HyperTEE: A Decoupled TEE Architecture with Secure Enclave Management

Yunkai Bai[†], Peinan Li[†*], Yubiao Huang[†], Michael C. Huang[‡], Shijun Zhao[†],
Lutan Zhao[†], Fengwei Zhang[§], Dan Meng[†], Rui Hou[†*]
[†]*Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS and School of Cyber Security, University of Chinese Academy of Sciences. Email: {baiyunkai, lipeinan, huangyubiao, zhaoshijun, zhaolutan, mengdan, hourui}@iie.ac.cn, [*]Corresponding author: Rui Hou and Peinan Li*
[†]*University of Rochester. Email: michael.huang@rochester.edu*
[§]*Southern University of Science and Technology. Email: zhangfw@sustech.edu.cn*

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00018

***Abstract*—Trusted Execution Environment (TEE) architectures have been deployed in various commercial processors to provide secure environments for confidential programs and data. However, as a relatively new feature against security threats, existing designs still face a number of problems. Exploiting the management vulnerabilities, attackers can disclose secrets via controlled-channel or micro-architecture side-channel attacks. To address these problems, this paper proposes a novel TEE architecture, named HyperTEE. In our architecture, enclave management tasks are decoupled from the original computing subsystem to a dedicated, physically isolated Enclave Management Subsystem (EMS). A properly architected EMS prevents current management vulnerabilities and offers more secure enclave communication. We implemented the HyperTEE prototype on the FPGA platform. Experiments show that HyperTEE only introduces less than 1% area overhead, and 2.0% and 1.9% performance overhead on average for enclaves and non-enclave workloads, respectively.**

I. Introduction

Trusted Execution Environment (TEE) is a crucial CPU security feature with widespread commercial adoption, including Intel SGX [1], [2] and TDX [3], AMD SEV [4], [5], ARM TrustZone [6], and CCA [7]. Its design goal is to establish a secure environment (enclave) dedicated to the protection of sensitive programs and data. Despite its increasing popularity, TEE remains a relatively new feature and is set against evolving security threats. As a result, it requires further enhancements to achieve robust security.

In TEEs, there are two categories of tasks: enclave management tasks and enclaves themselves. Typically, attacks targeting enclave themselves involve analysis on control flows and the deduction of secrets, which just compromises confidentiality. In contrast, attacks on management tasks pose significantly more severe security risks, as listed in Table I. Such attacks not only breach confidentiality but also undermines integrity through undetected modifications to enclave codes and data. Furthermore, they can compromise the availability of the entire TEE by intentionally justifying legitimate enclaves as untrustworthy and preventing their execution. Therefore, it is necessary to improve the security of enclave management tasks. Numerous attacks on management tasks have been revealed and they can be classified into two types:

TABLE I
Comparison of security risks.

Security Threats	Attack Enclave Management Tasks	Attack Enclaves Themselves
Compromise Confidentiality	Yes	Yes
Compromise Integrity	Yes	No
Compromise Availability	Yes	No

Attack Type 1: Microarchitectural side-channel attacks. In modern TEEs, management tasks are executed within an environment that is logically isolated but physically shared with untrusted environment. This exposes them to various microarchitectural side channels [8]–[24]. With these vulnerabilities, attackers can obtain critical keys in SGX attestation management tasks. Recent TDX design is also vulnerable to such threats. Generally, disclosure of sensitive data within individual enclaves just compromises their own confidentiality, but disclosure of attestation key of management task poses far more severe threats to the entire platform beyond confidentiality compromises. Such disclosure might allow attackers to: ① compromise integrity by tampering with enclave binaries, enabling it to bypass attestation check and thus carry out malicious operations. In contrast, merely breaching confidentiality of enclave themselves results in information leakage but can not cause malicious operations on them. ② compromise system availability by falsely declaring the platform untrustworthy to users, preventing the launch of any enclaves.

Attack Type 2: Controlled-channel attacks. In prevalent commercial TEE designs, enclave management tasks are deployed in untrusted operating systems (SGX) or hypervisors (SEV). This exposes opportunities for privileged attackers to exploit management tasks and take control over enclave execution, giving rise to numerous exploits known as controlled-channel attacks [25]–[33]. For instance, privileged attackers can disclose sensitive data of enclaves by observing three types of events in memory management: ① *Allocation-based attacks*: monitor on-demand memory allocation requests from enclaves [32]; ② *Page table management-based attacks*: compromise and observe access/dirty states in page table entry [25]–[31]; ③ *Page swapping-based attacks*: swap out

979-8-3503-5057-9/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00018

enclave pages and observe swap-in events by enclaves [32], [33]. Although some proposals employ trusted modules to perform these tasks, there still exist risks. In TDX module, page allocation and swapping can still be observed by untrusted hypervisor [34], leading to information leakage. Keystone proposes that enclaves perform these tasks, but attackers could potentially use malicious enclaves to attack the OS [32].

Upon examining existing TEE architectures and vulnerabilities, exploring alternative designs is crucial to provide secure enclave management tasks. Notably, several commercial processors have integrated dedicated cores to provide security services, such as Apple's Secure Enclave Processor (SEP) [35], Google's Titan [36], Qualcomm's Secure Processing Unit (SPU) [37], and AMD's Platform Security Processor (PSP) [38]. SEP, SPU, and Titan are designed primarily to facilitate vendor-specific security functions such as encryption services, biometric authentication, and pattern lock verification, but they do not offer secure management for user-programmed applications directly. PSP undertakes the encryption and attestation of confidential virtual machines (CVMs) in SEV architecture. However, most critical management tasks still rely on the untrusted hypervisor deployed on original computing cores, thus vulnerable to aforementioned attack threats [31], [39]–[41].

In this paper, we propose a novel decoupled TEE architecture, named HyperTEE: *enclave management tasks are offloaded to dedicated cores for security, while the enclaves are executed in original cores to maintain high performance and workload diversity.* With such a decoupled architecture, HyperTEE can effectively improve the security of enclave management tasks. In contrast, management tasks within TEE architectures such as TDX and SEV remain vulnerable to controlled-channel and microarchitectural side-channel attacks. The major contributions include:

- **Design of a physically decoupled TEE architecture.** HyperTEE consists of two subsystems: Computing Subsystem (CS) and Enclave Management Subsystem (EMS), utilizing original computing cores and dedicated cores respectively. Enclave management tasks are decoupled to EMS as enclave primitives. Applications within CS can invoke these primitives through HyperTEE APIs. Illegal cross-privilege invocation and forgery of primitive requests are eliminated by a trusted call gate, EMCall. Primitive requests are transmitted via a dedicated mailbox and protected against unauthorized access and timing side-channel attacks. The hardware of EMS, dubbed HyperTEE IP, is architected carefully to minimize complexity while maintaining security requirements. CS and EMS enforce a unidirectional isolation manner, allowing EMS to access CS memory and I/O resources, but not vice versa. In this way, controlled-channel and microarchitectural side-channel vulnerabilities on enclave management tasks are prevented. Our experiments show that the area overhead of HyperTEE IP is less than 1% of the whole SoC chip.
- **Design of an enclave memory management with secure allocation, page table management, and page swapping.** Enclave memory management tasks are deployed on EMS, but simply offloading is insufficient to ensure security. To thwart allocation-based, page table management-based, and page swapping-based attacks, we propose specific countermeasures: ① An enclave memory pool obscures on-demand page allocation requests from potential attackers. ② Each enclave is assigned a dedicated private page table that is protected as enclave memory, preventing unauthorized compromises and observation. ③ Enclave pages to be swapped are managed by EMS, which randomly selects the number and specific pages involved, thereby minimizing valuable traces. Besides, a bitmap-based enclave memory isolation is employed to prevent unauthorized access and facilitate non-contiguous enclave memory regions.
- **Design of an efficient enclave-to-enclave communication based on encrypted shared memory.** Communication based on enclave shared memory has become prevalent due to its high efficiency. But the management of enclave shared memory should be carefully designed to prevent potential security threats: Assigning encryption keys for the shared memory is crucial but the keys may be illegally shared or brute-force cracked; Shared pages may be maliciously mapped to unauthorized attacker applications, allowing them to read enclave secrets directly. Thus, in our design: ① EMS assigns dedicated keys for enclave shared memory, within which identity verification and registration authorization are employed to prevent key disclosure and brute-force attacks. ② EMS tracks the ownership of each shared enclave page and performs a series of access controls to prevent unauthorized access.
- **FPGA prototyping for performance and cost analysis.** We implemented the HyperTEE architecture on an FPGA-based RISC-V prototype system. Experiments show that HyperTEE introduces an average performance overhead of 2.0% and 1.9% for the enclave (RV8 and wolfSSL) and non-enclave processes (SPEC CPU2017 int), respectively.

II. Background

A. Trusted Execution Environment

Trusted Execution Environment (TEE) is designed to provide an isolated environment to protect sensitive applications and their data. Initially, TrustZone [6] provides a 'secure world' to execute all trusted applications. Subsequently, application-level TEEs like SGX enable individual applications to protect their data and code within enclaves. The latest development, VM-level TEEs such as SEV [4], [5], TDX [3], and CCA [7], extend this concept to secure entire virtual machines. Like in SGX, we call the trusted execution environment for each application as an *enclave*. To ensure the security of enclaves, various management tasks are necessary. Memory isolation is required to protect enclave data from being accessed by non-enclave or other enclaves. Remote attestation and local attestation are necessary to guarantee that enclaves are not tampered with and the execution platform is trusted. In addition, enclave memories are usually encrypted and measured to protect their confidentiality and integrity.

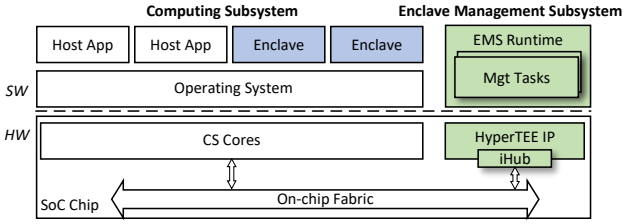


Fig. 1. Architecture overview of HyperTEE.

B. Threat Model

Our threat model is similar to existing TEE designs and assumes that attackers have the following capabilities:

- **Software attacks**. The attacker may compromise the privileged system software, including the operating system and hypervisor, to break the confidentiality and integrity of enclave codes and data. This can be achieved by constructing controlled channel attacks [25]–[31], [39], [40], [42], [43]. The attacker may construct microarchitectural side channels to disclose secrets in management tasks, such as attestation key [12], [19], [21], [24].
- **Physical attacks**. The attacker may carry out cold boot attacks to extract sensitive data from the memory [44]–[47]. Similar to SGXv2, SEV, and TDX, we do not consider physical memory replay attacks, which require expensive equipment and professional expertise.
- **Malicious enclaves**. The attacker may construct a malicious enclave to interfere with the execution of other enclaves or to break the isolation enforced by the privileged system software [14].

We do not consider intentional enclave secret leakage by enclave codes, or the security risks posed by unsafe enclave codes. Denial-of-service (DoS) attacks are also out of scope. For microarchitectural side-channel attacks on enclave execution, HyperTEE is at the same security level as existing TEEs.

III. HyperTEE Architecture

A. Design Overview

Deploying enclave management tasks within untrusted execution environment is demonstrated to pose severe security risks [12], [19], [21], [24]–[31]. To provide robust security, we propose the HyperTEE architecture, which divides the entire system into two subsystems: the original computing subsystem (CS) and the Enclave Management Subsystem (EMS), as illustrated in Figure 1. EMS is a standalone subsystem with its own dedicated hardware and software resources. Enclave management tasks are deployed on EMS to eliminate controlled channels and protect them against microarchitectural side-channel attacks. Enclaves themselves are still executed on CS to leverage the high-performance processing capabilities.

The primary hardware component of EMS is the HyperTEE IP, which contains EMS private core, memory, and necessary I/O devices to execute software management tasks and store corresponding management data. CS cores and HyperTEE IP are connected through an on-chip fabric, mediated by *iHub*. Considering that enclaves executed on CS may access any CS memory region, iHub supports both uni-directional isolation and secure management requests/responses to achieve efficient enclave memory accesses and minimize the attack surface. iHub allows uni-directional access to the entire CS memory space and I/O devices by EMS. Conversely, EMS private memory and its I/O devices remain invisible to CS.

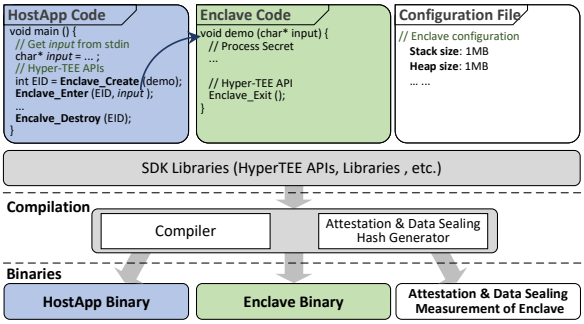


Fig. 2. Programming and compilation of HyperTEE.

TABLE II
HyperTEE primitives.

Mgt. Task	Primitive	Priv.	Semantics
Life Cycle Management	ECREATE	OS	Create an enclave
	EADD	OS	Load codes and data to an enclave
	EENTER	OS	Start executing an enclave
	ERESUME	OS	Resume enclave execution
	EEXIT	User	Exit enclave execution
	EDESTROY	OS	Destroy an enclave
Memory Management	EALLOC	User	Allocate enclave memory
	EFREE	User	Release enclave memory
	EWB	OS	Swap enclave memory
Communication Management	ESHMGET	User	Apply shared memory from EMS
	ESHMAT	User	Attach shared memory to enclaves
	ESHMDT	User	Detach enclave shared memory
	ESHMSHR	User	Share memory with an enclave
	ESHMDES	User	Destroy enclave shared memory
Key Management and Attestation	EMEAS	OS	Measure code and data of enclave
	EATTEST	User	Sign enclave and platform

B. Enclave Management Decoupling and Programming Model

HyperTEE adopts a programming model and compilation process similar to SGX, allowing an untrusted host application (HostApp) to use primitives for managing an enclave's environment. Differently, our primitives are implemented as software management tasks on EMS. They are invoked by what we call *EMCall*, similar to a remote procedure call (RPC). Although this paper primarily focuses on implementation for application-level TEE, VM-level TEE can be supported inherently, which is discussed in Section IX.

Programming model. As shown in Figure 2, programmers use HyperTEE APIs provided by the SDK to call enclave primitives. Such a call is translated into the RPC-like EMCall. In addition to preparing the HostApp and enclave codes, a configuration file is needed to declare the resource requirements of the enclave, including heap and stack memory sizes, etc. Upon compilation, executable files for HostApp and the enclave as well as the measurement value for the enclave are generated.

Trusted call gate: EMCall. EMCall is implemented at the highest privilege level on CS side (for example, machine mode

for RISC-V [48], EL3 privilege level for ARM [49], SMM mode for x86 [50]). Since EMCall interacts with sensitive information with EMS, it is included in the trusted computing base (TCB). In our current implementation, EMCall is in firmware. To guarantee its security, the integrity is checked during secure boot and it is protected in the highest privilege mode, preventing unprivileged accesses or interference from untrusted software. Additionally, the EMCall memory is protected with encryption and integrity checks.

Secure decoupling of enclave primitives. In HyperTEE, enclave management is performed through enclave primitives. These primitives are categorized into four types and listed in Table II, and all of them are decoupled to EMS. When CS software needs to invoke primitives, it does so through the trusted EMCall which assembles and transmits primitive requests to EMS in non-speculative mode. To prevent potential vulnerabilities, several security mechanisms are employed:

① Restrict cross-privilege primitive requests: Each enclave primitive is restricted to be invoked only within a specific privilege mode. EMCall checks the current privilege register during primitive invocation and blocks any cross-privilege request. ② Prevent primitive request forgery: For each primitive request, EMCall encapsulates the current enclave identification (enclaveID) as an argument. In this way, attackers cannot impersonate other enclaves or access the primitive responses to them. ③ Prevent illegal arguments with sanity check: Upon receipt of a primitive request, EMS conducts a sanity check on its arguments to ensure legitimacy, preventing maliciously crafted requests. ④ Atomically update CS registers during enclave context switches: While executing EENTER and ERESUME primitives, both enclave control structures and context registers have to be updated. Sensitive updates to enclave control structures are managed by EMS, whereas updates to CS registers, which are inaccessible to EMS, are handled directly by the EMCall. To prevent potential compromises or interference from untrusted CS interrupt handling, EMCall performs CS register updates atomically.

Secure handling of exception/interrupt in enclaves: During the enclave execution, any interrupts or exceptions encountered are initially processed by EMCall, which records critical information like the cause, program counter (PC), etc. Subsequently, according to the type of interrupt or exception, EMCall determines whether it is directed to EMS or CS. In our current implementation, exceptions related to memory management, such as page faults and misaligned memory accesses, are handled by EMS, while others, such as timer interrupts and illegal instructions, are responded by CS OS.

C. Enclave Management Requests and Responses

To request the enclave management, CS can send enclave primitive requests to EMS through a dedicated *mailbox* in iHub. Specifically, HostApp or Enclave on CS can call EMCall to transmit primitive requests to mailbox, which will trigger EMS to execute corresponding routine to read the requests. Figure 3 depicts the communication flow between CS and EMS. When a CS application initiates an enclave primitive

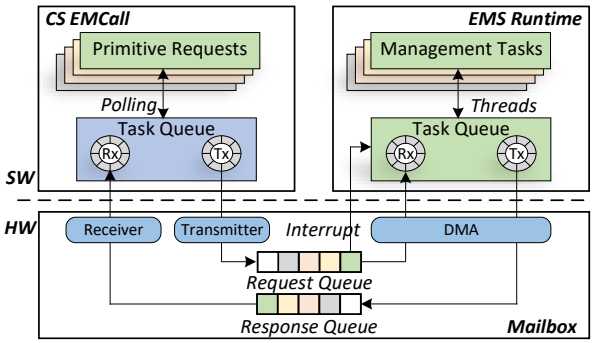


Fig. 3. Communication between CS and EMS.

request, EMCall generates request packets and stores them in a ring task queue for transmission (Tx). Once there is a request packet in Tx, a transmitter module automatically moves it to the request queue in mailbox and an interrupt is triggered to inform EMS. Then, EMS fetches the requests to its own task queue for receiving (Rx). As multiple requests may be invoked concurrently, EMS creates multiple threads to perform the management tasks. After the threads complete the requests, EMS generates response packets and sends them to CS via the response queue in mailbox. Due to potential vulnerabilities in the CS's interrupt handling module, EMCall employs a polling method to retrieve response packets securely.

Protect primitive transmission from unauthorized accesses. During the transmission procedure, both task queues in EMCall and queues in mailbox are invisible to CS. Each primitive request is bound with its response exclusively through a unique identification, and a request cannot access the other response packets. Notably, only primitive requests and responses are transmitted through the mailbox. Enclave private data are not required for enclave management tasks.

Prevent timing side channels on EMS. Though attackers are unable to execute their codes on EMS, they might attempt to deduce secrets by observing the response latency of EMS primitives. However, several measures in HyperTEE collectively introduce substantial noise, obfuscating precise observation and effectively thwarting potential attacks. In attacks that *target EMS management tasks*, from the perspective of CS attackers, EMS tasks are scheduled in primitive granularity, preventing any interference or intentional slowing to execute specific victim gadgets [8], [51]–[53]. Moreover, any potential observation is disturbed by the obfuscation during EMCall polling primitive responses. In cases where one *enclave might attempt to attack another*, additional complexities arise: ① Different enclave primitives sent to EMS are scheduled randomly on EMCall and transmissions on mailbox are non-interferable. ② When EMS receives these primitive requests, they are handled concurrently across multiple cores, stripping attackers of any influence over the execution order or timing.

D. Architecting the HyperTEE IP

The relationship between EMCall, EMS Runtime, and corresponding hardware components is shown in Figure 4. Given

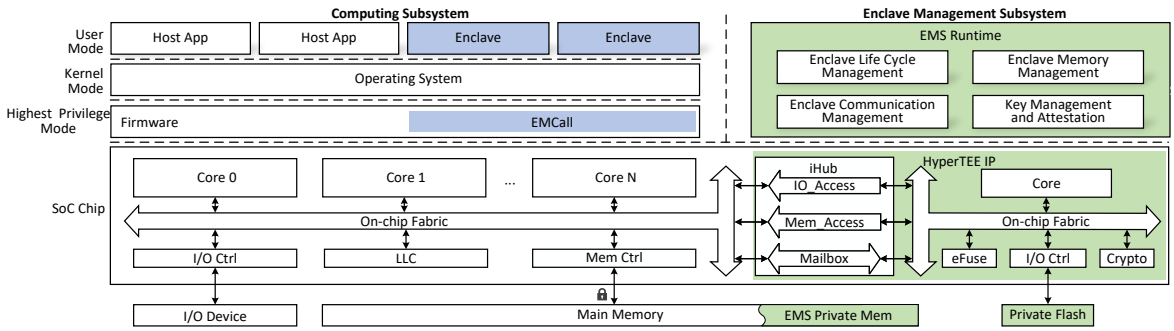


Fig. 4. Architecture details of HyperTEE.

that measurement, attestation, and many other encryption operations are necessary for enclave primitives, a crypto engine is deployed to accelerate these operations. To protect the EMS Runtime image from potential compromise, it is encrypted and hashed before being stored in a private flash. To ensure the security of the root key, it is programmed into a one-time programmable device eFuse during the manufacturing stage, which cannot be modified thereafter. To reduce the implementation overhead, we propose three optimizations:

① *Minimal complexity:* Given that the computation patterns of enclave management tasks are relatively fixed and simple, a design with minimal complexity is sufficient to deliver efficient services. We conduct experiments and the results in Section VII-B indicate that for a high-end embedded processor (no more than 4 cores), a single in-order core is sufficient to provide efficient enclave management. For a high-performance processor (32 or 64 cores), a dual-core out-of-order design is sufficient. This ensures that HyperTEE IP occupies only a minimal area of the entire SoC chip.

② *Unidirectional cache coherence:* Benefiting from the management features in HyperTEE, there is an opportunity to employ a unidirectional cache coherence protocol. The analysis is as follows: *i) Enclave private data:* These data locate in CS cache or memory and EMS does not need to access them in management scenarios, thereby eliminating the requirement of cache coherence. *ii) Enclave management-related data (e.g. enclave page tables, control structures):* Both CS and EMS may update these data. However, EMS performs updates only during the primitive execution, which is infrequent and involves less data. Therefore, a software-based method can be used to explicitly flush them to memory, allowing CS to access the latest data in subsequent accesses. When EMS requires these data, the existing cache coherence supported by on-chip fabric can provide the latest data to iHub. *iii) Private data of enclave management tasks:* CS cannot access these data and it does not load them into its private cache. Thus, when EMS accesses these data, it bypasses LLC and CS private cache and loads directly into EMS private cache, without the need for maintaining cache coherence with CS. The coherence between EMS private cache and memory is supported by iHub and on-chip fabric. In summary, CS cache responds to EMS data requests through existing hardware, while EMS cache does not respond to CS requests.

③ *Isolated EMS address space:* Although EMS has private memory from the design perspective, it can exclusively own part of CS memory in practice. During the first stage of SoC boot, the chip initialization logic can configure the address spaces of EMS and CS by the memory controller.

IV. Enclave Memory Management

In HyperTEE, enclave memory management tasks are deployed on EMS, ensuring the security of memory allocation and page table management. An isolation mechanism is proposed to protect enclave memory from untrusted operating systems, applications, and other enclaves. Besides, memory encryption and integrity protection are integrated.

A. Enclave Memory Allocation and Page Table Management

Prevent allocation-based controlled channel attacks with an enclave memory pool. Specifically, EMS proactively requests pages from CS OS and stores them in an enclave memory pool. When new requests arrive, they can obtain pages directly from this pool without notifying CS OS. This method conceals the allocation events effectively. Similarly, EFREE primitives release relevant pages to the pool. Before being returned to CS OS, EMS will zero these pages. Notably, the pool is dynamically enlarged when the number of used pages exceeds a threshold set by EMS. Furthermore, this threshold is randomized once the pool enlarges, complicating attempts by attackers to reverse-engineer the threshold.

To ensure the integrity of codes, all enclaves must undergo remote attestation before execution, requiring static allocation during creation. When the ECREATE primitive is invoked, EMS selects pages from enclave memory pool and then maps them to the corresponding enclave. Furthermore, EMS supports dynamic augment of enclave memory during execution. When an enclave requires more heap memory, the EALLOC primitive can be invoked. While encountering a page fault exception caused by a page miss, EMCall handles the exception and sends a request to EMS for memory allocation. Before being mapped, corresponding pages will be zeroed first.

Prevent page management-based controlled channel attacks by assigning a dedicated page table for each enclave. For each enclave, EMS maintains a dedicated enclave page table separate from the original page table. EMS handles the

page table management and creates virtual-to-physical address mappings. The page table is stored in enclave memory and inaccessible to both the enclave itself and any untrusted software, thus eliminating potential compromises and malicious observation [25]–[31]. While switching to the enclave context, EMCall updates page table base address register to reference the enclave page table.

Prevent swapping-based controlled channel attacks through concealing information of swapped enclave pages. In HyperTEE, CS OS cannot access enclave page tables. While performing enclave page swapping, CS OS must invoke EMS to execute EWB primitives to return the enclave pages to be swapped. EMS then selects a random number of pages from the enclave memory pool, encrypts them, clears their original bitmap bits, invalidates corresponding PTE, and returns the physical page addresses to CS OS. Then, relevant pages can be swapped out to persistent storage. It is noteworthy that during this process, swapping-based controlled channel attacks can be mitigated: ① The lack of visibility into enclave address mappings prevents attackers from precisely swapping out pages used by specific victim gadgets. ② EMS returns unused pages from the enclave memory pool, thus preventing direct swap-out of actively used enclave pages. ③ By randomly selecting multiple pages for swap-out and swap-in, EMS obscures access patterns that might reveal sensitive information.

Data movement between HostApp and Enclave. As the page table of enclave and its HostApp are kept separated, an enclave is unable to directly access data within the address space of HostApp. To allow data movement between HostApp and an enclave, HyperTEE supports mapping a shared memory region from HostApp to enclave. The size of the shared memory can be declared in the configuration file. Remote users can transmit encrypted sensitive data to HostApp, which then transfers them to the enclave through the shared memory.

B. Enclave Memory Isolation

Isolating enclave memory from untrusted software and other enclaves is essential for confidentiality. In HyperTEE, two isolation mechanisms are deployed.

Isolate enclave memory from untrusted CS software through hardware-based bitmap checking. To isolate enclave from untrusted memory, most TEE designs deploy permission check in page table walker (PTW), such as designating contiguous memory regions [1], [54] or utilizing address range checks [55]–[57]. However, an optimal solution would offer scalability and efficiency in allocating non-contiguous regions. Therefore, HyperTEE adopts a bitmap to record the state of every memory page, with each bit indicating whether a page belongs to enclave memory. The memory region of bitmap itself is marked as enclave memory for security.

HyperTEE integrates a bitmap checking logic into CS PTW. As shown in Figure 5, when a non-enclave memory access misses in TLB, PTW loads its PTE. Then, the translated physical page number is used to retrieve the bitmap. If the bitmap indicates it is not an enclave page, this access can be performed correctly. Otherwise, an access exception is thrown.

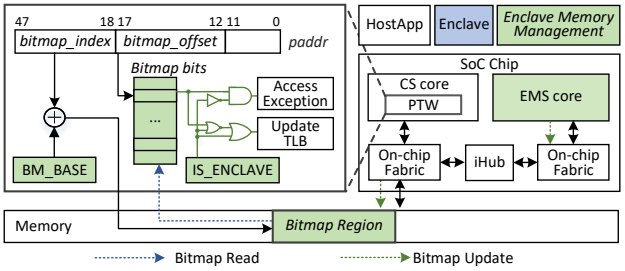


Fig. 5. Isolate enclave memory from host through bitmap checking after page table walking. BM_BASE register saves base address of Bitmap Region. IS_ENCLAVE register indicates whether the core is in enclave mode. Both of them can only be updated at the highest privilege mode.

Once verified, the TLB is updated to indicate that this page has been checked. Subsequent memory accesses hit in the TLB can thus proceed. To prevent circumvention of bitmap checking via stale TLB entries, EMCall flushes related TLB entries while encountering enclave context switches and bitmap changes. Notably, to prevent potential attacks, EMS protects enclave page table entries by disabling TLB sharing, a strategy aligned with existing literature [58].

Isolate enclave memory from other enclaves through page ownership checking. EMS maintains a page ownership table in private memory. Each entry records the unique enclaveID that owns a specific physical page. Before mapping a physical page to an enclave, EMS looks up and verifies the page ownership with the physical page number to ensure that the page has not been mapped to another enclave.

C. Enclave Memory Encryption and Integrity

Memory encryption. HyperTEE leverages a commercial multi-key memory encryption engine, similar to Intel MK-TME [59] and AMD SME [60]. Each enclave is assigned a unique encryption key and identification (KeyID), configured only by EMS via iHub and stored in the engine. The KeyID is stored to the high bits of PTE by EMS and is obtained by any memory access after address translation. In our implementation, the width of CS core front-side memory bus is 56 bits, among which the lowest 40 bits are used for the physical address, and the highest 16 bits are used for the KeyID. In case of KeyID exhaustion, EMS can suspend an enclave to release a KeyID. During this procedure, EMCall performs TLB flush and cache flush to avoid incorrect reuse.

Memory integrity. HyperTEE employs SHA-3 based MAC (28-bit) [61] employed by commercial TEEs [2]–[4], which is more suitable for large-size enclave memory than Merkle Trees [62]. In case of an integrity violation, an exception is triggered to prevent physical tampering attacks.

V. Enclave Communication Management

Enclave communication is critical and common in many real-world scenarios. In a typical scenario where an enclave interacts with I/O devices [63]–[66], the user enclave must send data and commands to the driver enclave which then forwards them to I/O devices. In existing TEEs, the communication

is performed on non-enclave memory, which requires time-consuming software encryption and decryption for security. Recently, communication approaches based on shared enclave memory have been proposed [43], [67], [68] and become prevalent due to high efficiency as plaintext communication. However, this method faces three security challenges: ① Key assignment: How to assign a dedicated shared encryption key and avoid disclosure or brute-force cracking. ② Page sharing: How to share enclave pages without compromising existing isolation among different enclaves. ③ Access control: How to protect the shared memory against a range of security threats, including unauthorized tampering [43], [69], releases, and I/O accesses. In HyperTEE, EMS can manage the shared memory and guarantee security during communication.

A. Shared Encryption Key Assignment

Benefiting from the physical isolation between CS and EMS, EMS can assign the keys securely for sharing enclave pages. To avoid interference between shared and private pages, EMS assigns unique keys for enclave shared memory, separate from private memory keys. EMS allows the sharing among multiple enclaves to support complex communication scenarios such as broadcasting. Given the unpredictable feature of communication participants and the possibility of new enclaves joining, it is unfeasible to negotiate keys according to the collective information of all participants before communication. Thus, we propose to derive keys using the initial sender EnclaveID and the shared memory identification (ShmID) assigned by EMS. Similar to enclave private memory, EMS writes the KeyID and corresponding key to the memory encryption engine directly. To ensure the security of keys for shared memory, two techniques are employed.

Guarantee confidentiality of keys. When a receiver enclave applies to share pages with sender enclave, it authenticates with the sender through local attestation, during which the EnclaveID and ShmID can be transmitted. Subsequently, the receiver enclave employs these two IDs as arguments of ESHMAT primitive, which invokes EMS to assign KeyID and store it in the relevant PTE. During this process, the software on CS side can only access the enclaveID and ShmID. Confidentiality of KeyID and Key is guaranteed.

Prevent brute-force key cracking. As for a shared memory region, an attacker may intentionally guess the ShmID in a brute-force method to gain access rights and break its confidentiality. To mitigate this threat, when a receiver enclave wants to share the region, it must first register the application with the sender enclave. If approved, the sender enclave requests EMS to append the receiver enclaveID to the *legal connection list* through ESHMSHR primitive. Thus, any illegal attempt to map this shared memory will be forbidden.

B. Ownership Management for Shared Enclave Page

EMS extends page ownership to allow pages to be shared between enclaves or between an enclave and a peripheral.

Enclave-Enclave shared memory. When an enclave requests EMS to create shared memory via ESHMGET primitive, EMS allocates a physical memory region and marks these pages as shared memory in page ownership table. Subsequently, these pages will not be mapped as private enclave memory, and a unique ShmID is assigned. Corresponding bitmap bits are set as well to prevent unauthorized non-enclave accesses. If a receiver enclave wishes to share the memory, it applies the ShmID from the sender enclave after local attestation. Then, using the ESHMAT primitive, the sender or receiver enclave can invoke EMS to map the shared memory.

Enclave-Peripheral shared memory. Similar to enclave-enclave shared memory, EMS also supports allocating enclave-peripheral shared memory. Differently, a peripheral cannot actively request EMS to map shared memory to its address space, instead, it relies on configuration from the driver enclave. Therefore, in our design, it is the driver enclave that requests EMS to grant the enclave memory access rights to I/O devices. Take typical devices with DMA engines as an example, EMS configures the physical address range, which determines the accessible memory region during data moving. In our current implementation platform, IOMMU is not supported, so we primarily focus on the peripherals with DMA engines. For peripherals relying on IOMMU, it is EMS to manage the IOMMU page tables to enhance security.

C. Access Control to Thwart Unauthorized Tampering

Protecting the communication process against potential threats from malicious participants is imperative. First, a receiver enclave may illegally modify shared memory designated as read-only, leading to unexpected results. Second, a receiver enclave may intentionally send requests to release and reclaim the shared memory, disrupting ongoing communications. Third, attackers may exploit an I/O DMA to circumvent the CS memory isolation mechanism and gain access to enclave memory. Targeting these threats, we propose countermeasures.

Permission check to prevent unprivileged tampering. When an initial sender enclave requests EMS to allocate shared memory, it specifies the maximum access permission for the receiver enclaves. When a receiver enclave wants to share the memory, it negotiates with the sender enclave to authorize its access permission. Meanwhile, the permissions are recorded in *legal connection list* for the shared memory. When the receiver enclave requests the EMS to map the shared memory through ESHMAT primitive, the permissions are set to its page tables. During communication, any attempt to modify permissions is handled by the sender enclave, which invokes EMS to update the permission in both *legal connection list* and page tables.

Identity and active connection check to prevent malicious release. To avoid malicious enclaves from releasing or reclaiming shared memory, EMS records the initial sender enclaveID in *shm control structure* during memory creation. The number of active connections is updated while executing ESHMAT/ESHMDT primitives. When there is no active connection, shared memory can be released and reclaimed only by the initial sender enclave through ESHMDES primitive.

Address legality check to prevent I/O compromises. The physical addresses of DMA accesses are continuous generally.

Therefore, HyperTEE employs the DMA whitelist in CS hardware. This whitelist consists of a set of register pairs and each register pair concludes the address, size, and permission to restrict the legal region for each DMA. Any DMA access beyond the legal region will be discarded. The whitelist is implemented as control registers within the on-chip fabric and is exclusively configurable by EMS.

VI. Other Management Tasks

Secure boot. Upon power on, EMS is booted up after the chip original initialization logic, and then followed by CS. Specifically, EMS BootROM is first executed to verify the EMS Runtime, which is encrypted and stored in EMS private flash. The hash value of Runtime is verified against pre-calculated hash value stored in an on-chip EEPROM to avoid physical tampering. Then, the hash of CS firmware and EMCall are verified similarly to prevent tampering. Finally, the CS OS starts its booting process.

Key management. HyperTEE derives all keys from the root keys, including Endorsement Key (EK) issued by certificate authority and Sealed Key (SK) randomly generated. Both EK and SK are burnt into the eFuse of EMS during manufacturing. Enclaves can invoke primitives on EMS to obtain keys for different purposes. For instance, the enclave memory encryption keys can be derived according to SK, enclave measurement. Attestation key (AK) can be derived from SK and a random salt. All key operations are carried out on EMS and are invisible to CS. When keys are no longer useful, EMS erases them with random values.

Remote attestation. HyperTEE implements remote attestation based on typical SIGMA protocol [70], which is widely employed in mainstream TEEs [1], [54], [56], [71], to prove the integrity of both hardware platform and enclaves. During secure boot and enclave creation, EMS measures the software TCB of platform and enclave to generate the measurement of them respectively. The attestation flow mainly contains: ① Remote user negotiates a symmetric key with enclave using Diffie-Hellman protocol. ② Enclave sends the certificates of platform and enclave to remote user for verification. The certificates are generated by EMS through signing the measurements of platform and enclave via EK and AK respectively. ③ Remote user verifies the integrity of platform and enclave, then sends its certificates to the enclave for verification.

Local attestation. Local attestation is enforced to prove the identities of enclaves and whether they are on the same platform. Same as existing TEEs, HyperTEE leverages the Elliptic-Curve Diffie–Hellman (ECDH) [72] key exchange protocol, including the major steps: ① When an enclave (*challenger*) attempts to attest with another enclave (*verifier*), they negotiate a symmetric key. *Challenger* sends its measurement to *verifier*. ② *Verifier* requests EMS to sign its measurement with a report key (derived from *challenger* measurement and SK) as a certificate and returns to *challenger*. ③ *Challenger* requests EMS to verify the certificate with the report key and sends its own certificate to *verifier* to prove its identity.

Data sealing. To protect enclave data in persistent storage, EMS derives a sealing key based on the enclave measurement and the device-unique SK. The relevant data is then encrypted with the sealing key to HostApp memory. Finally, HostApp can transfer the encrypted data to the disk.

TABLE III
Key parameters of HyperTEE FPGA prototype.

Parameter	CS core	EMS core in HyperTEE IP		
		Weak	Medium	Strong
Pipeline Type	OoO	In-order	OoO	OoO
Fetch / Decode	8/4	1/1	4/2	8/4
Mem/ Int/ Fp	2/3/1	1/1/1	1/2/1	2/3/1
BTB	256 × 4w	128 entries	128 × 2w	256 × 4w
PHT	TAGE 2048 entries	GShare 512 entries	TAGE 1024 entries	TAGE 2048 entries
Int/ Fp PhyRegs	128/128	None	96/96	128/128
ROB/ STQ/ LDQ	128/32/32	None	96/16/16	128/32/32
I/D/L2 TLB	32/32/1024	8/8/0	16/16/0	32/32/0
L1 I/D Cache	64/64 KB	16/16 KB	32/32 KB	64/64 KB
L2 Cache	1MB	256KB	512KB	512KB
Crypto engine	–	AES: 1.24Gbps, SHA-256: 16.1Gbps RSA Sign: 123ops/s Verify: 10Kops/s		
DNN Accelerator (Gemmini [73])				
Parameter	**PE**	**Global Buf**	**Accumulator**	**Dataflow**
Value	16×16	256KB	64KB	OS/WS

VII. Evaluation

A. Methodology

We implement a prototype of HyperTEE on Synopsys HAPS-80 S104 FPGA platform. Table III lists the major parameters. The cores are designed based on open-source RISC-V out-of-order processor BOOM [74] and in-order processor Rocket [75]. To investigate the complexity of EMS core in HyperTEE IP, we explore different combinations of EMS core and CS core. Moreover, we analyze the performance of enclave primitives and memory management. We evaluate the performance of enclave communication under two typical I/O device usage scenarios: one using an open-source DNN accelerator called Gemmini [73], and the other using a NIC controller. Additionally, we utilize typical Synopsys ASIC design flow and tools to investigate the hardware overhead.

Several benchmarks are selected for evaluation: ① To analyze the configuration of EMS, we port the wolfSSL and RV8 to enclaves. wolfSSL is an open-source SSL/TLS library that supports encryption, digests, and signature verification. RV8 is another typical benchmark that is widely used in many TEE studies [55]–[57]. ② To evaluate the impact on memory access latency, we utilize the MemStream. Additionally, we evaluate the impact of bitmap checking on non-enclave applications through SPEC CPU2017 Integer Rate and Speed [76] with *reference* input size. ③ In case of enclave communication, we employ typical DNN inference workloads including ResNet50 [77], MobileNet [78] and four multi-layer perceptions (MLP) [79]–[82]. The model codes and weights are considered confidential and protected in enclaves, while dataset input operations remain in HostApp.

In our experiments, we name different scenarios in the format "running environment-security mechanism". Specifically, "running environment" includes non-enclave *Host* and *Enclave* environment. The major security mechanisms include:

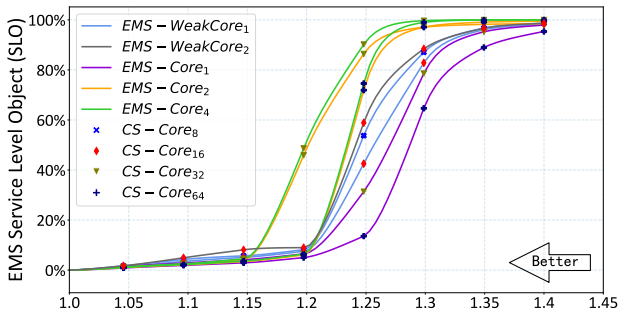


Fig. 6. Efficiency evaluation of resolving concurrent primitive requests from CS cores to EMS cores.

- ***Native***: None of security mechanisms are enabled.
- ***M_encrypt***: Memory encryption and integrity are enabled.
- ***Bitmap***: Bitmap checking is enabled. Notably, there is no bitmap checking for enclave applications, only non-enclave applications are affected by this mechanism.

For instance, *Host-Native* represents that the applications execute in *Host* without any security mechanisms enabled. This is considered as the **baseline** in our evaluation. The *Enclave-M_encrypt* represents that memory encryption and integrity mechanisms are enabled on enclave execution.

B. Evaluation of HyperTEE IP

Exploring EMS core configuration for different CS setups. While offloading enclave management tasks to a dedicated core, selecting an appropriate EMS core configuration within a specific CS setup is important. We conduct experiments to evaluate Service Level Objectives (SLO) for resolving concurrent primitive requests across various EMS core configurations within a given CS configuration. Due to FPGA hardware limitations, direct emulation of complex CS scenarios is infeasible, and we adopt software simulation. Specifically, multiple processes are employed to simulate CS and EMS cores, with each process representing a single core. CS cores concurrently initiate primitive requests to EMS cores using Inter-Process Communication (IPC). The primitives involved in evaluation include necessary enclave creation primitives and 16384 dynamic memory allocation (2MB) primitives. Utilizing latency data sampled from FPGA prototype, we adjust the EMS configuration with different computation capabilities and measured response latency. While improving the computation capability yielded minimal performance improvement, we consider that the EMS core configuration effectively meets performance requirements.

The experiment results are shown in Figure 6. For each curve, the *baseline* latency is defined as the maximum latency required to resolve 99% of requests (SLO) in non-enclave mode. Each point on a curve represents the percentage of primitives that can be resolved in enclave mode within x times the *baseline* latency. Under the same CS core setup, increasing the number of EMS cores improves efficiency in resolving concurrent primitive requests, and corresponding curve is closer to y-axis. It can be observed that a minimal EMS configuration can handle enclave management for CS cores. In particular, for a high-performance processor (32 or 64 cores), a dual-core out-of-order design can achieve similar SLO as the quad-core design, making the dual-core configuration adequate. Similarly, for a desktop processor (16 cores), a dual-core in-order design proves sufficient. In terms of a high-end embedded processor (no more than 4 cores), a single in-order core meets the requirement.

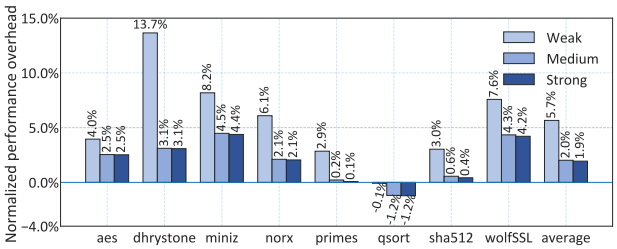


Fig. 7. Performance overhead of different EMS core configurations.

Performance of different EMS core configurations. To evaluate the overall performance overhead on enclaves while introducing EMS cores, we select the configuration of CS core and three configurations of EMS cores listed in Table III. Experiment results in Figure 7 show that the average overheads for three representative configurations are 5.7%, 2.0%, and 1.9% respectively. It can be observed that *medium* configuration outperforms *weak* configuration by 3.7%, but there is only a 0.1% difference between *medium* and *strong* configurations. This is because management tasks are generally simple and do not require advanced microarchitecture. In subsequent experiments, we select *medium* configuration to analyze the performance of different mechanisms in HyperTEE.

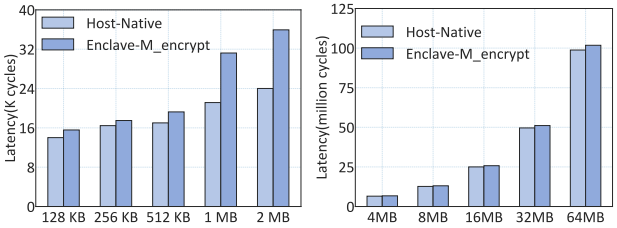
Performance of enclave primitives execution. We investigated the performance overhead deriving from the primitive execution and conducted a comparative analysis with and without the deployment of a crypto engine, which is integrated in existing dedicated cores [37], [38]. As depicted in Table IV, without the crypto engine, enclave primitives account for almost 10.4% of the total execution time in non-enclave mode. About three quarters of the 10.4% (7.8% to be exact) is attributable to the EMEAS primitive. While deploying a crypto engine, the execution time of enclave primitives is decreased from 10.4% to 2.5% of non-enclave mode. Time consumed by EMEAS primitive is also decreased from 7.8% to 0.1%. Note that the average performance overhead of *All Primitives* in *Enclave-Crypto* is slightly higher than the overall overhead (2.5%) in Figure 7. The reason is that static memory allocation during enclave creation shortens the execution time of enclaves in addition to primitive acceleration.

C. Performance Evaluation of Enclave Memory Management

Notably, enclave memory management tasks such as memory allocation, encryption, and integrity protection only impact the performance of enclave applications. Memory isolation with hardware bitmap checking only affects non-enclave applications. So we evaluate their performance impacts respectively.

TABLE IV
EXECUTION TIME OF ENCLAVE PRIMITIVES COMPARED TO EXECUTION TIME IN HOST-NATIVE.

	Enclave-Noncrypto		Enclave-Crypto	
	All Primitives	EMEAS	All Primitives	EMEAS
aes	6.8%	5.1%	1.6%	0.06%
dhrystone	19.0%	14.3%	4.5%	0.18%
miniz	8.1%	6.1%	1.9%	0.08%
norx	10.4%	7.8%	2.5%	0.10%
primes	5.1%	3.9%	1.2%	0.05%
qsort	2.8%	2.1%	0.7%	0.03%
sha512	10.8%	8.1%	2.6%	0.10%
wolfSSL	19.9%	15.0%	4.7%	0.19%
Average	10.4%	7.8%	2.5%	0.10%



(a) Memory allocation. (b) Memory encryption and integrity.

Fig. 8. Performance of enclave memory management.

Performance overhead of enclave memory allocation. In HyperTEE, the allocation of enclave memory pages requires specific requests to EMS, which introduces execution latency. To evaluate this overhead, we measure the latency to execute the `malloc` and `EALLOC` in non-enclave applications and enclaves respectively, with different memory sizes ranging from 128KB to 2MB, repeating each size 1000 times. As shown in Figure 8(a), enclave applications exhibit a performance overhead of 6.3% to 49.7%, compared to non-enclave applications. This overhead is primarily attributed to the transmission of primitives between the CS and EMS, and the execution within the EMS core, which is relatively weaker than the CS core. However, in most real-world programs, memory allocation requests are not frequent, resulting in minimal overall performance overhead.

Performance overhead of memory encryption and integrity. The latency caused by memory encryption and integrity protection only occurs when accessing off-chip memory. MemStream benchmark has a high cache miss rate and can reflect the worst performance overhead of encryption and integrity. As recommended by the MemStream, the memory sizes are at least four times greater than the size of last-level cache. Therefore, we evaluate the memory size ranging from 4MB to 64MB. Figure 8(b) shows that the average latency overhead is 3.1%. Notably, in most real-world applications, the performance overhead is further reduced benefiting from higher cache hit rates than MemStream.

While taking into account all memory management, the average performance overhead for wolfSSL in enclave mode is 0.9%, as demonstrated in Figure 9.

Performance overhead of memory isolation. Considering that non-enclave memory accesses are checked at page table walker after TLB miss, we evaluate the performance overhead

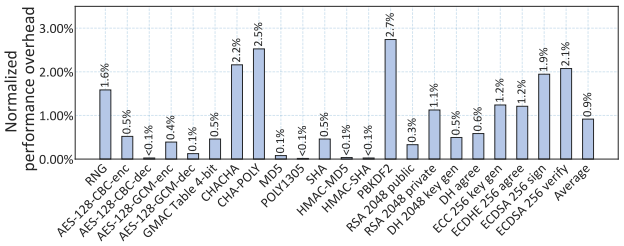


Fig. 9. Performance impact of enclave memory management on enclave applications (wolfSSL).

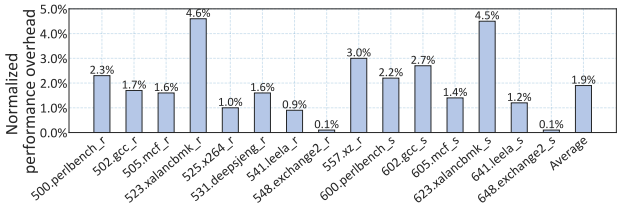


Fig. 10. Performance overhead of enclave memory isolation on non-enclave applications in Host-Bitmap (SPEC CPU 2017).

on non-enclave applications. As shown in Figure 10, the average overhead is 1.9%, which is minimal because only one additional bitmap retrieve operation is introduced and the bitmap checking is performed in parallel with the original permission check. For most cases, the overhead is marginal due to high spatial locality and TLB hit rates. While in the case of memory-intensive `xalancbmk_r`, the TLB miss rate (0.8%) is much higher than other cases (<0.2%), and thus more bitmap checking operations are introduced, leading to relatively high overhead (4.6%).

Performance overhead of TLB flush triggered by bitmap updates. During enclave execution, bitmap updates require TLB flush to guarantee security, which may incur performance overhead for non-enclave applications. Notably, most bitmap updates centralize in static allocation during enclave creation and dynamic enclave memory augment via `EALLOC`. This means that such updates are relatively infrequent. Evaluation of enclave workloads demonstrates that there are only 16.72 flushes per billion instructions on average. With this flush frequency, the average performance overhead on non-enclave SPEC CPU2017 benchmark is less than 0.7%. To evaluate the performance overhead of TLB flushes on enclaves with more context switches, we select *miniz* in rv8, a typical data compression program, with memory size ranging from 2MB to 32MB. As for enclave context switch frequency, we select standard frequency (100Hz) as baseline, and improve the frequency to 1.5×, 2×, and 4×. As shown in Figure 11, while increasing the switch frequency, a minimal overhead is introduced. When the memory size is 32MB at the frequency in 400Hz, the performance overhead is no more than 1.81%.

D. Performance Evaluation of Enclave Communication

To evaluate the performance of enclave communication, we use two typical I/O usage scenarios that involve *user enclave*

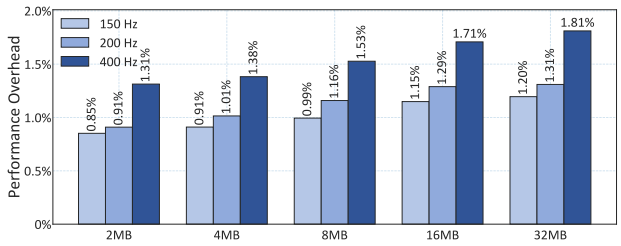


Fig. 11. Performance overhead of TLB flush on enclaves.

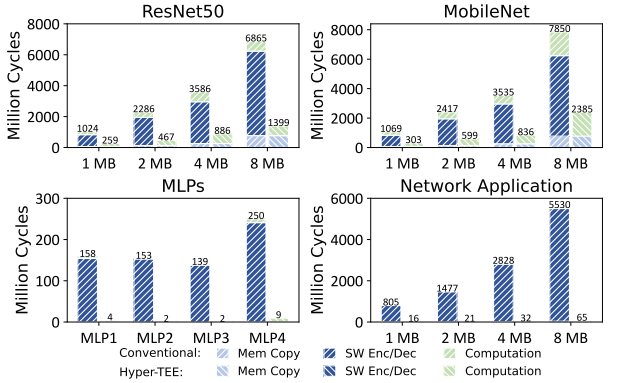


Fig. 12. Performance evaluation of enclave communication.

to driver enclave and *driver enclave to peripheral* communication. ① AI accelerator (Gemmini [73]) usage in enclaves: Figure 12 shows the experiment results. In ResNet50, software encryption and decryption account for the majority of overall execution time (more than 74.7%) in conventional designs, and this proportion increases as the size of input data increases. By eliminating the bottleneck, HyperTEE achieves a performance speedup of more than 4.0×. Similarly, MobileNet shows a boost of more than 3.3×. For MLPs, with fewer network layers than ResNet50 and MobileNet, the proportion of time for encryption and decryption is higher, leading to performance improvements of more than 27.7×. ② NIC controller usage in enclaves: Our experiments show that network applications have less computation, and the encryption and decryption operations occupy more than 98.0% of the total transmission time. HyperTEE achieves 50× performance improvement.

E. Hardware Overhead

Based on TSMC 7nm technology, we utilize Synopsys ASIC design flow and tools to assess the implementation cost of HyperTEE. Table V shows the synthesis results with physical place and route information for different CS configurations. Notably, the EMS area consists of the core and other hardware modules, including the crypto engine that occupies $0.20mm^2$. It can be observed that in all cases, EMS cores occupy less than 1% of SoC chip. For the high-performance 64-core case, EMS cores occupy only 0.25%. Timing analysis shows that adopting HyperTEE architecture has a negligible timing impact on CS cores. The maximum frequency of CS core and EMS core are 2.5GHz and 750MHz respectively.

TABLE V
AREA OVERHEAD OF EMS CORES FOR DIFFERENT CONFIGURATIONS.

CS Core	4	8	16	32	64
CS Area	35mm^2	74mm^2	151mm^2	304mm^2	612mm^2
EMS Core	1 Weak Core		2 Weak Cores	2 Medium Cores	
EMS Area	0.34mm^2		0.51mm^2	1.5mm^2	
Overhead	0.97%	0.46%	0.34%	0.49%	0.25%

VIII. SECURITY ANALYSIS

A. Trusted Computing Base

The TCB of HyperTEE includes the following components. ① Hardware: the whole SoC package, which contains CS hardware, EMS hardware, as well as peripheral devices used by the enclaves. ② Firmware: the SoC initialization codes and secure booting codes of the EMS, and the firmware of the CS (e.g., codes for handling the EMCall). The SoC initialization codes and secure booting codes are generated in the manufacturing phase and are stored securely within on-chip storage, which cannot be altered after being shipped. ③ EMS Runtime: It comprises 3843 lines of code written in memory-safe Rust programming language. The memory safety problems in the interface with EMCall are eliminated, such as buffer overflows. Its codebase is small enough to be formally verified by state-of-the-art frameworks [41], [83], which can verify codebases comprising tens of thousands of lines.

B. Security Comparison with Existing TEEs

In terms of the execution of enclaves themselves, HyperTEE maintains the same security level as other TEEs, all susceptible to microarchitectural side-channel attacks. However, HyperTEE significantly outperforms these architectures in the security of enclave management tasks, as demonstrated in Table VI. For instance, enclave management tasks in SGX are untrusted and are exposed to various controlled-channel and microarchitectural attacks. TDX, the new generation of SGX, employs a dedicated TDX module (logically isolated and physically shared) to perform enclave management tasks. It can defend against page table management-based controlled-channel attacks but is still vulnerable to other three types of controlled-channel attacks. In contrast, HyperTEE can defend against all these controlled-channel attacks. Moreover, HyperTEE effectively mitigates microarchitectural side-channel attacks on enclave management tasks via physical isolation.

C. Attack Surface Analysis

CS cache hierarchy. The data managed by EMS that can be loaded into CS cache hierarchy only includes the bitmap, enclave control structure, and PTEs. When attackers try to create cache side channels targeting EMS Runtime, their observation is limited to updates to these data. Actually, updates to these data occur only when CS applications proactively invoke primitive requests to EMS. This implies that any cache state changes resulting from updates to these data are a consequence of CS-initiated actions and do not reveal sensitive information about EMS tasks. Therefore, constructing side channels in CS cache hierarchy is impractical.

TABLE VI
DEFENSE CAPABILITY AGAINST MANAGEMENT TASK ATTACKS.

TEE	Controlled channel Attacks				μArch Attacks
	Memory Management			Communication Management	
	Allocation	Page table	Swapping		
SGX	○	○	○	○	○
SEV	○	○	○	○	◐
TDX	○	●	○	○	○
CCA	○	●	○	○	○
TrustZone	●	●	●	○	○
KeyStone	●	●	●	○	◐
Penglai	○	●	○	○	◐
CURE	○	●	○	○	◐
HyperTEE	●	●	●	●	●

○represents that the attacks cannot be defended. ●represents the attacks can be defended. ◐represents that some attacks can be defended while others cannot.

CS PTW. An attacker may exploit PTW to load enclave data and construct side channels by manipulating page tables. This can be prevented in HyperTEE. First, this channel cannot be constructed from an enclave since attackers cannot modify enclave page tables. Second, in non-enclave applications, even though attackers can manipulate PTEs to map to enclave memory, PTW cannot decrypt enclave data correctly. This is because different KeyIDs are used while loading non-enclave page tables and enclave data. Moreover, the KeyIDs cannot be modified by attackers.

Mailbox. To prevent forging fake requests, HyperTEE only allow EMCall to send primitive requests through mailbox. When EMS receives primitive requests, a sanity check is performed on parameters to filter out susceptible requests. Moreover, to read primitive responses from the mailbox, EMCall employs a polling method, avoiding the untrusted interrupt handling codes in CS OS. Besides, a primitive request and its response are exclusively bound by EMCall, preventing malicious enclaves from reading other primitive responses.

On-chip Fabric. Attackers can exploit the traffics on the on-chip fabric [84], [85]. However, such attacks are impractical in our design. First, attackers are constrained to invoke management tasks on EMS at the granularity of primitives, significantly complicating their ability to precisely observe the specific victim instruction sequences. Additionally, during the execution of a targeted victim primitive, attackers are unable to prevent EMS from initiating other memory or I/O accesses. Then, it is difficult for attackers to distinguish whether an operation observed is from the victim instruction sequence or other concurrent tasks accurately.

IX. DISCUSSION

Support for VM-level TEEs. From the design perspective, HyperTEE can naturally support the lifecycle management of CVMs and the deployment of encrypted VM images by adding dedicated primitives in EMS. EMS can perform CVM memory management and provide memory isolation and encryption. In scenarios where CVMs communicate with each other, EMS can also allocate protected shared memory between CVMs. To support CVM snapshot, save, and restore, EMS ensures the confidentiality and integrity of CVM memory by encrypting it using AES algorithm and creating a Merkle tree. The encryption key and the root hash value are stored in the private memory of EMS. To support CVM migration, EMS can perform remote attestation between the source and destination nodes to establish an encrypted channel for transmitting the CVM encryption key and root hash value, and then transfer the encrypted CVM. We leave this in the future work.

TEE for GPU. HyperTEE adopts the same solution as existing designs to support TEE for GPU [63], [64], [66]. ① Dedicated driver enclave for the GPU driver: user enclaves can send data and commands to the driver enclave, which forwards them to the GPU. ② Control path isolation: HyperTEE binds the GPU control path exclusively to an enclave (Section IV-B) and uses bitmap checking to prevent access from non-secure world software. ③ Data path protection: HyperTEE utilizes the EMS communication management task (Section V-B) to create bitmap-protected shared memory between the GPU and the driver enclave for data and command transmission. Additionally, IOMMU-enabled GPUs can be supported as well, with IOMMU being managed by EMS for security, including register configuration, IOTLB cache invalidation, and address translation table maintenance. The address translation table records memory regions accessible to GPU DMA and protects enclave memory from unauthorized DMA accesses.

Control Flow Integrity (CFI) and memory safety of enclave codes. It is promising to integrate conventional CFI protection and memory safety into HyperTEE to avoid vulnerabilities in enclave codes. There are three typical approaches: software instrumentation [86], [87], hardware monitoring [88], [89], and software monitoring [90]–[92]. The first two approaches only need the support of compilers or hardware on CS, enabling seamless integration in HyperTEE without modification. The third approach uses hardware to record enclave control flow transfers in a buffer, which are analyzed by a monitoring task. Once detecting malicious behaviors, the enclave is terminated. This monitoring task can be deployed in the EMS, with access to the buffer in the enclave's private memory. While the task may change CS cache states, these changes are only associated with the monitoring tasks and are irrelevant to the enclave or sensitive management tasks, thus posing no risks to them.

Orthogonal with countermeasures against microarchitectural side-channel attacks on enclave execution. Regarding microarchitectural side channels on the execution of enclaves themselves, HyperTEE has the same security level as existing TEEs. There are many countermeasures and some of them have been deployed in commercial processors. For instance, CAT [93], CEASER [94], [95], and other studies [54]–[57], [96], [97] propose to perform cache partition or randomization. Additionally, invalidating branch prediction tables [98]–[101] upon context switches or privilege changes effectively prevents attackers from deducing secrets from branch history. As for many attacks that exploit frequent interrupts [51], [52], Varys [102] proposes to terminate enclave execution upon detecting abnormal interrupt frequency. Importantly, these studies are orthogonal to HyperTEE and

can be incorporated to enhance security.

X. Related Work

Multi-core processor architecture with dedicated security cores. Several commercial processors have integrated dedicated cores [35]–[38] to provide security services. SEP [35], Titan [36], and SPU [37] aim to protect vendor-provided sensitive functions, but they do not protect user-programmed applications. PSP [38] provides encryption and attestation services for CVMs in SEV. Some academic studies [103], [104] propose to provide secure boot or other services. However, most critical management tasks including page table management and memory allocation remain in the original computing cores, exposing enclaves to controlled-channel and microarchitectural side-channel attacks. Differently, HyperTEE is the first decoupled architecture, within which enclave management tasks are offloaded to dedicated cores for security, while the enclaves are executed in original cores to maintain high performance and workload diversity.

Enclave memory management. Memory management in conventional TEEs [1], [4], [105] relies on untrusted OS or hypervisor, leading to various controlled-channel attacks. One solution is to deploy the management within enclaves [6], [7], [55], [56]. However, malicious enclaves may exploit management tasks within enclaves to attack other non-enclave software [32], [106], [107]. Another approach is to utilize trusted hypervisors or software modules at a dedicated privilege mode [3], [108], [109], but they are vulnerable to microarchitectural side-channel attacks. In contrast, HyperTEE decouples enclave memory management to the physically isolated EMS cores, effectively eliminating all these threats.

Enclave communication management. Some studies [54], [57] support enclave communication through data copy or page remap via the highest privilege level, but they require time-consuming privilege switches. Timber-V [110] proposes to employ shared enclave memory but it lacks permission management, which is susceptible to compromises by malicious enclaves. PIE [67] proposes to restrict the access permission of shared memory to read-only. Elasticlave [43] introduces a management mechanism to allow more access permission. However, it does not address three problems, including how to assign and protect shared encryption keys; how to allow memory sharing in both enclave-to-enclave and enclave-to-peripheral communication; how to prevent compromises from I/O devices. In HyperTEE, all these problems are tackled.

XI. Conclusion

This paper proposes HyperTEE, a novel TEE architecture that divides the entire system into two subsystems: the original CS for enclave and the EMS for enclave management tasks. Enclave management tasks are deployed on EMS to eliminate controlled-channel attacks and protect them against microarchitectural side-channel attacks. Experiment results on FPGA prototype demonstrate that HyperTEE introduces 2.0% and 1.9% performance overhead on average for enclaves and non-enclave processes respectively.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. We thank Xin Tian for prototype implementation and technical support. We thank Prof. Yungang Bao, Prof. Dan Tang and Biwei Xie, Jian Chen, Shan Liu, Jian Zhang, Yinan Xu and Prof. Ninghui Sun from the XiangShan team at Beijing Institute of Open Source Chip (BOSC) and Institute of Computing Technology, Chinese Academic of Sciences (ICT, CAS) for their invaluable support in subsequent work on the integration of HyperTEE to the open-source high-performance RISC-V XiangShan processor. This work is supported by the National Science Fund for Distinguished Young Scholars under Grant No. 62125208.

References

[1] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *Hasp@ isca*, vol. 10, no. 1, 2013.

[2] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, 2016, pp. 1–9.

[3] Intel, "Intel® trust domain extensions (intel® tdx)," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2022.

[4] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper*, 2016.

[5] A. SEV-SNP, "Strengthening vm isolation with integrity protection and more," *White Paper, January*, 2020.

[6] A. ARM, "Security technology building a secure system using trustzone technology (white paper)," *ARM Limited*, 2009.

[7] ARM, "Arm confidential compute architecture," https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2022.

[8] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing," in *26th {USENIX} security symposium ({USENIX} security 17)*, 2017, pp. 557–574.

[9] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 693–707, 2018.

[10] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level directional predictor based side-channel attack against sgx," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 321–347, 2020.

[11] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," *CoRR*, vol. abs/1807.07940, 2018. [Online]. Available: http://arxiv.org/abs/1807.07940

[12] F. Dall, G. De Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks," 2018.

[13] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 69–90.

[14] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[15] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure:{SGX} cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[16] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[17] M. Hähnel, W. Cui, and M. Peinado, "{High-Resolution} side channels for untrusted operating systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 299–312.

[18] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 538–570, 2019.

[19] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[20] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 339–354.

[21] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice," https://sgaxeattack.com/, 2020.

[22] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1852–1867.

[23] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[24] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[25] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[26] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 317–328.

[27] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1041–1056.

[28] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2421–2434.

[29] D. Kim, D. Jang, M. Park, Y. Jeong, J. Kim, S. Choi, and B. B. Kang, "Sgx-lego: Fine-grained sgx controlled-channel attack and its countermeasure," *computers & security*, vol. 82, pp. 118–139, 2019.

[30] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "{CopyCat}: Controlled {Instruction-Level} attacks on enclaves," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 469–486.

[31] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting unprotected {I/O} operations in {AMD's} secure encrypted virtualization," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1257–1272.

[32] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: Closing controlled channels with self-paging enclaves," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[33] S. Aga and S. Narayanasamy, "Invisipage: oblivious demand paging for secure enclaves," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 372–384.

[34] Intel, "Architecture specification: Intel® trust domain extensions (intel® tdx) module," https://cdrdv2.intel.com/v1/dl/getContent/733568, 2024.

[35] Apple, "Secure enclave," https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web, 2022.

[36] Google, "Pixel 6: Setting a new standard for mobile security," https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html, 2022.

[37] Qualcomm, "Secure processing unit spu230 core security target lite," https://www.commoncriteriaportal.org/files/epfiles/1045b_pdf.pdf, 2024.

[38] R. Lai, "Amd security and server innovation," *UEFI PlugFest-March (2013)*, pp. 18–22, 2013.

[39] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "{CIPHERLEAKS}: Breaking constant-time cryptography on {AMD}{SEV} via the ciphertext side channel," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 717–732.

[40] ——, "Tlb poisoning attacks on amd secure encrypted virtualization," in *Annual Computer Security Applications Conference*, 2021, pp. 609–619.

[41] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A secure and formally verified linux kvm hypervisor," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1782–1799.

[42] M. Li, Y. Zhang, and Z. Lin, "Crossline: Breaking" security-by-crash" based memory isolation in amd sev," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2937–2950.

[43] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An efficient memory model for enclaves," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4111–4128.

[44] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[45] M. Gruhn and T. Müller, "On the practicability of cold boot attacks," in *2013 International Conference on Availability, Reliability and Security*. IEEE, 2013, pp. 390–397.

[46] P. Stewin and I. Bystrov, "Understanding dma malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 21–41.

[47] C. Schwarz, V. Reusch, and M. Planeta, "Dma security in the presence of iommus," *Tagungsband des FG-BS Frühjahrstreffens 2022*, 2022.

[48] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual volume 2: Privileged architecture version 1.7," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[49] A. Holdings, "Arm architecture reference manual, armv8, for armv8-a architecture profile," *ARM, Cambridge, UK, White Paper*, 2019.

[50] Intel, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 2: Instruction Set Reference, Part*, 2022.

[51] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.

[52] ——, "Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 178–195.

[53] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal attack: Leaking {Control-Flow} in {SGX} via the {CPU} frontend," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 663–680.

[54] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 857–874.

[55] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "{CURE}: A security architecture with customizable and resilient enclaves," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[56] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[57] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the penglai enclave," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 275–294.

[58] Y. Zhang, Y. Hu, Z. Ning, F. Zhang, X. Luo, H. Huang, S. Yan, and Z. He, "{SHELTER}: Extending arm {CCA} with isolation in user space," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6257–6274.

[59] Intel, "Intel architecture memory encryption technologies specification." https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf, 2023.

[60] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper*, 2016.

[61] S. Gueron, "A memory encryption engine suitable for general purpose processors," *Cryptology ePrint Archive*, 2016.

[62] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[63] S. Weiser and M. Werner, "Sgxio: Generic trusted i/o path for intel sgx," in *Proceedings of the seventh ACM on conference on data and application security and privacy*, 2017, pp. 261–268.

[64] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 455–468.

[65] K. Xia, Y. Luo, X. Xu, and S. Wei, "Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 301–306.

[66] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on {GPUs}," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 681–696.

[67] M. Li, Y. Xia, and H. Chen, "Confidential serverless made efficient with plug-in enclaves," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 306–318.

[68] D. Lee, K. Cheang, A. Thomas, C. Lu, P. Gaddamadugu, A. Vahldiek-Oberwagner, M. Vij, D. Song, S. A. Seshia, and K. Asanovic, "Cerberus: A formal approach to secure and efficient enclave memory sharing," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1871–1885.

[69] J. Z. Yu, S. Shinde, T. E. Carlson, and P. Saxena, "Elasticlave: An efficient memory model for enclaves," https://www.usenix.org/system/files/sec22_slides-yu_jason.pdf, 2022.

[70] H. Krawczyk, "Sigma: The 'sign-and-mac'approach to authenticated diffie-hellman and its use in the ike protocols," in *Annual international cryptology conference*. Springer, 2003, pp. 400–425.

[71] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using tee," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1723–1740.

[72] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*. Springer, 2006, pp. 207–228.

[73] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 769–774.

[74] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5, 2020.

[75] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.

[76] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[77] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[78] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[79] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural computation*, vol. 22, no. 12, pp. 3207–3220, 2010.

[80] U. Meier, D. C. Ciresan, L. M. Gambardella, and J. Schmidhuber, "Better digit recognition with a committee of simple neural nets," in *2011 international conference on document analysis and recognition*. IEEE, 2011, pp. 1250–1254.

[81] X. Lu, Y. Tsao, S. Matsuda, and C. Hori, "Speech enhancement based on deep denoising autoencoder." in *Interspeech*, vol. 2013, 2013, pp. 436–440.

[82] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng, "Multimodal deep learning," in *ICML*, 2011.

[83] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "{CertiKOS}: An extensible architecture for building certified concurrent {OS} kernels," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 653–669.

[84] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU On-Chip ring interconnect are practical," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 645–662. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella

[85] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, "Don't mesh around: Side-Channel attacks and mitigations on mesh interconnects," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2857–2874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/dai

[86] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[87] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 941–951.

[88] L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele, "{PHMon}: A programmable hardware monitor and its security use cases," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 807–824.

[89] Intel, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: system programming guide, Part*, vol. 2, no. 11, pp. 1–64, 2011.

[90] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 173–184.

[91] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 927–940.

[92] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 195–211.

[93] I. Corp, "Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family," https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, 2022.

[94] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 775–787.

[95] ——, "New attacks and defense for encrypted-address cache," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 360–371.

[96] D. Townley, K. Arıkan, Y. D. Liu, D. Ponomarev, and O. Ergin, "Composable cachelets: Protecting enclaves from cache {Side-Channel} attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2839–2856.

[97] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "Dr. sgx: Automated and adjustable side-channel protection for sgx using data location randomization," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 788–800.

[98] L. Zhao, P. Li, R. Hou, M. C. Huang, J. Li, L. Zhang, X. Qian, and D. Meng, "A lightweight isolation mechanism for secure branch predictors," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1267–1272.

[99] T. Zhang, T. Lesch, K. Koltermann, and D. Evtyushkin, "Stbpu: A reasonably safe branch predictor unit," *arXiv preprint arXiv:2108.02156*, 2021.

[100] I. Corp, "speculative execution side channel mitigations," https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf, 2022.

[101] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 40–51.

[102] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting {SGX} enclaves from practical side-channel attacks," in *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 227–240.

[103] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "{ReZone}: Disarming {TrustZone} with {TEE} privilege reduction," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2261–2279.

[104] K. Suzaki, "Trusted rv: 64bit risc-v tee with secure coprocessor and software on them," https://riscv.or.jp/wp-content/uploads/TRASIO-RISC-V-Day-Tokyo-Spring-2021-a.pdf, 2024.

[105] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-x: A flexible architecture for hardware-managed isolated execution," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 190–202.

[106] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1741–1758.

[107] W. Liu, H. Chen, X. Wang, Z. Li, D. Zhang, W. Wang, and H. Tang, "Understanding tee containers, easy to use? hard to trust," *arXiv preprint arXiv:2109.01923*, 2021.

[108] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 2010, pp. 1–12.

[109] Y. Jia, S. Liu, W. Wang, Y. Chen, Z. Zhai, S. Yan, and Z. He, "Hyperenclave: An open and cross-platform trusted execution environment," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Jul. 2022.

[110] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v." in *NDSS*, 2019.