SPEAR-V: Secure and Practical Enclave Architecture for RISC-V

David Schrammel
david.schrammel@iaik.tugraz.at
Graz University of Technology
Graz, Austria

Moritz Waser*
moritz.waser@iaik.tugraz.at
Graz University of Technology
Graz, Austria

Lukas Lamster
lukas.lamster@iaik.tugraz.at
Graz University of Technology
Graz, Austria

Martin Unterguggenberger*
martin.unterguggenberger@iaik.tugraz.at
Graz University of Technology
Graz, Austria

Stefan Mangard
stefan.mangard@iaik.tugraz.at
Graz University of Technology
Graz, Austria

ABSTRACT

Trusted Execution Environments (TEEs) and enclaves have become increasingly popular and are used from embedded devices to cloud servers. Today, many enclave architectures exist for different ISAs. However, some suffer from performance issues and controlled-channel attacks, while others only support constrained use cases for embedded devices or impose unrealistic constraints on the software. Modern cloud applications require a more flexible architecture that is both secure against such attacks and not constrained by, e.g., a limited number of physical memory ranges.

In this paper, we present SPEAR-V, a RISC-V-based enclave that provides a fast and flexible architecture for trusted computing that is compatible with current and future use cases while also aiming at mitigating controlled-channel attacks. With a single hardware primitive, our novel architecture enables two-way sandboxing. Enclaves are protected from hosts and vice versa. Furthermore, we show how shared memory and arbitrary nesting can be achieved without additional performance overheads. Our evaluation shows that, with minimal hardware changes, a flexible, performant, and secure enclave architecture can be constructed, imposing zero overhead on unprotected applications and an average overhead of 1% for protected applications.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Security in hardware**.

KEYWORDS

RISC-V, enclave, isolation, memory protection, memory tagging

1 INTRODUCTION

In recent years, strong software isolation has become increasingly important. All major CPU vendors offer built-in technologies that allow software to run in shielded execution environments (e.g., TEEs, enclaves, secure virtual machines). Cloud computing platforms like Intel SGX [16] or AMD SEV-SNP [25, 46] allow tenant software to run in isolated computing environments. These technologies protect the encapsulated software from other, potentially malicious, tenants and even the hypervisor or operating system (OS). This allows removing the trust from the cloud provider by shifting it to the underlying hardware. Unfortunately, research shows that most of the established designs have substantial shortcomings. Intel SGX imposes significant performance overhead on code running in an enclave, and it does not allow for nesting enclaves or for protecting the host from a potentially malicious enclave. Furthermore, numerous successful attacks have been mounted on both SGX [18, 29, 29, 37, 55] and AMD's SEV [31, 32], proving that the isolation guarantees do not hold under certain circumstances. Recently, several enclave technologies have also arisen from the open RISC-V platform, such as Sanctum [15], Keystone [30], or Penglai [22]. However, these open-source architectures also have certain drawbacks, such as limiting memory to a few physically contiguous regions, that must be addressed in consideration of current and possible future use cases.

Existing work either uses insecure page tables managed by an untrusted OS or duplicates them to a separate area, which introduces large memory overheads and adds complexity to enclave software. Thus we identify the following research question:
Can enclaves be secure while still relying on OS-managed paging structures without the overhead of duplicating them?

In this work, we propose SPEAR-V, an enclave architecture based on a lightweight memory tagging hardware extension for RISC-V CPUs. SPEAR-V imposes a minimal performance overhead on isolated software and virtually no overhead on unprotected applications. Unlike other designs, SPEAR-V allows for dynamic enclave memory allocation, enclave nesting, and copy-free memory sharing. With our approach, we can efficiently share memory from the host to enclave and enclave to enclave, thus allowing for much greater flexibility than existing designs. Additionally, we do not limit sharing to strictly bijective mappings. Instead, all enclaves running in the context of a host application can share memory with an arbitrary number of other enclaves running in the same context. In addition to solving controlled-channel attacks and contrary to Intel's SGX, our hardware extension can also protect the host application from an enclave without relying on other hardware features (e.g., MPK). Thus, we offer protection both from malicious enclaves, as well as malicious hosts (OS or host application). Our design demonstrates that the necessary security can be guaranteed without moving or duplicating page table information to a separate protected region. Instead, we use OS-managed page tables without forfeiting any of the given security guarantees. This approach greatly reduces the complexity of enclaves, as they do not need to handle their own paging structures, thus avoiding the overheads associated with self-paging [40]. We show the feasibility of our design by implementing it on the CVA6 RISC-V CPU [65]. The imposed overhead on software running in enclaves is below

*The work was done while the authors were at Lamarr Security Research.

1% for representative benchmarks. We further demonstrate that SPEAR-V can be used without imposing any additional latencies on unprotected host applications.

Contributions. In short, our contributions are as follows:

- We present SPEAR-V, a lightweight enclave architecture for RISC-V CPUs that facilitates OS-managed, yet secure, and flexible memory management.
- We show how our architecture protects against accidental information leakage, controlled-channel attacks, malicious enclaves (sandboxing), and malicious hosts while also allowing for efficient, arbitrarily nested enclaves.
- We evaluate the performance of our architecture, showing minimal impact on the software running in an enclave and virtually no overhead for unprotected applications.
- We provide a flexible and extendable hardware implementation based on the CVA6 CPU, which allows for reproducible evaluation of our design and facilitates further research based on page-granular memory tagging.
- We open-source our implementation to facilitate future research in this area: https://github.com/IAIK/spearv

2 BACKGROUND

In this section, we briefly introduce the necessary background knowledge about enclaves, trusted execution environments, and controlled channel attacks.

Secure enclaves and trusted execution environments aim to provide a protected execution environment for applications and protect data from powerful adversaries. In Intel SGX, enclaves can run within a host application and have access to that address space, but neither the host application nor the OS has access to the code and data within an enclave. Enclaves ensure integrity and confidentiality of their contained data. Existing designs provide confidentiality by limiting access to memory regions through hardware features (e.g., physical memory protection) and memory integrity through memory encryption [16, 25, 46], which also protects against physical attackers. Enclaves are not meant to protect against destructive attackers, so DoS attacks are usually out of scope.

Past research has shown that leaking data from enclaves like SGX is possible through so-called 'controlled channel attacks' [26, 63]. This type of side-channel attack assumes knowledge about the application binary that is executed inside an enclave and a fully compromised OS. The attacker monitors page faults through the compromised OS and derives information about the processed data inside an enclave by observing data-dependent control flow changes and data accesses. This is usually done by unsetting the valid bit in the page table entry (PTE) or by observing the accessed (A) and dirty (D) bits in the PTE [8]. Mitigating such an attack requires either manual rewriting and recompilation of an application such that access patterns do not depend on secret data [1] or disabling paging for enclaves on a system level. However, recent work showed that accesses to the PTEs are also susceptible to cache attacks. Mitigations require hardware changes to explicitly flush the cached paging structures used by the translation lookaside buffer (TLB) [8].

3 THREAT MODEL

The CPU and a so-called security monitor (SM), which is a trusted software component running in M-mode, form the trusted computing base (TCB) of SPEAR-V. All other hardware and software of the system is considered untrusted and potentially malicious. Our goal is twofold. First, in line with Intel's SGX model, our design should protect enclaves against software-based attacks mounted by other enclaves or a potentially malicious OS. Second, the host application and OS should also be isolated and protected from a potentially compromised or malicious enclave. Hence we distinguish the following two attacker models:

Compromised OS. An attacker can control the host OS and execute privileged instructions or spawn malicious enclaves. Such an attacker may also mount controlled-channel attacks through manipulation of page tables or interrupts. In such a scenario, SPEAR-V must guarantee that the attacker is not able to directly access memory from other enclaves. Furthermore, a compromised OS may not adhere to the defined behaviour for syscalls. We provide a small API to the host and enclave (e.g., to enter/exit an enclave) through which an enclave can also make syscalls routed through the host application. We assume that the runtime and enclave sanitizes return values to thwart Iago attacks [11].

Malicious Enclave. An attacker-controlled enclave might try to compromise the host application, the host OS, or other enclaves running on the system. In this setting, the attacker is constrained to unprivileged instructions that are executed in the context of an enclave. The attack is successful if the adversarial enclave can read or write any memory that is not explicitly shared with the enclave.

In both scenarios, we consider same-core side-channel attacks, such as branch shadowing attacks [29], as part of the threat model. We assume that the system mitigates against such side channels, as described by Wistoff et al. [60, 61]. As our platform uses a single-core processor, we do not consider cross-core side-channel attacks in our baseline design. For multi-core systems, we assume that additional mitigations such as cache-line-locking [22] are implemented, as designing secure caches is outside the scope of this paper. We exclude physical attacks (e.g., memory-bus snooping, malicious DRAM, physical side channel) from our threat model. However, orthogonal techniques like memory encryption can be integrated to protect against certain physical attacks. We assume that the SM and enclave code are free of bugs.

4 DESIGN OVERVIEW

Our proposed design consists of two key parts that, in combination, provide strong software isolation. First, page-granular memory tags are used to check and enforce access permissions for enclave memory. They are managed by our second component: A software SM running in M-mode that manages enclaves, validates page tables, and handles security-critical operations that cannot be delegated to the untrusted OS. Both user applications and the OS can interact with the SM through a simple API. A general overview of the SPEAR-V design is shown in Figure 1.

Our design is motivated by several observations. 1.) Existing tagged-memory architectures are very fine-grained (typically word-granular). However, for sandboxing and enclave use cases, much

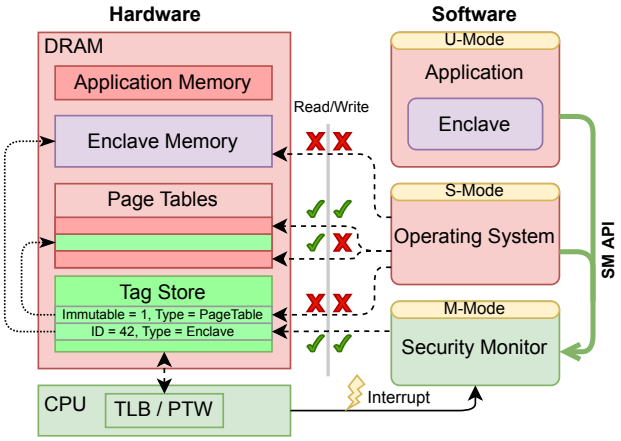

Figure 1: Design overview. The DRAM contains the tag store which holds tags for page tables and normal pages. The SM runs in M-mode and manages the tags.

larger granularities are desired. 2.) Page table updates by the hardware or by the untrusted OS lead to leakage. If we can tag the entire DRAM, we can also tag individual page tables such that they become immutable. Thus, we can let the OS manage them without compromising security. This does not require page table duplication and provides availability guarantees, as enclave pages will always be mapped and available. 3.) Once virtual memory is set up, OSs do not require physical access to the DRAM anymore. Instead, all accesses go through the MMU/TLB, where access policies can be enforced if just a few tag bits are cached in the TLB. This leads to a much higher context-switching performance compared to Intel SGX, which required to flush the TLB at each switch.

Tag Store. For each page in the DRAM, we store a corresponding tag in our tag store, which in-turn is also stored in the RAM. The OS can configure the location and the size (*i.e.*, which parts of the DRAM are covered) of the tag store during system startup. For the taggable memory range, which can cover the entire DRAM, each tag holds information on the ownership of a memory page and is fetched and checked by the PTW during address translation. After a successful translation, we cache a subset of the tag bits in the TLB entry to speed up permission checks on subsequent accesses. Tagging page tables, and thus PTEs, has three key advantages. First, contrary to Intel's SGX, we do not need to duplicate any information from the PTE. Second, we prevent controlled-channel attacks through PTE manipulation by tagging page tables as immutable. This ensures that all modifications have to be verified by the SM. Finally, unlike other designs, which require all page tables to be in a specific DRAM region [22], we do not require any OS modifications, nor do we affect the performance of unprotected page tables.

Paging. To prevent certain (e.g., remapping) attacks, enclave architectures need a trusted mapping from virtual to physical pages. This often means that the page tables are duplicated or moved into a secure area, which adds overhead and software complexity. A defining feature of SPEAR-V is the ability to securely use unmodified OS-based paging for enclaves, which also eliminates the need for self-paging. We achieve this by allowing page tables to be tagged as immutable, such that write accesses are only possible through the SM. In this setting, the OS can still create PTEs for an enclave. However, since these mappings cannot be trusted, the SM verifies them and sets their respective page tables to immutable. Thus, the entire address translation process, from the virtual address and the SATP register, which points to the root page table, to the final physical page, becomes trusted (cf. Section 5.4). When adding a page to an enclave, the SM checks that this page does not already belong to another enclave by reading its corresponding tag, thus ensuring that each enclave page has a unique mapping. If the page table of that page is not yet immutable, meaning no valid enclave mappings exist yet, the SM also ensures that no other PTE exists that maps to the same physical page by validating all PTEs from that page table. This step is repeated for all higher-level page tables until the "root" page table, as referred to by the SATP register, is reached. Once a page table is set to immutable, it is guaranteed not to contain invalid mappings. The final part of the translation process that needs to be secured is the SATP register. We link an enclave to a single process (*i.e.*, address space) by checking the contents of this register when entering an enclave. At runtime, the hardware ensures that enclave pages can only be accessed through verified and immutable PTs. Since our security is based on checks done by the PTW & TLB, our hardware also monitors the SATP register for changes that would disable virtual memory, which in turn bypasses these checks. To avoid attacks in which an OS observes memory access patterns through PTE bits (*i.e.*, accessed and dirty), the hardware also respects their immutable nature and does not update them. Note that while swapping algorithms typically depend on these bits, not updating them for enclaves does not impact the swapping performance of normal/host applications. As the OS cannot unmap enclave pages, we also protect against attacks that infer enclave secrets via monitoring of page faults. Thus, we protect against side-channel attacks relying on paging information [53].

Shared Memory & Nesting. Enclaves can share memory through the commodity shared memory interface in combination with SM API calls. Shared pages that are tagged by the SM experience the same hardware-based protection as regular enclave pages and can only be accessed when given explicit access by the respective owner. Since each enclave is also a sandbox and to facilitate more complex use cases, we also allow them to be arbitrarily nested. By isolating parts of an enclave memory region from the enclave itself, enclaves can spawn protected child enclaves that are again protected from their host (*i.e.*, the parent enclave), but also the other way around.

Given the above components and features, we formulate the following invariants that guarantee the security of our design:

$\mathcal{I}1$: The tag store is not accessible from U- or S-mode. As the tag store plays a central role during access permission checks, we must guarantee that memory tags are never writeable by unprivileged software or a possibly malicious OS. The tags themselves currently do not contain secrets. However, to prevent possible future side channel attacks we disable read access to the tag store.

$\mathcal{I}2$: Private enclave memory is only accessible for the owner enclave through a unique mapping. Pages tagged as enclave pages can only be accessed by the enclave with the respective ID, given that the memory is not explicitly shared. Each enclave page may only be reachable via exactly one valid virtual-to-physical mapping. This means that each enclave page may only be pointed to by exactly one valid immutable page table. With these constraints,

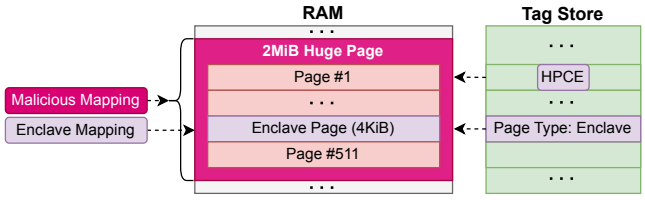

Figure 2: When tagging a enclave page, the SM also tags the 2MiB and 1GiB regions with the HPCE bit. Thus, a malicious OS cannot map the huge page to bypass the access checks.

we guarantee the confidentiality and integrity of enclave memory while also protecting against aliasing attacks.

$\mathcal{I}3$: A shared memory region is accessible *iff* an enclave has received permissions from its owner. As we also protect enclave memory that is shared with other enclaves, we must guarantee that only enclaves that have received the corresponding access permissions may access a shared memory region. In the presence of potentially malicious enclaves, we must ensure that enclaves accessing shared memory have received access through a legitimate and trusted channel, *i.e.*, the SM.

$\mathcal{I}4$: The mapping (position and ordering) of enclave pages is immutable and unique. Once pages are committed to an enclave, their position and order may not be altered. Furthermore, all enclaves that are accessing the same shared memory region must map the region in the same (virtual) position and order as the enclave sharing that region. This constraint helps to avoid attacks that rely on the reordering of enclave pages by the OS.

$\mathcal{I}5$: (Unshared) host memory is inaccessible to enclaves. To account for potentially malicious enclaves, enclaves are prohibited from accessing host memory. Through this invariant, each enclave is confined to perform memory accesses only within its own enclave memory region and potentially available shared memory regions. A host can only share memory with an enclave that has been verified by the SM. Within an enclave, accesses to untagged memory will fail, as the hardware cannot check if the access is legitimate.

5 HARDWARE DESIGN

During translation from virtual to physical addresses, on a TLB miss, our hardware PTW loads the tags of the requested page and all related page tables. A small subset of the memory tag is cached in the TLB to speed up future access checks. Additionally, we add new CSRs that are only accessible by the SM. They control the behavior of the PTW and the TLB regarding tag fetching and access permission checks. For PTEs or memory that does not reside in the taggable memory region, tag loads and access checks are skipped. In this section, we detail these necessary changes to the hardware.

5.1 Memory Tagging

At its core, SPEAR-V relies on page-granular memory tagging. Each memory tag resides in the special tag store DRAM region that is inaccessible for all software except the SM.

Protection. To protect the tag store and thus enforce $\mathcal{I}1$, RISC-V physical memory protection (PMP) may be used. By configuring the PMP registers such that the tag store may only be accessed from M-mode, thus blocking reads and writes from U- and S-mode. However, this would detract one or two entries from the already limited number of available PMP entries. Hence, since the tag store can cover the entire DRAM, it can also protect itself from unprivileged access by configuring the tags of the memory pages where the tag store lies. The SM sets this up at boot time and falls back to PMP-based protection if the tag store is unable to protect itself due to its memory layout. This self-protection works analogously to protecting enclave pages: We set the owner/ID field of the tag to a special "enclave" ID, which we reserve for the SM itself. Thus, the SM, which owns the tag store, can be viewed as a separate enclave, and its data is protected using the same primitive.

Huge Pages. In our design, we reserve one tag for each 4KiB of memory. As the OS can also map so-called huge pages (e.g., 2MiB or 1GiB), they require special consideration. Given a 4KiB enclave page, the OS might try to map a huge page that contains this enclave page to another virtual address. It is vital that memory accesses to that smaller enclave page using the huge page mapping are subject to the same permission checks as accesses targeting the enclave page directly. For this, our tag contains a so-called 'huge page containing enclave' (HPCE) bit, which indicates if the memory range of this particular huge page contains pages that are relevant for security (e.g., enclave pages). This is shown in Figure 2. Due to memory alignment constraints of huge pages, whenever the SM tags a critical 4KiB page, it can also set the HPCE bit for both 2MiB and 1GiB pages that contain this 4KiB region. If the OS maps this huge page, the PTW will detect the HPCE bit and perform an additional tag fetch for the affected 4KiB range. It also downgrades the resulting TLB entry to a 4KiB entry. This preserves compatibility with OSs that may use huge pages for managing their page tables and only affects the TLB miss rate negatively for memory ranges that actually contain enclave data. We also consider the case that an enclave itself tries to obtain a huge page mapping. As all 4KiB pages enclosed by the huge page belong to the enclave, we must tag all contained pages separately, thus resulting in 512 or 512^2 subsequent tag writes. This approach only impacts the performance of the mapping operation itself. Once the tags are set, accessing the huge page yields the same latencies as access to a tagged 4KiB page. Alternatively, since the tag also stores the page size, the PTW could detect any malicious access attempts of huge enclave pages without the necessity to tag all enclosed 4KiB regions. This, however, would increase the TLB-miss latency, since the TLB always has to check the 2MiB and 1GiB tags as well.

5.2 Memory Tag Fields

In the following, we detail the different fields of our memory tag and their use. A visual representation of the fields, including those that are cached in the TLB, is given in Figure 3.

Validated (1 Bit). When the host adds a new page to an enclave, this field is not set until the enclave acknowledges this mapping. Only when it is set will the page be usable by an enclave. Thus the enclave will always be aware of any page that it has access to.

ID (16 Bit). This field stores the ID of the enclave that the page belongs to. With this field, the TLB and PTW can validate that the accessed page is owned by the currently running enclave. This ensures that $\mathcal{I}2$ and $\mathcal{I}5$ hold true for all memory accesses that are not aimed at shared memory. The size was chosen to be aligned

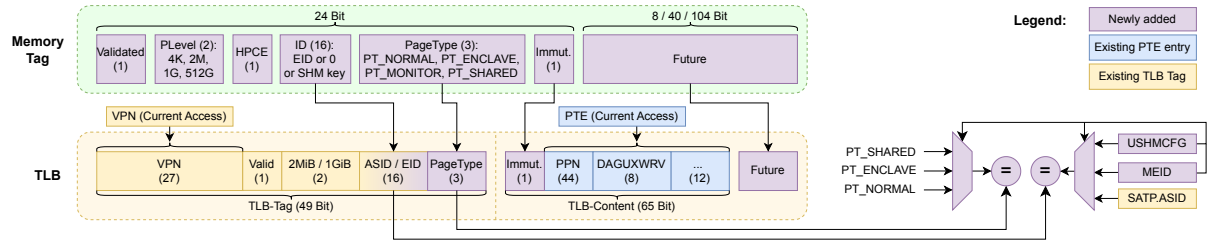

Figure 3: Overview of how memory tag fields are stored inside TLB entries. For enclave pages, the ASID field in the TLB entry stores the enclave ID. We add a field for the page type to the TLB tag, which, during a TLB lookup, decides whether the `ASID/EID` is compared to the `ASID` field inside the SATP CSR, the `MEID` CSR, or the `USHMCFG` CSR.

with RISC-V's address space identifier (ASID), which is also 16 bits wide. Since we store this ID in the same position in the TLB where the ASID is stored, we avoid adding additional bits. Thus, currently, we are limited to a maximum number of parallel valid enclaves of $\approx 2^{16}$. However, if the cost of increasing the TLB size is acceptable, this field can be enlarged to support a larger number of enclaves. E.g., we can use the 8 reserved bits to extend the ID to support over 16M enclaves. In Section 8 we explore the area overheads introduced when adding such additional bits to the TLB entries.

Immutable (1 Bit). This bit is essential for SPEAR-V as it allows an untrusted OS to manage the paging structures of enclaves. Once a page is marked as an enclave page, all its page tables in the hierarchy are validated by the SM and subsequently tagged as immutable. Write accesses to immutable pages trigger an exception that traps to the SM, where they are verified. Thus, all modifications of the page tables of an enclave are verified by the SM. This prevents a malicious OS from tampering with the page tables of an enclave once the mapping is established, hence enforcing $\mathcal{I}2$ and $\mathcal{I}4$. Furthermore, the SM may use this bit to protect its own code or data.

Page Level (2 Bit). In this field we store the size of the tagged page. Thus, the tag holds information on whether the page is a 4KiB page, a 2MiB page, or a 1GiB page. This can be used in addition to the HPCE bit to enable huge enclave pages without the necessity of tagging every 4KiB page contained inside the huge page. E.g., a 1GiB mapping only needs 1 tagging operation instead of 512^2. The PTW fetches tags of the corresponding 2MiB- and 1GiB-aligned huge pages during every translation.

Page Type (3 Bit). We distinguish between five different page types and allow for 3 future page types. Pages marked with `PT_NORMAL` belong to the host application or the OS and are regularly accessible. For enclave pages, we use the `PT_ENCLAVE` page type. Pages that belong to the SM are marked as `PT_MONITOR` and are only accessible from M-mode. When an enclave or a host application decides to share a page, the corresponding page is marked as `PT_SHARED`. Finally, `PT_PAGETABLE` is used to mark immutable page tables if they contain an enclave mapping.

HPCE (1 Bit). This helps against huge page attacks (Section 5.1).

5.3 Control and Status Registers

To manage the tag store and enforcement of access permissions, we add the following control and status registers. Registers that start with `M` are only writable by the SM, whereas `U` registers are writable by the enclave itself.

Tag Store. For the tag store itself, we add the following four registers, which are configured at boot time by the SM. First, `MTAGMODE` defines the tag size that is used. While our enclave design only requires 24-bit tags, for evaluation and exploratory purposes we currently support 0-, 32-, 64-, and 128-bit tags. When set to 0-bit, running enclaves is not possible, however, there is also no memory overhead introduced since no tag store is necessary. The `MTAGBASE` register holds the physical address of the tag store, and the `MDRAMBASE` register holds the physical start-address of the protected/taggable address range. Both are aligned by the chosen tag size and the largest supported huge page size, respectively. Finally, `MDRAMSIZE` defines the size of the taggable DRAM region. Based on these registers, the PTW calculates the address for the tag lookups.

The tag store may cover the complete DRAM or only a contiguous, huge-page-aligned, subset of it. Furthermore, it is not necessary that the tag store region resides in the taggable DRAM region itself, as we can fall back to PMP-based protection. During boot, the SM chooses an appropriate protection method (either PMP or using the tag store for self-protection) and configures the necessary PMP registers or tags accordingly. For the latter, the SM tags all pages of the tag store itself such that they are only accessible by the SM.

The tag store may also cover a larger region than just the DRAM to protect accesses to MMIO regions. E.g., this allows an enclave or the SM itself to exclusively access an (MMIO-)programmable interrupt controller, which, combined with an enclaved scheduler, can give strong availability guarantees as shown by Alder et al. [3].

Enclave Operation. For basic enclave operation, we add two more registers. The `MEID` register is set to 0 outside an enclave or otherwise contains the current enclave's 16-bit ID. It is used by the caches, particularly by the TLB and PTW (cf. Section 5.4), to enforce that enclave pages can only be accessed by their respective enclave ($\mathcal{I}2$ and $\mathcal{I}5$). The `MTCS` register holds the unique physical address of the thread control structure, a page that contains information about the current enclave thread.

Memory Sharing and Host Memory Isolation. Host memory is protected because an enclave can only access host memory if it has been explicitly shared. It is treated the same as shared memory between enclaves. For this, we add the `MSHMCFG` register, which is split into 3x 16-bit key fields and 3x 2-bit (*i.e.*, read and write) permission fields, respectively. Shared memory regions are represented by a *key*. Within our tag structure, enclave IDs and shared memory keys are stored in the same position (*i.e.*, the ID field), since a memory page can only be used for either one at the same time. When shared

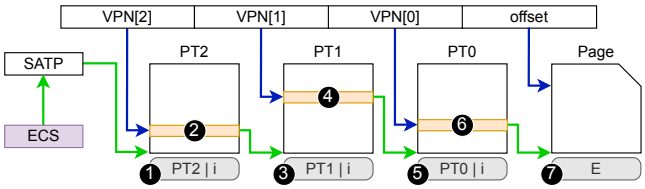

Figure 4: Illustration of a page table walk within an enclave.

memory regions are set up, the SM adds the respective keys to the enclave's metadata page (ECS) and the `MSHMCFG`. Enclaves may have access to more shared memory regions than fit in this register. In case a memory access to such a memory region happens, it triggers an exception that traps to the SM, where the SM looks up accessible keys in the ECS and adds it to `MSHMCFG` register.

To support more fine-grained protection and sandboxing capabilities, we additionally define the `USHMCFG` register, where the enclave itself can, at any point in time, decide which of its available keys it allows access to. This allows an enclave to prevent accidental information leakage, similar to x86's SMAP (supervisor mode access prevention), and also allows for memory safety applications and sandboxing within an enclave itself. The register has the same format as `MSHMCFG` and can hold zero to three active keys and their desired permissions. These slots can be empty, meaning that no shared memory is accessible, or they may contain the shared memory region keys, in which case only these memory regions are accessible. When shared memory is accessed whose key is not in the `MSHMCFG`, the SM will copy the value from `USHMCFG` assuming all its keys are valid. If an enclave wants to access both shared and private memory at the same time, it can add the ID "1", which is never used as an enclave ID or shared memory key, but instead represents the enclave's own memory. Thus, an enclave can choose to disable access to its own memory completely when accessing shared memory to prevent data leakage. Conversely, an enclave can also clear the register to only allow access to its own memory. Read and write permissions can be given independently for each key. Execute permissions are not available on shared memory since enclave code can only exist in the private (attested) enclave memory. A special case is memory shared by the host itself, which always has the ID "2", and is accessible to any enclave running in the same address space, assuming that the shared memory region is mapped.

5.4 TLB & Page Table Walker

We extend the PTW such that tags are fetched during the translation. There, the PTW always checks if the physical address that is accessed next is covered by the tag store. For enclave memory, all levels of the paging hierarchy must reside in the taggable region. Apart from that, we do not impose any restrictions on the combination of taggable and non-taggable memory. As each translation of taggable memory results in additional tag lookups, the translation latency is slightly increased. To lower the performance impact of SPEAR-V on non-enclave memory, the OS can explicitly use pages that are not covered by the tag store for applications that do not use enclaves. Thus, the PTW would not need to perform any additional fetches, resulting in no performance loss.

Figure 4 shows memory translation with tagged memory. First, the PTW reads the PPN of the root of the paging hierarchy from the SATP register and uses it to fetch the tag of the first page table (❶). After that, it uses the PPN and part of the virtual address to fetch the page table entry pointing to the next page table (❷). The PPN in this PTE is then used to first fetch the tag and then the next PTE. This is repeated until a PTE pointing to a leaf page is found (❸–❻). The PTW keeps track of the immutability of the fetched page tables at every paging level. In the final step, the PTW fetches the tag of the leaf page and performs access checks based on both the tags of the related page tables as well as the tag of the leaf page itself (❼). If the leaf page belongs to an enclave, all of the corresponding page tables have to be immutable. If this check succeeds, the PTE of the leaf page and important tag bits that are required for access checks are stored in the TLB. For this, we minimally extend TLB entries by 4 bits, which avoids re-fetching the tags on future accesses.

The MMU performs access checks for all TLB hits based on the contents of the page type, the ASID/EID field, and the immutable bit. If the host or the OS tries to access an enclave page, the MMU triggers an exception that traps into the SM. The same applies to enclaves that try to access (non-shared) pages of other enclaves or the host. Since alias mappings might exist in the TLB and to ensure consistency, we flush affected TLB entries when tags are written. Also, to avoid potential side-channel attacks based on memory fetches initiated by the PTW [8], we mark such memory requests as uncacheable within an enclave.

6 SOFTWARE DESIGN

In our design, the SM orchestrates the entire enclave lifecycle and manages their permissions. The OS, host application, and enclaves can use so-called SM API calls, which is implemented as a separate instruction, to manage enclaves and their shared memory. These are similar to syscalls, but they are handled by the SM, in the M-mode, instead of the OS. In this paper, we focus on the isolation mechanism and core features of a SM. In the future we foresee that other software features like attestation or advanced syscall filtering will be ported from other enclave systems ([22, 28, 30]) or vice versa. However, re-implementing these features would impose significant engineering efforts and is thus outside the scope for this paper.

In the following, we give an overview of our SM and its most important functions concerning the enclave lifecycle.

Security Monitor. The security monitor (SM) is software running in M-mode and is part of our TCB. All security-critical operations, such as enclave and tagging operations, must be done through the SM. For this, it provides several API functions that can, similar to syscalls, be called from the enclave, host application, or the OS. We currently provide `E_CREATE` for enclave creation, `E_DESTROY` for enclave destruction, `E_ADD`/`E_REMOVE` for adding/removing enclave pages, `E_ENTER`/`E_EXIT` for entering/exiting enclaves, as well as `E_SHARE` and `E_REVOKE` for managing shared memory.

Enclave Creation. Before an enclave can be started, it has to be created with `E_CREATE`. Here, the host first reserves memory for the code, data, and stack pages of the enclave, as well as pages for managing the enclave's metadata (*i.e.*, the so-called ECS (enclave control structure) and TCS (thread control structure)), and supplies it to the SM. Each enclave has one such ECS, and each enclave thread has a TCS that also points to the ECS. The SM assigns a new

enclave ID and protects/tags these pages accordingly. The ECS is tagged such that only the SM can access it, while the ownership of the other pages is transferred to the enclave itself. Once the pages belong to an enclave, the host can no longer access them or change their mapping. When creating an enclave, the SM stores the current SATP value in the ECS such that only the same address space can be used to start the enclave. It also marks any page tables that contain PTEs related to enclaves as immutable. This ensures a trusted mapping of virtual to physical addresses. Furthermore, by tagging a physical page as an enclave page, it is also ensured that exactly one valid and immutable mapping for the given physical page exists. Once an enclave is destroyed, its ID can be reused since the SM ensures that no physical page, tagged with this ID, exists.

Enclave Execution. Once an enclave is set up, the host application transfers control to the enclave by invoking `E_ENTER`. Upon entering, the SM stores the host execution context on the TCS page and sets the `MEID` as well as the `TCS` register, which switches the CPU into the enclave mode. From there, the enclave starts at a predefined entry point. Like a function call, `E_ENTER` accepts a arguments that are passed to the enclave and, when exiting, returns a return value.

Interrupt Handling. In our design, the SM may enforce a flexible policy that can disallow external interrupts on a given core and also enforce a minimum timer interrupt frequency. If an interrupt occurs during enclave execution, the OS may want to schedule another process on that core. During enclave execution interrupts always trap to the SM, which then stores the execution context (*i.e.*, CPU registers) on the TCS page before restoring the host context from TCS. Furthermore, when entering or exiting an enclave, the SM also flushes any known microarchitectural buffers that may be prone to leakage. For this, we integrate the work of Wistoff et al., which provides a separate flushing instruction for the CVA6 core [60].

Enclave Nesting. For a flexible and future-proof TEE design, we anticipate that enclaves need to spawn isolated environments themselves as well. Such situations can arise when an enclave depends on third-party IP that is required to be executed in its own TEE or sandbox. In our design, enclaves can spawn arbitrary child enclaves with no performance impact. Assuming an enclave has access to enough memory (either given at `E_CREATE` or via `E_ADD`), it can call `E_CREATE` itself with no intervention or knowledge from the OS or host application. In this case, ownership of the necessary pages is simply transferred from the outer enclave to the inner enclave without changing host page tables. For fast context switches, each TCS page stores three different pointers. First, a pointer to the parent TCS, which created/entered the current enclave. Second, a pointer to the root enclave's TCS, which is the outermost enclave that was created by the host application. And third, a pointer to the TCS that is currently executing. During context switches (e.g., when a timer interrupt happens), we only need to look up the root level TCS, update the currently executing enclave, and restore its own host context. Any intermediate enclaves are skipped and only needed when explicitly exiting an enclave, which returns control to its respective parent. Memory accesses within a nested enclave are treated the same as others and experience no slowdown. During context switches, independent of the nesting level, only the root level TCS structure is updated. Each nested enclave page is managed by the same paging structures as top-level enclaves or host applications. Thus, even for arbitrarily deep enclave nestings, memory accesses never experience performance deterioration due to increased page translation latencies. Also, the OS does not learn of the existence of nested enclaves because it cannot directly inspect the tag store to learn the true owner of the page and the SM handles transitions such that the host application only interacts with the outer enclave. Nested enclaves follow the same lifecycle as top-level enclaves, with one exception: Before parent enclaves can be destroyed, they must also relinquish control of or destroy their child enclaves. In case the number of child enclaves, which are tracked in the ECS, is not zero at this time, the SM can simply traverse the processe's page tables to find and destroy the remaining child enclaves and their respective pages. These pages are easily found since page tables that contain enclave pages are tagged as immutable and also the enclave pages themselves are tagged.

Shared Memory. Our SM provides two API calls for memory sharing. First, `E_SHARE` takes the virtual start and end addresses of the memory to be shared, a permission flag, and the receiving enclave (*i.e.*, ECS page) as arguments and returns a key that is unique for each shared memory region on the system. The SM tags shared pages with the page type `PT_SHARED` and sets the ID field to the respective key. If called by the non-enclave host application, the key will be "2", to which any enclave within the same process has access. Enclaves can only access host memory if it has been shared with them using this API call ($\mathcal{I}3$). If the target enclave runs in another address space, the host is responsible for mapping that shared memory region using the same virtual addresses. Enclaves can only receive a shared memory request if the SM verifies the mapping, makes sure it is immutable and marks the key as valid within the ECS for the given enclave. During this step, the SM ensures that the mapping for each physical and virtual page in a shared region is also the same for the recipient's address space. Each enclave can further share each region it has access to with other enclaves to the extent of their own respective permissions. The second API call is `E_REVOKE`, which allows enclaves and hosts to remove their own access to a shared memory region. Our prototype implementation allows each enclave to have access to 64 different shared regions. However, the maximum number is only limited by the size of the ECS structure, which can be extended in practice. Each of these entries also tracks its owner, recipient, as well as a reference counter to ensure all uses can be revoked before cleanup.

Swapping. To prevent controlled-channel attacks, only pages from enclaves that are currently not running can be swapped out. If an enclave page is swapped out, the enclave becomes unrunnable. Before giving the OS access to a page, such that it can be swapped out, the SM would encrypt it and add metadata in the ECS in order to swap it in again. The SM itself does not implement any swapping policies. Instead, the host OS can decide to swap out any enclave page apart from ECS pages as long as they are not currently running.

7 SECURITY ANALYSIS

Enclave and shielded execution systems provide mechanisms to protect data from being leaked or corrupted. In this section, we detail how SPEAR-V provides the confidentiality and integrity guarantees we introduced with our invariants in Section 4.

Malicious Memory Mappings. As our threat model assumes a possibly compromised OS, we must account for the fact that the OS

can generate arbitrary malicious entries in the page tables of a host application. This is because SPEAR-V does not have (duplicated) page tables in a separate trusted region but instead uses the ones set up by the OS. Thus, we have to protect against attacks that rely on any kind of malicious page table entries. This includes remapping enclave pages at different addresses or unmapping them entirely. As described by our invariants $\mathcal{I}2$/$\mathcal{I}4$, enclave-private pages must only be accessible by the enclave itself, and their unique mapping (virtual to physical translation) must not change. Each enclave-private page can only have one valid virtual address within a single address space. When creating enclave pages (e.g., using `E_CREATE` or `E_ADD`), the SM guarantees the correctness of the mappings by performing several checks. In the following, we denote mutable page tables (*i.e.*, pages whose tags have the immutable bit not set) as mPT and immutable page tables (*i.e.*, pages whose tags are marked as immutable and as `PT_PAGETABLE`) as iPT.

An enclave page is created by tagging an unprotected page (that is already mapped) as well as the associated page tables. This requires two checks, performed by the SM, to avoid violating $\mathcal{I}2$ & $\mathcal{I}4$. First, when an enclave page is created, its respective (parent) page tables are only set to immutable iff no other PTE contained within points to an enclave page or an iPT. Tagging such a PT as immutable would create an alias mapping for another already established and valid mapping. Because of this, the SM must scan through all PTEs of all PTs that are relevant for the translation (up to $3 \cdot 512$ accesses). Second, the SM must verify that the page is not already mapped within another iPT. If a page is mapped more than once and one of these mappings is within an iPT, tagging this page would create an alias mapping. For this reason, the SM traverses all iPTs beginning from the `SATP` and checks if any PTE (within an iPT) points to the page that is currently being added. In the worst case, this might require up to 512^n lookups with n being the number of memory hirearchy levels, but this is both unrealistic as well as optimizable. The amount of lookups is limited by both the number of valid PTEs and the number of iPTs. Since page tables are usually very sparse, higher-level page tables typically have very few "valid" (read: existing) PTEs. Furthermore, the host might cluster all enclave pages within a process such that they have virtual addresses that are close to each other, thus limiting the number of iPTs. Furthermore we implemented an optimization to avoid scanning iPTs that only contain enclave pages. At `E_ADD`, we set the "validated" bit for such last-level iPTs. Analogous, the bit in higher level iPTs can be set if all its entries only point to such validated iPTs. During the recursive iPT scan, we omit paths with validated iPTs, as they cannot contain (alias) mappings to regular pages.

When entering an enclave, the SM ensures that it is run in the same address space that it was created in by comparing the `SATP` with the value stored in the ECS. When accessing enclave pages (pages tagged as `PT_ENCLAVE`), the PTW checks that the ID of the currently running enclave (`MEID`) matches the ID of the tag and that each used page table is tagged as immutable and `PT_PAGETABLE`. In the following, we describe the possible ways an OS can try to create invalid (alias) mappings and how we protect against this.

Attack 1. The simplest type of aliasing is to generate a duplicated mapping in a mutable PT for a page that will later be added as an enclave page. Such an attack is depicted in Figure 5 (left) as the additional connections to the leaf page (1, 2). When the leaf page (P) is added to an enclave, one of the virtual addresses leading to the page is passed to the SM. As mentioned above, the SM scans the entire page table hierarchy for other mappings to the page that is being added. If the SM finds another PTE referencing the same physical page number, the API call fails, the PT remains mutable, and the page is not added as an enclave page. In the benign case that no duplicate mapping is detected, the SM marks the page as an enclave page and the relevant PTs as immutable.

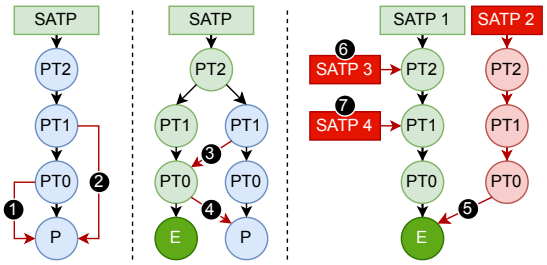

Figure 5: Possible malicious page table setups that could lead to alias mappings. Green PTs are immutable.

Attack 2. Two more complex cases are depicted in Figure 5 (center). In the first case, a mPT points to an iPT that is part of an enclave mapping (3). During a TLB miss, the PTW asserts that all associated PTs are immutable, before a valid TLB entry for an enclave page is created. Thus, in this attack scenario, the access would fail. However, if an application tries to add the leaf page (P) to an enclave, this would create an alias mapping (4) for the existing enclave page. The SM detects this before adding the page because the mutable PT1 has a PTE pointing to an immutable PT (3), which is not allowed. In the second case, an iPT contains a PTE for a normal page (4). If the application tries to add this page to an enclave using the mapping on the right, checking the entries of all directly involved PTs is not sufficient to detect the created alias mapping. For this reason, the SM traverses all iPTs and scan for potential alias mappings.

Attack 3. The OS might try to access enclave pages through other processes, as shown in Figure 5 (right). In the simple case (5), the enclave page is simply mapped as a leaf page in another process (SATP 2). More complex cases (6, 7) involve malicious processes with SATP values that point to existing iPTs. Our design mitigates these attacks through a combination of the memory tag and the ECS. An access with a mapping like (5) would simply fail because the PTs are not immutable (cf. Attack 2). As described in Section 4, an enclave page is only accessible if the current value of the `MEID` register matches the `ID` stored in the tag. This implies that an enclave page is only accessible if the current execution context matches that of the enclave owning the page. For this reason, accessing the enclave page through mapping (6) would trigger an exception. Furthermore, using mapping (7) also results in an exception, but for a different reason: The page level, stored in the tag of PT1, does not match the PTW's expected value. When switching the execution context to an enclave, the SM checks that the current value of the `SATP` matches that stored in the ECS of that enclave. As the ECS is inaccessible for all software except the SM, the OS cannot forge the entry in the ECS. The SM detects the mismatch when entering (or continuing execution of) the enclave and prohibits enclave execution, thus protecting against such malicious mappings.

In short, before tagging a PT as iPT, the SM ensures that none of its PTEs points to an already existing enclave page or an iPT. And before tagging a page as an enclave page, the SM ensures that, in the current address space, no existing iPT has another PTE pointing to that new page. Thus, within an address space, there exist no immutable and valid double mappings for enclave pages. When removing an enclave page, the SM checks all PTEs in the affected iPTs. In case none of them point to an iPT or enclave page anymore, the iPT itself is converted to a mutable PT. The hardware TLB & PTW checks, the checks done by the SM during enclave entry, and the verification of page tables allow us to protect all steps of the address translation process, as indicated by the green arrows in Figure 4, while the mappings are still managed by an untrusted OS.

Physical Memory Access. Attackers with physical access to the memory are out of scope, however orthogonal memory encryption solutions [58] can be used to also prevent such attack vectors.

Our security is based on checks done by the TLB & PTW. For this, we require virtual memory to be active. Most OSs, like Linux, only require physical memory access during boot. Once virtual memory is set up, they typically also map the entire DRAM into the virtual memory. On RISC-V the `SATP` register decides if memory accesses are using virtual memory or not. Once tagging is active, our hardware monitors writes to this register such that virtual memory cannot be disabled anymore without the SM noticing. This does not pose any performance or compatibility issues with Linux and normal process scheduling since, while this register is updated at every process context switch, the bits that decide if virtual memory should be enabled or not will always stay the same. Hence, once tagging is active, it is ensured that any memory access will adhere to the permissions set within the tags.

DMA & IOMMU. Devices with direct memory access may access the DRAM directly, which would bypass our checks. Hence, we deny any DMA request belonging to a memory range covered by the tag store. The OS can set up the tag store, such that part of the DRAM is not covered by it, to support devices that may require direct memory access. Alternatively, cores with an IOMMU can also implement the same security mechanism as the normal MMU.

Indirect Memory Attacks. Indirect accesses are prevented by secure interruption. At each interrupt or exception during enclave execution, the enclave context (*i.e.*, CPU registers) is written to its secure ECS page. Also, microarchitectural per-core buffers are cleared by the SM before continuing the execution of untrusted code. While the CVA6 CPU does not have a shared (last level) cache, and preventing attacks on such shared caches is outside the scope of this paper, SPEAR-V is compatible with the countless already existing cache side-channel mitigations [20, 27, 34, 43, 44, 54, 59].

Memory safety vulnerabilities within the enclave code can be another attack vector. This is generally an active research topic on its own. However, our `USHMCFG` feature, as well as nested enclaves, help enclave developers in minimizing the potential for leaked data. It allows the enforcement of the privilege of least principle, even within an enclave, backed by strong hardware guarantees.

Iago Attacks. Given a malicious host OS, it is possible that the attacker tries to compromise the enclave state by returning forged values from syscalls. E.g., it may return already mapped enclave memory when calling `mmap` instead of newly mapped memory, which could lead to corrupting its own memory when writing to it. We assume that the SM and enclave (runtime) already sufficiently check any values returned from the OS. An enclave can also use the `USHMCFG` register to temporarily disable access to enclave memory, e.g., while writing to the newly mapped public memory region, to help thwart such attacks.

Code Reuse and Memory Corruption Attacks. Code reuse attacks, such as ROP [47] and JOP [6], link existing code *gadgets* to an instruction chain that performs operations beneficial to the attacker. The initial foothold for such an attack is a memory corruption vulnerability that allows the attacker to alter the control flow of the program. In the past, such attacks were proven effective against TEEs and trusted runtimes [5, 9, 13, 45]. While SPEAR-V is not specifically designed to protect against memory corruption or code reuse attacks, it still provides a higher degree of protection for enclaves than for non-enclave code. As each enclave only holds mappings to enclave memory, it is impossible to craft a gadget chain that escapes the enclave memory region [45]. Trying to execute gadgets outside the enclave memory will result in a page fault on a page that is considered untrusted. Thus, the SM detects the control flow violation and can terminate the enclave. ROP and JOP attacks that are confined to enclave memory are, however, not detected by this mechanism. It is still possible to reduce the impact of such attacks by dividing an application into multiple child enclaves. One could, for example, confine each function (or groups of related functions) to a separate child enclave. Doing so reduces the impact of JOP and ROP attacks significantly since, it is impossible to escape the memory of a child enclave. Thus, the gadget space is limited to the gadgets found in the child enclave. Depending on the usage of trusted runtimes, it is still possible to perform memory corruption attacks that target the runtime instead of the enclave code itself [5]. A holistic protection against such attacks would require a verified error-free runtime as well as error-free enclave code.

Protecting Host Memory. In our design, host memory is never accessible from a (sub-)enclave by default. The host has to explicitly convert memory to shared-memory regions before both the host and an enclave can access them. Using the API calls for shared memory, the sharer (*i.e.*, host) has full control over the permissions of each shared region.

8 EVALUATION

In the following, we evaluate our design using a mix of micro- and macro-benchmarks. We synthesized our modified CVA6 for a Genesis 2 FPGA and used this to evaluate performance within a Linux environment. Our design relies on memory tags that need to be fetched at every TLB-miss. This increases the latency of every initial access to a page that is inside the taggable memory region. Subsequent accesses suffer no penalty since all relevant tag bits used for access checks are cached inside the TLB. For evaluation, we use 64- and 128-bit tags to also account for potential future bits stored in the tags. Smaller tags (24- or 32-bit) would only save memory, but not increase performance due to the memory interface width, which is usually larger than 64-bit. We measure the resulting memory access times and throughput using LMbench [36]. For benchmarking overall performance overhead and to compare enclaves with nested enclaves, we use Embench [23]. Finally, we

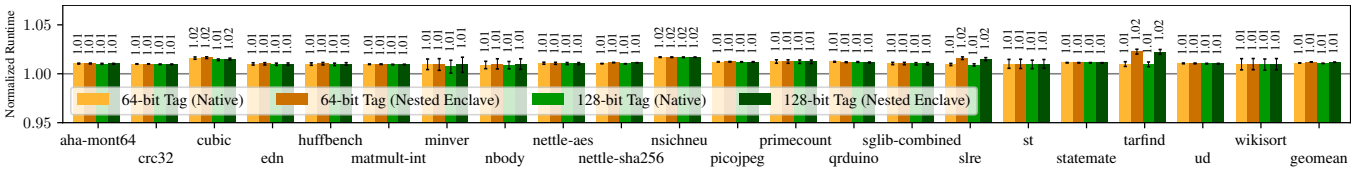

Figure 6: Relative runtime overhead of all Embench benchmarks for both 64 and 128-bit memory tagging setups in- and outside an enclave compared to baseline performance without memory tagging.

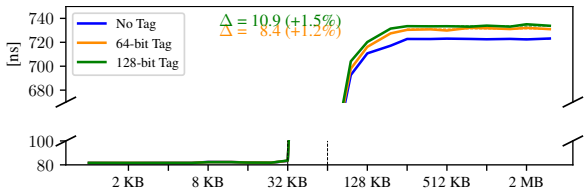

Figure 7: Memory latency for different tag sizes.

	No Tag	64-bit Tag	128-bit Tag
rd	22.00	21.86 (-0.63%)	21.79 (-0.95%)
wr	56.00	55.98 (-0.04%)	55.95 (-0.09%)

Table 1: Memory bandwidth in MB/s for different memory tag sizes and LMBench *bw_mem* testbenches.

use microbenchmarks to evaluate the overhead for modifying immutable page tables, swapping, and page table scanning.

Embench. Embench is a collection of benchmarks for embedded systems. They are ideal for benchmarking enclave systems because they do not require any syscalls to function. For a baseline, we run the benchmarks without tagging and without enclaves. Then, we run them within a nested enclave, both with 64 and 128-bit tagging, and repeat this 50 times. For nested enclaves, we take the total time it takes to enter an enclave, creating and entering a nested enclave, within which we start Embench, return the result to the outer enclave, and finally return it to the host application. Figure 6 shows the relative performance overheads of all Embench benchmarks both out and inside an enclave compared to the baseline with no memory tagging. To account for outliers (e.g., due to Linux's scheduling), we only plot the 95% percentile. The introduced overhead stems mostly from timer interrupts, where the SM saves and swaps the enclave's and host's states. Also, for memory that is tagged, the TLB performs one additional memory access per translation level for 64-bit tags and two additional accesses for 128-bit tags. Across all benchmarks, the overhead caused by memory tagging alone is 1.1%, while the nested enclaves show only a 1.2% runtime overhead.

LMbench. To simulate memory-intensive workloads, we show the worst-case overhead by using LMbench's *lat_mem_rd* benchmark to measure memory latency and the *bw_mem* benchmark to measure memory bandwidth. Figure 7 shows these memory latency differences. There are two significant latency jumps. The first occurs at 32 kilobytes which is the cache size of the CVA6 CPU. Bigger working sets lead to cache misses that increase latency. The second jump marks the end of the TLB, which, on our CPU, holds entries for up to 64 KiB of memory. Afterwards, the latency diverges for different configurations due to the increased TLB-miss latency for tagged memory. On average, for working sets above 256 KiB, the latency is increased by 1.5% for 128-bit tags. We use *bw_mem* to measure memory bandwidth with a working set of 4 MiB. Table 1 shows the results of the different memory bandwidth benchmarks. Overall the impact on memory bandwidth is less than 1%. 128-bit tagging shows a slightly higher overhead since the memory interface used by the PTW can only read one word at each cycle. The write bandwidth shows the smallest reduction with 0.09% while the read bandwidth presents the highest reduction of 0.95%. The small overhead for the write bandwidth is caused by write buffering of both the CVA6 core and the AXI4 bus it is connected to.

FPGA Utilization. We evaluate the FPGA utilization of our design in two configurations. The first configuration is optimized for our enclave design, which uses 24-bit tags, out of which we only store the necessary bits in each TLB entry (cf. Figure 3). The second configuration also caches the remaining 104 "future" tag bits in every TLB entry. In anticipation of future use cases, this simulates the 'worst case' scenario in terms of hardware size. The unmodified baseline uses 68212 LUTs and 50498 Flip-Flops. Our optimized enclave design uses 68907 LUTs (+1.01%) and 50864 Flip-Flops (+0.72%). Caching all tag bits in the TLB requires 3.58% more LUTs and 5.22% more flip-flops compared to the baseline. The critical path of the core remains unchanged for both configurations. Our hardware changes in the PTW simply extend the existing state-machine with an additional state to fetch memory tags. The access checks performed on the cached tag bits in TLB entries evaluate in parallel to all preexisting checks based on permission bits and privilege levels.

Immutable Page Tables. Our design requires that the SM sets enclave-related page tables to immutable. Thus, whenever the OS writes to these pages, they trap to the SM. This happens, e.g., when mapping new pages near enclave pages. This increases the latency of write operations on such page tables. We evaluate this latency increase by allocating an enclave page and measuring the time of a `mprotect` syscall on an adjacent page that is covered by the same page table. On average, our experiment shows that the syscall takes around 21.7% longer. The overhead stems from the exception that is delivered to the SM, where the executed instruction and target memory address are evaluated. If it was a write to an immutable page table, but the write itself does not affect any enclave-related page table entries, the SM emulates the write.

Swapping. When swapping out enclave pages, the SM is required to encrypt them before swapping them out. Conversely, it decrypts them when swapping them in. To estimate the additional time this takes, we benchmark the Ascon cipher [21] and measure the time it takes to encrypt and decrypt pages. For this, we measure 260k cycles for encrypting a 4 KiB page and 267k cycles for decryption. When implementing this primitive in hardware such that it is usable

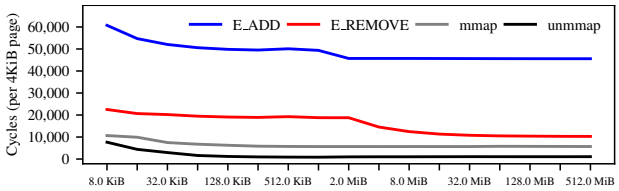

Figure 8: Runtime in cycles per page for adding and removing enclave memory compared to mmap/munmap.

as a CPU instruction [50], according to Steinegger et al., the performance increases by a factor of 50. With dedicated hardware, the time it takes to encrypt or decrypt a 4KiB memory page would only be around 8k cycles. This is even faster than the simplest system call (getpid) which takes 10k cycles on our system.

Page table scanning. When adding pages to an enclave, the SM must scan the page tables to ensure that they are not mapped twice. To measure this added overhead, we create a new enclave and request/relinquish memory from the host with different sizes. Thereby, the SM ensures that all added pages are mapped and are zero-initialized and all removed pages are also zeroed. For comparison, we also benchmark the time of `mmap` and `munmap` on the same memory areas. Figure 8 shows this overhead. We use the `MAP_POPULATE` flag to ensure all PTEs exist before adding them to the enclave. The previously mentioned `E_ADD` optimization, where we can omit scanning PTs that only contain enclave pages, has an effect starting at around 2MiB. For larger sizes `E_ADD` takes around 8.0x the time of `mmap`, while `E_REMOVE` only takes 1.8x compared to `mmap`. Even at the clock speed of our FPGA at 50MHz, the added time per page is only 0.7ms. On faster systems, or enclave workloads with a runtime of more than a few milliseconds, this overhead becomes negligible. Adding 160MiB takes around $1.8 \cdot 10^9$ cycles on our system. Due to different feature sets, architectures, and microarchitectures, we cannot directly compare this to other systems. However, for reference, Ngoc et al. measure around $5 \cdot 10^9$ cycles when starting an SGX enclave with the same size [38].

9 RELATED WORK

In addition to commercial products like Intel SGX [16], research proposals for enclaves like Keystone [30], Sanctum [15], Stockade [42], Bastion [10], Elasticlave [64], SERVAS [49], and CURE [4] exist that target application-class processors. Sancus [39], TyTan [7], and TIMBER-V [57] implement similar memory isolation only for lightweight devices. In contrast, our work targets CPUs with virtual memory without some of the restrictions of inflexible memory management like [15, 16, 30] and with low performance overheads.

Penglai [22] is a RISC-V-based enclave aiming to solve the problem of scalable memory protection. However, their proposal requires *all* page tables, including unprotected ones, to reside in a protected memory region of a fixed size. This requires non-trivial OS modifications such that any used page table is in this region. Furthermore, any write access to page tables must be verified by their monitor, which slows down all applications running on the system. SPEAR-V is able to only protect necessary page tables on a fine granularity without changing the existing memory layout. Thus, unprotected applications do not suffer from the same performance degradations when PTEs are updated.

Park et al. [41] extend SGX to support nested enclaves. However, in contrast to our work, their design adds latency to all access validations for each additional nesting level.

On SGX, previous work aimed to solve the issue of malicious enclaves by means of monitoring enclave behavior or statically analyzing the enclave code [16]. However, these methods are either not practical or cannot protect against dynamically generated code. SGXJail [56] isolates enclaves in separate sandbox processes and also proposes hardware changes based on MPK. SGXLock [12] presents a more scalable solution based on in-process sandboxing using MPK and the x86 single-step debug mode. Instead of adding sandboxing to SGX's design, in this work, we present an architecture with *mutual distrust* at its core.

Recently, most CPU manufacturers also offer solutions for confidential virtual machines. Intel's TDX [14], in contrast to our work, has completely separate page tables that are managed separately by a trusted software component. AMD's SEV-SNP [46], similar to SGX, also uses page-granular metadata. However, compared to our work, both have much larger memory overheads since they require 128 bits instead of our 24 bits. Since the CVA6 CPU does not support virtualization, we did not implement such secure virtual machines. However, we expect that SPEAR-V can be easily adapted for these use cases as well, given the necessary hardware support.

Memory tagging allows the enforcement of fine-grained policies. However, existing work typically only improves memory safety and not isolation. Previous work [17, 48, 51] used tagged memory for dynamic information flow tracking, while others [19, 24, 35, 52] allow for partially configurable security policies. ARM MTE [33] and SPARC M7 [2] offer a 4-bit tag per 16 and 64 byte, respectively, which facilitate probabilistic memory safety but not isolation.

Tagged architectures that provide isolation, such as Loki [66], Mondrian [62], or TIMBER-V [57], do so by associating every 32-bit word in memory with a 32-bit, 2-bit, and 2-bit tag, respectively. However, Loki and Mondrian provide insufficient isolation in hostile environments. TIMBER-V, while providing strict isolation, only targets embedded devices without virtual memory. Also, these schemes typically have a large memory overhead of 6% to 100%.

10 CONCLUSION

In this paper, we proposed SPEAR-V, a fast and flexible RISC-V-based architecture that provides both sandboxing and enclaving based on the same hardware primitive. Thus, providing protection against malicious hosts and malicious enclaves alike. Through minimal hardware changes, our design, which is based on page-granular memory tagging, allows partially trusted page tables that can be managed by an unmodified OS while still mitigating page table-based controlled-channel attacks. We showed how arbitrarily nested enclaves and shared memory can be efficiently implemented. Protected applications (*i.e.*, enclaves) have only a small performance overhead of 1%, while unprotected applications have no overhead. Finally, we open-source our architecture to facilitate research and evaluate future protection mechanisms based on coarse-grained memory tagging.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (grant number 888087). Additional funding was provided by a generous gift from Intel.

REFERENCES

[1] Shaizeen Aga and Satish Narayanasamy. 2019. InvisiPage: oblivious demand paging for secure enclaves. In *ISCA'19*.
[2] Aingaran et al. 2015. M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro* (2015).
[3] Alder et al. 2021. Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves. In *CCS'21*.
[4] Bahmani et al. 2021. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *USENIX'21*.
[5] Biondo et al. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *USENIX'18*.
[6] Bletsch et al. 2011. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS'11*.
[7] Brasser et al. 2015. TyTAN: tiny trust anchor for tiny devices. In *DAC'15*.
[8] Bulck et al. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX'17*.
[9] Bulck et al. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *CCS'19*.
[10] David Champagne and Ruby B. Lee. 2010. Scalable architectural support for trusted software. In *HPCA'10*.
[11] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: why the system call API is a bad untrusted RPC interface. In *ASPLOS'13*.
[12] Chen et al. 2022. SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. In *USENIX'22*.
[13] Cloosters et al. 2022. RiscyROP: Automated Return-Oriented Programming Attacks on RISC-V and ARM64. In *RAID'22*.
[14] Intel Corporation. 2020. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html
[15] Costan et al. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX'16*.
[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).
[17] Crandall et al. 2006. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.* (2006).
[18] Cui et al. 2021. SmashEx: Smashing SGX Enclaves Using Exceptions. In *CCS'21*.
[19] Dalton et al. 2007. Raksha: a flexible information flow architecture for software security. In *ISCA'07*.
[20] Dessouky et al. 2020. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX'20*.
[21] Dobraunig et al. 2021. Ascon v1.2: Lightweight Authenticated Encryption and Hashing. *J. Cryptol.* (2021).
[22] Feng et al. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *OSDI'21*.
[23] Free and Open Source Silicon Foundation. [n.d.]. Embench: Open Benchmarks for Embedded Platforms. https://github.com/embench/embench-iot/
[24] Kannan et al. 2009. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *DSN'09*.
[25] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).
[26] Kim et al. 2019. SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure. *Comput. Secur.* (2019).
[27] Kiriansky et al. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO'18*.
[28] Lebedev et al. 2019. Sanctorum: A lightweight security monitor for secure enclaves. In *DATE'19*.
[29] Lee et al. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX'17*.
[30] Lee et al. 2020. Keystone: an open framework for architecting trusted execution environments. In *EUROSYS'20*.
[31] Li et al. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX'21*.
[32] Li et al. 2021. CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In *CCS'21*.
[33] Arm Limited. 2019. Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety
[34] Liu et al. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA'16*.
[35] Liu et al. 2018. TMDFI: Tagged Memory Assisted for Fine-Grained Data-Flow Integrity Towards Embedded Systems Against Software Exploitation. In *TrustCom'18*.
[36] Larry W. McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *USENIX ATC'96*.
[37] Moghimi et al. 2017. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *IACR Cryptol. ePrint Arch.* (2017).
[38] Ngoc et al. 2019. Everything You Should Know About Intel SGX Performance on Virtualized Systems. *Proc. ACM Meas. Anal. Comput. Syst.* (2019).
[39] Noorman et al. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.* (2017).
[40] Meni Orenbach, Andrew Baumann, and Mark Silberstein. 2020. Autarky: Closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[41] Park et al. 2020. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *ISCA'20*.
[42] Park et al. 2021. Stockade: Hardware Hardening for Distributed Trusted Sandboxes. *CoRR* (2021).
[43] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *MICRO'18*.
[44] Gururaj Saileshwar and Moinuddin K. Qureshi. 2021. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *USENIX'21*.
[45] Schwarz et al. 2019. Practical Enclave Malware with Intel SGX. In *DIMVA'19*.
[46] AMD SEV-SNP. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020).
[47] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS'07*.
[48] Song et al. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *S&P'16*.
[49] Steinegger et al. 2021. SERVAS! Secure Enclaves via RISC-V Authenticryption Shield. In *ESORICS'21*.
[50] Stefan Steinegger and Robert Primas. 2020. A Fast and Compact RISC-V Accelerator for Ascon and Friends. In *CARDIS'20*.
[51] Suh et al. 2004. Secure program execution via dynamic information flow tracking. In *ASPLOS'04*.
[52] Venkataramani et al. 2008. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *HPCA'08*.
[53] Wang et al. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS'17*.
[54] Ruisheng Wang and Lizhong Chen. 2014. Futility Scaling: High-Associativity Cache Partitioning. In *MICRO'14*.
[55] Weichbrodt et al. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS'16*.
[56] Weiser et al. 2019. SGXJail: Defeating Enclave Malware via Confinement. In *RAID'19*.
[57] Weiser et al. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *NDSS'19*.
[58] Werner et al. 2017. Transparent memory encryption and authentication. In *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*.
[59] Werner et al. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX'19*.
[60] Wistoff et al. 2021. Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core. In *DATE'21*.
[61] Wistoff et al. 2022. Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning. *IEEE Trans. Comput.* (2022).
[62] Witchel et al. 2002. Mondrian memory protection. In *ASPLOS'02*.
[63] Xu et al. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P'15*.
[64] Yu et al. 2022. Elasticlave: An Efficient Memory Model for Enclaves. In *USENIX'22*.
[65] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Trans. Very Large Scale Integr. Syst.* (2019).
[66] Zeldovich et al. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *OSDI'08*.

Received 15 December 2022; revised 21 April 2023; accepted 26 April 2023