime overhead reduces from 2.05x (in SGX) and 1.60x (in VAULT) to 0.40x (PENGLAI's MMT). This is because 128MB memory is insufficient in this case. Thus, both SGX and VAULT need to swap and encrypt pages from secure memory to non-secure



(a) Random memory access.



(b) MMT performance breakdown.

Figure 13: Worst case performance for Mountable Merkle Tree. 64MB and 512MB in figure(a) mean the memory size for random access. *Mount.* represents the cost of the mount/unmount operations, *Integ.* represents the cost of the memory integrity check, *Other* represents original runtime cost.

memory, which can bring 10x overhead in some memory-intensive cases [92]. Instead, MMT can mount corresponding SubTrees to protect more than 128MB secure memory with less than 1% extra overhead (mounting only costs about 300 cycles). We also test the performance of MMT when the memory used is less than 128MB (gobmk, milc). The result shows that MMT will not bring extra overhead when all SubTrees can be mounted in SoC.

To further evaluate the worst-case performance of the MMT, we test the MMT and other memory protection schemes on a random memory access benchmark, as shown in Figure 13(a). We choose two different memory access ranges: 64MB and 512MB. The first one is smaller than the capacity of SoC-protected memory (128MB), while the second is larger. As for 64MB memory size, MMT incurs 67% overhead and SIT incurs 118% overhead. The performance improvement mainly comes from the optimized tree structure compared with MMT and SIT. As for 512MB memory size, MMT incurs 0.97x overhead and SIT incurs 98x overhead. Copying and encrypting the pages inside SoC-protected memory into the normal memory causes an enormous overhead in SIT (takes up 97% of total runtime cycles). This result is also validated by SCONE [28] — *"random memory access beyond available EPC may cause an overhead of three orders of magnitude"* and the real SGX-enable machine (Intel Core i7-7567U @ 3.5GHz, 300x overhead for random access of 512MB memory). MMT significantly reduces the overhead of page copying and encryption by the SubTree mounting mechanism. To further calculate the overhead of mount/unmount operations, we inventory the performance breakdown of the MMT. For random memory access, mount/unmount operations only take up ten percent of the total execution time and twenty percent of the integrity protection cost. Meanwhile, each mounting operation costs about 300 cycles on average. Hence, the mount/unmount overhead is much less than the prior art, even in the worst-case situation.

Costs of cache line locking. We perform a microbenchmark to evaluate the performance costs of cache line locking. The cache configuration of our FPGA implementation is 64 cache

Table 2: Costs of cache line locking. PENGLAI runs on FPGA.

	Latency (Kcycles)
Cache line locking	66.976
Normal	56.888

sets, 4 ways and 64-Byte cache line. The test case will sequentially read and write 16KB contiguous memory. We compare the end-to-end latency of two systems, PENGLAI enabling locking (using a single way) and PENGLAI disabling locking. As shown in Table 2, the locking can cause 17.73% overhead in the case. Although the costs are non-trivial, PENGLAI will only enable the locking for a critical section, which can significantly mitigate the overheads.

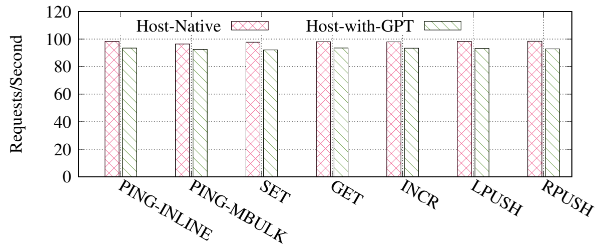


Figure 14: Redis benchmark suite. Evaluate the performance of GPT on the Redis benchmark. *Host-Native* means the native linux kernel. *Host-with-GPT* means the modified linux kernel with GPT extension.

6.3 Benchmark Suites

Redis benchmark suite. We use Redis benchmark suite to evaluate the worst-case performance of Guarded Page Table. As Redis performs as an in-memory database, it needs to call map/unmap operations frequently. Prior art also uses Redis to evaluate the performance degradation due to the mapping overhead (Shadow Page Table may incur 80% overhead on Redis [48]). As shown in Figure 14, Guarded Page Table only introduces 5% overhead in SET requests and 6% overhead in GET requests, which significantly mitigates overhead compared with Shadow Page Table. The main optimization comes from the fact that the monitor only checks page status in its bitmap. On the contrary, Shadow Page Table needs to re-construct the combined page table (traverses the host page table and extended page table), which is much more complicated and time-consuming.

RV8 benchmark suite. We use RV8 benchmark suite to answer two questions: 1) whether the isolation will incur performance overheads to enclaves, and 2) whether the Guarded Page Table hardware extensions will cause performance degradation on CPU-intensive applications. We port the benchmarks to PENGLAI, with 85LoC modifications to use PENGLAI API. In addition, we run the benchmark suite in the host kernel with two settings. One is that the host OS running in unmodified hardware without HPT Area extension, and the other is that the OS running in PENGLAI hardware with Guarded Page Table extension.

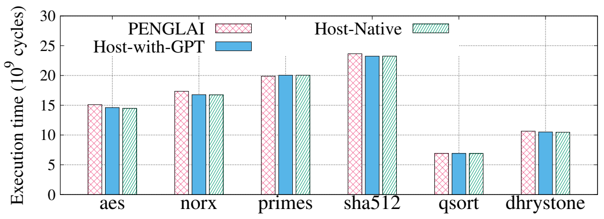


Figure 15: RV8 benchmark suite. PENGLAI can achieve almost the same performance as native Linux. Guarded Page Table only incurs minor overhead for host applications.

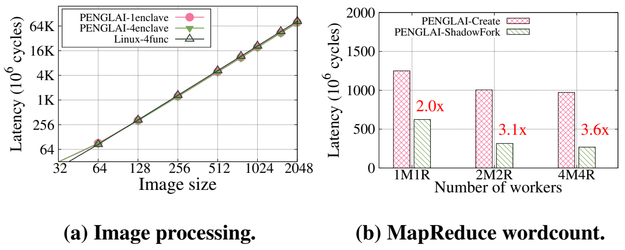


Figure 16: Evaluation of case studies. Figure (a) evaluates image processing on PENGLAI and linux. Figure (b) evaluates MapReduce performance with multiple workers.

As shown in Figure 15, the performance overheads in PENGLAI are <4.3% in all the cases and 1.7% on average, and Guarded Page Table will not affect the performance of CPU-intensive applications (the overhead is negligible compared with the GPT and Native). The overhead in PENGLAI is mainly caused by memory allocation in enclaves, which is slower than the host, as the monitor will dynamically allocate secure memory and perform the security check.

Coremark. We port Coremark with 43LoC modifications to meet PENGLAI API, and take the native Linux as our baseline and run Coremark in two systems on our FPGA board. The result shows that the score for native Linux is 2,018 and the score for PENGLAI is 2,049, which proves the strong isolation provided by PENGLAI will not hurt the performance of CPU-intensive applications.

6.4 Case Study: Serverless Computing

Existing serverless platforms [1, 3–5] use processes, containers, or VMs to isolate each function. In the case study, we try to illustrate the possibility of isolating serverless functions using enclaves (with higher security assurance) and analyze the performance impacts. We choose a representative serverless application, image processing [6, 25].

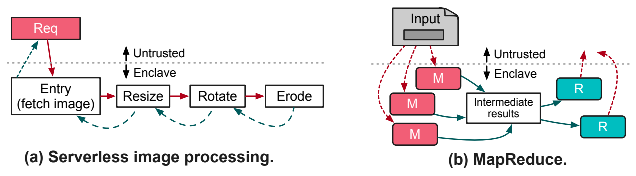


Figure 17: Case studies using PENGLAI.

As shown in Figure 17 (a), the image processing application is composed of four functions, and each function is running inside a different enclave. The application is triggered when an untrusted user issues a request (with an image file). The function "Entry" is fired to get the command and fetch the host's image file through untrusted shared memory. Then, the image file will be passed into the subsequent handling functions, i.e., "Resize", "Rotate" and "Erode". Finally, the resulting image is returned to the untrusted host. Since the images may contain sensitive information, it is necessary to protect them in enclaves. The workload is ported from AWS-examples repository [6] to a C version using Sod library [19].

We evaluate the performance of image processing in PENGLAI with two settings, "1enclave" (four functions in one enclave) and the default "4enclave" (four enclaves). We implement a baseline in native Linux, using four functions running in four processes. We carefully tune the Linux performance to be state-of-the-art, i.e., using a mixed approach of the socket (for timely notification) and shared memory (for zero-copy data transfer). The input image size for the evaluation is from 32x32 to 2048x2048. The result is shown in Figure 16 (a). Compared with Linux, PENGLAI can achieve even better performance (7%–9% improvement when image size is larger than 128x128), as PENGLAI's communication does not include kernel scheduling costs. Moreover, benefiting from efficient communication, using four functions in PENGLAI incurs minor costs. This can motivate more modular applications to be deployed in enclaves.

6.5 Case Study: Secure MapReduce

MapReduce [47] is a popular programming model in the cloud for data processing, and the recent study [67] shows that even a single machine can be competent with large-scale data processing. Since the processed data may contain sensitive information, VC3 [87] and Civet [96] are proposed to enhance the security with SGX enclaves. However, using SGX to run MapReduce workloads has some limitations: first, it needs a long latency to boot a worker enclave (i.e., mapper or reducer); second, it needs to load redundant enclave code into secure memory.

PENGLAI overcomes the issues with shadow fork, as shown in Figure 17 (b). Each mapper or reducer can instantiate itself from a pre-prepared shadow enclave, as they all have the same processing logic. With shadow fork, a MapReduce scheduler running in the host can instantiate multiple workers into different enclave instances with low startup latency and significantly save secure memory when running all workers in enclaves. During the processing, the mapper nodes invoke the map function on the input and produce intermediate key-value pairs. All the intermediate results are saved in memory and distributed to reducer nodes. The reducer nodes invoke the reduce function to produce the final result and return it to the host. We have implemented a prototype of MapReduce in PENGLAI with two settings: PENGLAI-Create and PENGLAI-ShadowFork. PENGLAI-Create creates each worker enclave using normal enclave creation, and PENGLAI-ShadowFork leverages the shadow enclave to fork a new worker instance. As shown in Figure 16 (b), PENGLAI-ShadowFork can achieve 2.0x lower latency over PENGLAI-Create when both systems use one mapper and one reducer. If we create more workers for the same job, PENGLAI-ShadowFork can gain 3.6x speedup compared with the PENGLAI-Create (4 mappers and 4 reduces, on a 4-core machine). The speedup mainly comes from the fast startup with shadow fork. As for normal startup, PENGLAI calculates enclave measurement every time. The tremendous attestation overhead will significantly affect the performance in the multi-workers situation. However, as shadow fork can reduce the overhead of secure memory initialization, enclave measurement is not the bottleneck for the whole job. Hence, PENGLAI-ShadowFork can gain better performance improvement with more worker enclaves.

6.6 Hardware Costs

We use Vivado [11] tool to generate the hardware and get the report of resource utilization in FPGA. The report shows that the overall hardware costs are small (0.56%–0.81% in LUT and 0.00% in RAM) over the original resources (the RISC-V core). It means that the extensions incur small costs, which is essential to add the extensions into real hardware.

7 Discussion

Architectures assumptions. Although the implementation of our prototype is based on RISC-V, the design is independent of specific architectures and can be adopted by other enclave systems. We highlight two major assumptions. First, the ISA should have a privileged level higher than the OS and hypervisor. This is a reasonable assumption for the prevailing ISA; cases include RISC-V's machine mode, ARM's EL3 mode, etc. Second, to support fine-grained memory management, the CPU shall have an MMU module, which is common in modern high-performance cores.

Security discussion. PENGLAI is designed to provide the same (or stronger) security guarantees compared with prior work. Besides the isolation and integrity protection mentioned above, PENGLAI can also defend against following attacks.

- **Controlled-channel attacks.** PENGLAI allows enclaves to validate the presence of some expected mappings, similar to Autarky [81]. The monitor will verify all these mappings when an enclave is invoked for the first time and check the validity when the OS changes the mapping. As the OS cannot directly access the enclave page tables, it cannot perform controlled-channel attacks by monitoring the access/dirty bits in page tables.

- **Cache-based side-channel attacks.** Existing enclaves still suffer from the cache-based side-channel attacks [74, 82, 107, 109, 110] or incur high overhead to solve it [45, 53, 75]. Our on-demand cache line locking mechanism (§4.4) defends the attacks with minor costs. A recent work, CURE [21], adopts a similar approach with on-demand partition cache. CURE binds the cache ways with certain enclave ID, while PENGLAI assigns cache ways to each core with less hardware modification, and is more suitable for scenarios like shared memory.

- **Other attacks.** There are some attacks caused by specific CPU bugs, including Foreshadow, Spectre [65], Meltdown [72], etc. The defense mechanisms are orthogonal to our design. We also believe that PENGLAI's monitor-assisted enclave design is more suitable for handling the emerging attacks than HW-based enclave systems, which are hard to update after release.

8 Related Work

Secure processor assisted enclave architecture. Some prior work uses a secure processor to support the trusted execution environment [12, 33, 39, 42, 44, 53, 70, 71, 85, 90, 104]. The Security-enhanced processor integrated with encryption and integrity engine can support compartments that are immune to both modification and observation. XOM [70, 71], SecureME [44] and SecureBlue++ [33] allow the trusted user processes running on the untrusted OS, as OS cannot obtain the plaintext content of the process. SEV and Hyper-Coffer [104] allow VMs to run on an untrusted hypervisor. However, such a method using encryption to isolate memory space brings non-negligible overhead, and the key management and traditional integrity scheme may restrict the enclave size and number.

Memory integrity scheme. Several different schemes have been proposed for memory integrity protection [38, 78, 85, 91, 92]. BMT [85] introduces the counter-based message authentication algorithm into the integrity protection scheme and reduces in-memory overhead. Bastion [91] unifies the integrity of in-memory pages and on-device pages. VAULT [92] adopts the various arity for different levels of tree nodes. However, all these prior work do not solve the inherent overhead in memory and SoC, and are not scalable at all. PENGLAI proposes the MMT with a mounting mechanism, which can reduce both on-die and in-memory overhead to achieve scalable memory integrity protection.

Virtualization-based isolation. Virtualization-based isolation [42, 60, 62, 91, 104, 111] has been researched in past decade. They rely on hypervisor to enhance the isolation among VMs using techniques like shadow page table [94, 98], nested/extended page table [31, 97], or HLAT [8]. These techniques have similarities to our Guarded Page Table—we both rely on higher privileged software to validate memory mappings. However, nested virtualization usually introduces non-trivial performance costs, e.g., tests show that the shadow PT can incur 40% overhead in Memcached [54] and 80% overhead in Redis [48], and Nested page table also causes 20% overhead in Memcached [54]. Shadow paging needs to re-construct shadow page table costly, and nested/extended page table incurs extra overhead due to 2-level page table walker during the TLB miss. However, PENGLAI proposes the Guarded Page Table and achieves page-grained isolation without introducing high performance overhead (only 5% even for memory-intensive benchmark: Redis). What's worse, the hypervisor and cloud service providers may not be trusted in cloud scenarios. Recent works utilize TEE techniques to propose secure VM, e.g., AMD SEV [12, 27, 61], Intel TDX [16] and vTZ [58], which can protect the VMs from the untrusted hypervisor. Nevertheless, the protection has defects, e.g., both SEV and TDX cannot defend against physical rollback attacks, and vTZ does not consider physical memory attacks. Also, the secure VM suffers the same performance degradation as the traditional VM due to the memory virtualization overhead. Compared with virtualization-based isolation methods, PENGLAI achieves better security guarantees, higher performance and scalability.

Cross-zone communication. Cross-zone communication or IPC has been extensively researched for microkernel and user-level processes [30, 49, 63, 95, 101]. SCONE [28] and HotCalls [103] optimize the host-enclave communication in SGX using asynchronous approaches, e.g., polling and shared untrusted memory. However, they are not suitable for E-E communication and tend to waste the CPU cycles. XPC [49] has proposed the ownership transfer based communication and reduced the remapping overhead. PENGLAI shares the same idea of XPC, but overcoming new challenges to transfer pages crossing the boundary between the secure and non-secure world. It outperforms existing enclave systems with the zero-copy and secure data transfer mechanism for both E-E and E-H communication.

9 Conclusion

This paper has presented a hardware-software co-design of scalable memory protection based on the PENGLAI enclave system. Our evaluation shows that PENGLAI can significantly optimize enclave number, secure memory capacity with integrity protection, enclave startup latency, as well as resource flexibility for both microbenchmarks and real-world applications.

10 Acknowledgments

We sincerely thank our shepherd Edouard Bugnion, Andrew Baumann and anonymous reviewers for their insightful suggestions. This work is supported in part by National Key Research and Development Program of China (No. 2020AAA0108500), China National Natural Science Foundation (No. 61972244, U19A2060, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Yubin Xia is the corresponding author.

References

[1] Apache openwhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/. Referenced December 2018.

[2] Arm corelink tzc-400 trustzone address space controller technical reference manual. https://developer.arm.com/documentation/ddi0504/c/. Referenced November 2020.

[3] Aws lambda - serverless compute. https://aws.amazon.com/lambda/. Referenced December 2018.

[4] Azure functions serverless architecture. https://azure.microsoft.com/en-us/services/functions/. Referenced December 2018.

[5] Google cloud function. https://cloud.google.com/functions/. Referenced December 2018.

[6] lambda-refarch-imagerecognition. https://github.com/aws-samples/lambda-refarch-imagerecognition. Referenced December 2019.

[7] Lmbench. http://lmbench.sourceforge.net/, 2005. Referenced Nov 2020.

[8] Hlat. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, 2018. Referenced May 2018.

[9] Intel 64 and ia-32 architectures software developer manuals. https://software.intel.com/en-us/articles/intel-sdm, 2018. Referenced August 2018.

[10] Sifive. https://www.sifive.com/, 2018. Referenced November 2018.

[11] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html, 2018. Referenced August 2018.

[12] Amd secure encrypted virtualization (sev) - amd. https://developer.amd.com/sev/, 2019.

[13] Keystone | an open framework for architecting tees. https://keystone-enclave.org, 2019.

[14] Amd epyc 7002 series processors. https://www.amd.com/en/processors/epyc-7002-series, 2020. Referenced Aug. 2020.

[15] The intel sgx memory encryption engine. https://software.intel.com/content/www/us/en/develop/blogs/memory-encryption-an-intel-sgx-underpinning-technology.html, 2020. Referenced May 2020.

[16] Intle tdx. https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html, 2020. Referenced Nov 2020.

[17] RISC-V Proxy Kernel. https://github.com/riscv/riscv-pk, 2020. Referenced May 2020.

[18] riscv/opensbi: Risc-v open source supervisor binary interface. https://github.com/riscv/opensbi, 2020.

[19] Sod - an embedded, modern computer vision and machine learning library. https://sod.pixlab.io, 2020. Referenced May 2020.

[20] wolfssl embedded ssl/tls library. https://www.wolfssl.com, 2020.

[21] CURE: A security architecture with customizable and resilient enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association.

[22] Intel scalable software guard extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html, 2021. Referenced Mar 2021.

[23] Intel software guard extensions. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html, 2021. Referenced Mar 2021.

[24] Penglai enclave. https://github.com/Penglai-Enclave, 2021. Referenced Mar 2021.

[25] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, pages 923–935, 2018.

[26] Tiago Alves. Trustzone: Integrated hardware and software security. *White paper*, 2004.

[27] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020. Referenced Aug. 2020.

[28] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind,

Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[29] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[30] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight remote procedure call. *ACM SIGOPS Operating Systems Review*, 23(5):102–113, 1989.

[31] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, 2008.

[32] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[33] Risk Boivie and Perter Williams. Secureblue++: Cpu support for secure execution. pages 1–9. IBM, IBM Research Division, 2012.

[34] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the" micro" back in microservice. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 645–650, 2018.

[35] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[36] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 33–43, 2019.

[37] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, pages 1–13, 2016.

[38] David Champagne, Reouven Elbaz, and Ruby B Lee. The reduced address space (ras) for application memory authentication. In *International Conference on Information Security*, pages 47–63. Springer, 2008.

[39] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[40] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[41] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.

[42] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 42(2):2–13, 2008.

[43] Yi-Lin Cheng, Ching-Chi Lin, Pangfeng Liu, and Jan-Jan Wu. High resource utilization auto-scaling algorithms for heterogeneous container configurations. *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 143–150, 2017.

[44] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing*, pages 108–119, 2011.

[45] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[46] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[48] Boris Teabe Djomgwe, Peterson Yuhala, Alain Tchana, Fabien Hermenier, Daniel Hagimont, and Gilles Muller. (no) compromis: Paging virtualization is not a fatality. In *VEE 2021-17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–12, 2021.

[49] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 671–684. ACM, 2019.

[50] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[51] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, New York, NY, USA, 1995. ACM.

[52] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305. ACM, 2017.

[53] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8, 2012.

[54] Jayneel Gandhi, Mark D Hill, and Michael M Swift. Agile paging: exceeding the best of nested and shadow paging. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016.

[55] Shay Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. https://eprint.iacr.org/2016/204.

[56] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[57] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11, 2013.

[58] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vtz: Virtualizing arm trustzone. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017.

[59] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–32, 2018.

[60] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 272–283. IEEE, 2011.

[61] David Kaplan. Protecting vm register state with sev-es. *White paper, Feb*, 2017.

[62] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 350–361, 2010.

[63] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[64] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 427–444, 2018.

[65] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[66] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 393–404. IEEE, 2009.

[67] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[68] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[69] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *Proceedings of the 48th International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14–19, 2021*. IEEE, 2021.

[70] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *Acm Sigplan Notices*, 35(11):168–177, 2000.

[71] David Lie, Chandramohan A Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, 2003.

[72] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[73] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[74] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[75] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324, 2013.

[76] Anupama Mampage, S. Karunasekera, and R. Buyya. Deadline-aware dynamic resource management in serverless computing environments. 2020.

[77] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[78] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[79] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 225–242, 2019.

[80] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[81] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[82] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[83] Daniel Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, 2003.

[84] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[85] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 183–196. IEEE, 2007.

[86] Aakanksha Saha and Sonika Jindal. Emars: Efficient management and allocation of resources in serverless. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 827–830, 2018.

[87] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.

[88] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[89] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020.

[90] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368, 2003.

[91] Jakub Szefer and Ruby B Lee. Architectural support for hypervisor-secure virtualization. *ACM SIGPLAN Notices*, 47(4):437–450, 2012.

[92] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 665–678, 2018.

[93] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 44–54, 2019.

[94] Eric P Traut, Matthew D Hendel, and Rene Antonio Vega. Enhanced shadow page table algorithms, November 20 2007. US Patent 7,299,337.

[95] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[96] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E Porter. Civet: An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[97] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[98] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[99] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[100] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[101] Robert NM Watson, Robert M Norton, Jonathan Woodruff, Simon W Moore, Peter G Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, et al. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 2016.

[102] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[103] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. *ACM SIGARCH Computer Architecture News*, 45(2):81–93, 2017.

[104] Yubin Xia, Yutao Liu, and Haibo Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *HPCA*, pages 246–257, 2013.

[105] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 859–874, 2017.

[106] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360. IEEE, 2017.

[107] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World*, page 0. IEEE, 2019.

[108] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2014:140, 2014.

[109] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[110] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[111] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 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*. ACM, 2011.

[112] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.

[113] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 28–41. IEEE, 2020.