SERVAS! Secure Enclaves via RISC-V Authenticryption Shield

Stefan Steinegger[1(✉)], David Schrammel[1], Samuel Weiser[1], Pascal Nasahl[1], and Stefan Mangard[1,2]

[1] Graz University of Technology, Graz, Austria
{stefan.steinegger,david.schrammel,samuel.weiser, pascal.nasahl,stefan.mangard}@iaik.tugraz.at
[2] Lamarr Security Research, Graz, Austria

Abstract. Isolation is a long-standing security challenge. Privilege rings and virtual memory are increasingly augmented with capabilities, protection keys, and powerful enclaves. Moreover, we are facing an increased need for physical protection, e.g., via transparent memory encryption, resulting in a complex interplay of various security mechanisms. In this work, we tackle the isolation challenge with a new extensible isolation primitive called authenticryption shield that unifies various isolation policies. By using authenticated memory encryption, we streamline the security reasoning towards cryptographic guarantees. We showcase the versatility of our approach by designing and prototyping SERVAS – a novel enclave architecture for RISC-V. SERVAS facilitates a new efficient and secure enclave memory sharing mechanism. While the memory encryption constitutes the main overhead, invoking SERVAS enclave requires only 3.5x of a simple syscall instead of 71x for Intel SGX.

1 Introduction

Today, software vulnerabilities are omnipresent and penetrate the whole software stack, e.g., application software [52] and operating systems [12,27]. To reduce their impact, different isolation mechanisms can separate privileges [19], isolate individual processes [35], protect virtual machines [5,6], and segregate applications into smaller parts, also denoted as in-process isolation. Typical in-process isolation mechanisms are segmentation [31] and capabilities [61], memory coloring (e.g., protection keys) [30,49,55], or enclaves. Enclaves give strong security guarantees even in the event of a system compromise and found ample resonance both in academia [11,15,18,39,53,57] and industry [9,24,43]. In addition, cloud computing increasingly demands physical protection, for which modern CPUs provide transparent memory encryption [6,43]. Having proper integrity protection can cause worst-case throughput penalties of over 400% for Intel SGX [28].

Unfortunately, due to the zoo of isolation mechanisms, reasoning about their security becomes increasingly complex. For example, an application might depend on protection keys in combination with the Memory Management Unit

© Springer Nature Switzerland AG 2021
E. Bertino et al. (Eds.): ESORICS 2021, LNCS 12973, pp. 370–391, 2021.
https://doi.org/10.1007/978-3-030-88428-4_19

(MMU) and the memory mappings configured by the operating system [35]. Unifying these isolation mechanisms is desirable from a security standpoint, but most cover only a subset of scenarios. For example, Intel SGX isolates unprivileged user code, but its memory encryption is not utilizable for other purposes.

In this work, we first simplify the overall security reasoning by introducing a strong and generic isolation primitive. Second, we explore our primitive's synergies and features. Third, we use it to design a novel enclave architecture.

New Isolation Primitive. We unify various isolation policies with our novel RISC-V Authenticryption Shield (RVAS). By using memory encryption, we map isolation properties to the well-studied field of cryptography. More specifically, encryption ensures that the CPU and the memory are in a particular state. *Thus, RVAS achieves memory isolation with cryptographic guarantees.*

We design RVAS atop the RISC-V architecture. RVAS builds upon authenticated memory encryption whose associated data input, which we call encryption tweak, is exposed to software. This encryption tweak represents a security context composed of software-defined and CPU-internal components. It serves to achieve domain separation and can enforce a variety of different isolation mechanisms simultaneously, e.g., privilege separation, virtual memory protection, segmentation, and page coloring, etc. Traditionally, each of these mechanisms needs to securely store trusted metadata (e.g., the address mapping or the page colors). *RVAS implicitly secures this metadata by feeding it into the encryption tweak.* A proper generalization of encryption tweaks is non-trivial.

SERVAS Enclaves are our novel enclave system atop RVAS. SERVAS protects enclaves against software and physical attacks by means of RVAS encryption. In contrast, Intel SGX uses memory encryption only against *physical* attacks, while *software* attacks are prevented through a trusted metadata storage – the so-called EPCM [31]. Our design makes the EPCM obsolete, which yields two advantages: First, we remove trust from the address translation, *i.e.*, the MMU and the Translation Lookaside Buffer (TLB) configuration, and our security argument boils down to encryption tweaks. Second, SGX enclaves can typically only use 128 MB of encrypted physical memory [26]. RVAS encryption can be applied to the whole DRAM and also to non-enclave code.

SERVAS introduces the novel concept of secure sharing of enclave memory, a key requirement for many application scenarios but impractical with current enclave systems (e.g., requiring costly software-based encryption). SERVAS enables secure zero-cost memory sharing by sharing encryption tweaks.

We prototype RVAS on an FPGA using an open-source encryption core. A small stateless Security Monitor (SM) running in RISC-V machine mode ensures a proper tweak configuration. Invoking SERVAS enclaves only takes 3.5x the time of a syscall. Our evaluation indicates an overhead of 16.7% to 24.5% over the used encryption core. An extended version of this paper is available [51], and we plan to open-source our prototype[1]. In summary, our contributions are:

[1] https://github.com/IAIK/servas.

- A generic isolation primitive using authenticated memory encryption.
- A novel enclave architecture called SERVAS.
- A novel and fast and secure memory sharing mechanism between enclaves.
- An evaluation of the prototype implementation of SERVAS.

2 Challenges of Memory Isolation

Here, we give an overview of the most widely used isolation schemes and present their challenges concerning security and functional limitations we want to overcome. This paves the way for understanding the RVAS and SERVAS design.

Process Isolation requires privilege separation between the operating system (OS) and processes and isolating processes from each other. Privilege separation is achieved via privilege rings protecting CPU resources from unprivileged access. The OS also needs to configure the virtual memory subsystem:
Challenge $\boldsymbol{\mathcal{C}}_1$*: "The privileged software must ensure that the virtual memory mappings of* ***all*** *unprivileged processes (i) cannot access privileged memory, and (ii) are not unintentionally aliasing with each other."*

Segmentation is a fine-grained in-process isolation mechanism using address ranges. Segmentation forms the basis of hardware capabilities [61]. However, these systems are not suitable for enforcing cross-application policies, e.g., protecting cryptographic keys from other applications or the OS.
Challenge $\boldsymbol{\mathcal{C}}_2$*: "Segmentation should allow flexible cross-application policies."*

Memory Coloring is another in-process isolation mechanism labeling each memory block with a "color". If the color is loaded, the memory becomes accessible. Unfortunately, the number of colors is often quite limited [47] and not suitable to enforce cross-application policies, e.g., for shared memory.
Challenge $\boldsymbol{\mathcal{C}}_3$*: "Memory coloring should provide significantly more colors and also allow cross-application policies."*

Memory Mapping Protection is needed to protect enclaves from the OS:
Challenge $\boldsymbol{\mathcal{C}}_4$*: "The memory mapping of enclaves must be protected against privileged software."* This is challenging because privileged software is in charge of managing enclaves. E.g., manipulating the page tables could cause data pages to become executable. Three security invariants need to hold here:

- **Attribute Invariant** $\boldsymbol{\mathcal{IA}}$: *"Enclave pages must only be mapped with their intended page table attributes."*
- **Spatial Invariant** $\boldsymbol{\mathcal{IS}}$: *"A physical enclave page must only be mapped to its corresponding virtual page."*
- **Temporal Invariant** $\boldsymbol{\mathcal{IT}}$: *"At any time for every virtual enclave page, there must be at most one valid physical page mapping."*

$\boldsymbol{\mathcal{IT}}$ specifically addresses double mapping attacks, where the attacker could silently replace the page to replay old data.

Protected Sharing is typically achieved via shared memory. However, the hard isolation boundary of enclaves prohibits secure, shared memory by design: *Challenge* $\mathcal{C}_5$*: "Shared memory must allow for efficient and confidential interaction between different enclaves."*

Memory Encryption. The DRAM can be attacked via passive [8,29] and active [36] physical attacks. Encrypted and authenticated DRAM is necessary to protect data from physical attacks. Memory encryption should not be restricted to specific code (e.g., enclaves) or specific parts of the DRAM.
Challenge $\mathcal{C}_6$*: "The DRAM shall be hardened against active and passive physical attacks."*

3 RISC-V Authenticryption Shield (RVAS)

RVAS presents a generic mechanism to cryptographically enforce the challenges expressed in Sect. 2. At its core, it uses an authenticated Memory Encryption Engine (MEE) for encrypting the DRAM and incorporates a security context into its associated data. If encrypted data is accessed with the wrong security context, the MEE triggers an authentication error. Since the MEE gives cryptographic security guarantees for detecting authentication issues, the security argumentation boils down to one question: *Who controls the security context?*

The composition of the security context arguably lies at the heart of RVAS. For readability, we also call it "tweak" in the rest of the paper. The tweak consists of both software- and CPU-defined components, allowing for fine-grained, unforgeable isolation. In this section, we discuss the tweak composition, how RVAS solves the challenges from Sect. 2, and the requirements for the MEE.

3.1 RVAS Tweak Design

Our tweak design comprises five components explained in the following, each of which can be selectively enabled, depending on the specific use case.

Integrity Counter. The MEE maintains integrity counters for each memory block, which it increments at each write operation. Integrating the counter into the tweak ensures that the correct memory block is used at any time and thus, prevents reverting the memory to a previous state (*i.e.*, replay attacks).

Segmentation and Address Information holds metadata about the accessed address and whether it matches software-defined segments that can be configured at each privilege level. This allows to protect the page mapping, in particular the address translation, and page ordering and prevents double mappings. The address information can hold an absolute address or an offset relative to one of the segments. Each segment has a base address and a size and belongs to a privilege level. Depending on which segment(s) the address belongs to, the segment bitmap, which is also included in the tweak, is set. This further acts as a domain separation and influences which memory color is used.

Privilege Level. Including the CPU privilege level (e.g., M-mode, S-mode, U-mode) in the tweak ensures that memory is only accessible at a specific level.

Page Table Attributes such as read, write and execute permissions are included in the tweak to prevent undetected altering of the page mapping.

Memory Color. This field is extremely versatile and can be configured by software on each privilege level. By choosing appropriate colors, one can segregate memory pages at runtime and facilitate sharing across security domains.

3.2 Solving the Challenges

Process Isolation with RVAS could significantly enhance the security of processes, e.g., inside encrypted virtual machines [6]. Two components solve Challenge $\mathcal{C}_1$: First, we use the CPU privilege level in the tweak to achieve privilege separation, *i.e.*, *without the need for inspecting page tables*. Second, we use an OS-chosen process identifier in the tweak's memory color field to separate processes. By using RISC-V's Supervisor User Memory (SUM) bit to temporarily force the privilege level to U-mode in the tweak, the OS can be granted temporary access to user memory (e.g., for syscall handling). Of course for an enclave system one would disable this feature.

In-process Isolation. To solve the segmentation challenge $\mathcal{C}_2$ for cross-privilege policies, we can supply information from all privilege levels to the tweak's *Segment and Address Information* field. A segment-relative address offset in the tweak makes these policies compatible between different applications (*i.e.*, different address spaces), as we will show for cross-enclave shared memory.

To solve the memory coloring challenge $\mathcal{C}_3$, we support a vast number of 2^{80} colors (cf. 16 for Intel MPK [55]). Thus, RVAS makes trusted metadata storages for memory colors (*i.e.*, tagged memory) obsolete [64]. We will show how RVAS achieves brute-force resistance when using colors as a shared secret.

Enclaves are the most complex isolation technique discussed in this paper and involve the challenges $\mathcal{C}_2 - \mathcal{C}_6$. Current enclave systems like Intel SGX [43] use trusted metadata stores, *i.e.*, the Enclave Page Cache Map (EPCM), to ensure the attribute- $\mathcal{IA}$, the spatial- $\mathcal{IS}$, and the temporal invariant $\mathcal{IT}$. However, the EPCM has a few drawbacks: (1) It increases the Trusted Computing Base (TCB). (2) It takes up memory. (3) The enclave's TLB entries must be flushed during context switches [17,31]. (4) It permits only a single owner enclave for each page, precluding flexible enclave memory sharing.

We leverage RVAS to solve challenge $\mathcal{C}_4$ and make the EPCM obsolete: First, we guarantee $\mathcal{IA}$ by feeding the *page table attributes* into the tweak. Second, we enforce $\mathcal{IS}$ and $\mathcal{IT}$ by using the *segmentation and address information* and the *memory color* field in the tweak. Thus, that pages can only be mapped correctly to their legitimate enclave. More details are given in Sect. 4.3.

To solve the protected sharing challenge $\mathcal{C}_5$, we combine the memory color field ($\mathcal{C}_3$) with an enclave-defined segment ($\mathcal{C}_2$) for specifying the shared memory.

The memory color essentially comprises a shared secret established between two or more enclaves that want to communicate. The relative addressing of segments allows the enclave to choose the exact location of the shared memory. Only if the memory color and the segment is set up correctly, the enclaves will have the same encryption tweak and, thus, can access the shared memory.

Memory Encryption protects against active and passive physical attacks, thus solving challenge $\mathcal{C}_6$. For RVAS, an MEE needs to fulfill three properties: (1) confidentiality, authenticity, and integrity of the data, (2) replay protection, (3) the used cryptographic primitive must be tweakable. Integrity is typically ensured by storing authentication codes in a tree structure. The replay protection from (2) is usually done via authenticated counters that are typically fed into the encryption scheme as a tweak or nonce [17,23,54,59,60]. To fulfill (3), we require a tweakable block cipher or authenticated encryption scheme with a sufficiently large tweak size (*i.e.*, associated data), such as [21,62]. E.g., SGX's underlying MEE would require changes to fulfill the third property.

4 SERVAS

SERVAS is an innovative enclave architecture and, thus, the most complex RVAS use case we present. As shown in Fig. 1, SERVAS consists of the RISC-V Authenticryption Shield (RVAS) and a software Security Monitor (SM). The SM is the trusted intermediary that handles the enclave's lifecycle, acting as a universal entry and exit point and manages the RVAS encryption tweak via an Instruction Set Architecture (ISA) extension.

SERVAS follows SGX's design choices to keep a minimal TCB while simultaneously avoiding the drawbacks of large trusted metadata storages (*i.e.*, the EPCM). Instead, we feed the relevant security metadata into the RVAS tweak. By carefully controlling the tweak, SERVAS maintains cryptographic segregation of various security domains. SERVAS also enables dynamic enclave memory and secure sharing of enclave memory, avoiding costly software-based encryption [7].

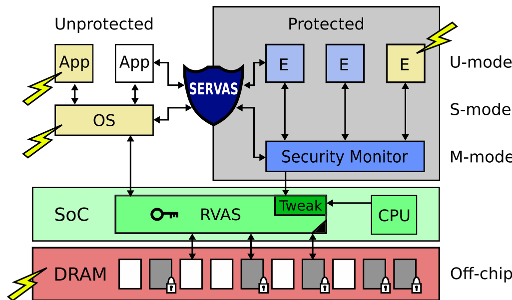


Fig. 1. SERVAS protects enclaves (*E*) from applications (*App*), the OS, and physical attacks. Thunderbolts mark the attack surface. RVAS encrypts and authenticates pages in the untrusted DRAM.

4.1 Threat Model

SERVAS protects code and data inside enclaves against a powerful, privileged software and physical attacker (cf. Intel SGX [17,43]). Non-enclave software (*i.e.*, the OS or user applications) is viewed as untrusted and can be attacker controlled. The OS can launch rogue enclaves. Our Trusted Computing Base (TCB) comprises software and hardware components: On the software side, we trust the enclaves and a small Security Monitor (SM). The enclave developer is responsible for avoiding vulnerabilities in the enclave code [13,40]. The SM is an integral part of our CPU hardware, similar to Intel SGX's microcode implementation [17]. It can be protected via a trusted on-chip ROM or a secure boot mechanism [38].

On the hardware side, anything outside the System on Chip (SoC) (e.g., the CPU and RVAS) is untrusted. In particular, the attacker can tamper with the DRAM, mount bus probing, cold-boot [29], or fault attacks on the encrypted DRAM [33], which are detected by RVAS' authenticated encryption.

Denial-of-service attacks are outside of our threat model. It is up to the OS and the applications to invoke an enclave. Whether performed in software or hardware, side-channel attacks are an orthogonal challenge, for which plenty of literature is available that could also be applied to SERVAS [15,16,20,54,58].

4.2 Enclave Life Cycle

SERVAS enclaves are built on top of RVAS and our Security Monitor (SM). The SMprovides an API for managing all phases of an enclave's life cycle, namely, initialization, entering, interruption, exiting, attestation and sealing.

Enclaves are identified via their so-called Enclave Identifier (EncID), which is computed via a cryptographic authentication code over the enclave binary (cf. SGX's MRENCLAVE [17]). Enclave binaries are encrypted with Ascon and a per CPU key. Similarly to SecureBlue++ [14] this implicitly attestates the enclave and allows for embedding secrets without the need for remote attestation and provisioning. During loading, the SMcompares the cryptographic authentication code. If the enclave binary was corrupted, the SMwill yield a different EncID and the SMwill refuse to execute the enclave.

User-mode applications can load and run enclaves within their own virtual address space, for which the SMassigns a unique Runtime Identifier (RTID) to each running enclave. For entering an enclave, the SMconfigures its tweak and transfers control to its single developer-specified entry point. An interrupt causes the SMto save the enclave's register state on private enclave memory and wipe the registers. Resuming from interruption and exiting an enclave is analogous. To store enclave secrets, the SMprovides a sealing functionality based on a key-derivation function involving the EncID and a per-CPU key. For more details about the enclave life cycle, we refer to our extended paper version [51].

4.3 Enclave Memory Management

Security of the enclave memory hinges on the *spatial* ($\mathcal{IS}$), *temporal* ($\mathcal{IT}$), and *attribute invariant* ($\mathcal{IA}$), as specified in Sect. 2, which need to hold over the whole enclave's lifecycle. To achieve these invariants, our trusted SMexclusively controls parts of the RVAS tweak. In particular, the initialization of enclave memory can only be done by the SM, for which it can override most parts of the RVAS tweak as if the enclave itself initialized the memory. Similarly, the SMcan invalidate an enclave page by writing it under a different tweak.

In the following, we show how to secure static and dynamic enclave page mappings and achieve shared memory and swapping.

Static Page Mapping and Code Sharing. Our three invariants protect static enclave pages, *i.e.*, code and data sections, from OS compromise. $\mathcal{IA}$ is ensured by adding page attributes to the tweak. To keep $\mathcal{IS}$, we link between virtual and physical enclave pages, as follows: An SM-controlled address segment distinguishes enclave memory from the rest. Any enclave memory access includes the segment-relative virtual address offset in the tweak. This field guarantees a correct address translation and also accounts for position-independent enclaves.

To ensure $\mathcal{IT}$ for private enclave memory, we put the enclave's unique Runtime Identifier (RTID) into the tweak's memory color field. This field binds each enclave page to exactly one enclave instance. Developers can optionally deduplicate enclave code to help reduce memory load and TLB pressure. In this case, we use the Enclave Identifier (EncID) instead of the Runtime Identifier (RTID) inside the tweak. To enforce $\mathcal{IT}$, these code pages need to be read-only and shared between enclave instances of the same binary only (*i.e.*, the same EncID).

Dynamic Page Mapping. SERVAS enclaves may use dynamic memory, which has been allocated by the host user-mode application. In principle, our security invariants are upheld in the same way as for static page mappings. However, the invariant $\mathcal{IT}$ requires special care to prevent double mapping attacks since dynamic mappings change during runtime: The enclave (runtime) keeps track of all of its valid page mappings *inside* the enclave's address range, e.g., in a private bitmap similar to SGX [42]. Thus, when the enclave receives new memory from the host, it can verify the mapping. Therefore, the enclave effectively acknowledges each dynamic memory page before it is initialized by the SMto be used. If an enclave releases dynamic memory, it explicitly invokes the SM, which invalidates the page content, e.g., by destroying its integrity. This prevents use-after-free scenarios and upholds $\mathcal{IT}$.

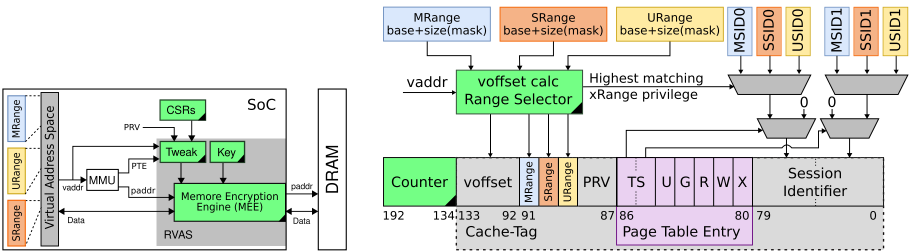
Data Sharing. SERVAS introduces a novel concept of data sharing between enclaves. Shared data memory is writable and can be used for data exchange at native speed (*i.e.*, without copying or re-encryption [7]). This memory is realized via a shared secret, supplied as memory color, that enclaves can directly

manage. However, the SMcan assist in establishing the secret between enclaves, e.g., as a trusted entity attesting the enclaves. Upholding our security invariants for data sharing is critical and highlights the versatility of our RVAS design. Enclaves can enforce the invariants by simply configuring the shared memory and keeping the shared secret confidential. Data sharing also seamlessly scales to multiple enclaves. A user-mode segment register points to the desired shared virtual memory range to prevent double mapping attacks (e.g., aliasing shared- with private pages) and upholding $\mathcal{IT}$.

Swapping. In contrast to Intel SGX, SERVAS allows using the full DRAM for enclaves, nevertheless, out-of-memory situations can occur. To ensure the SM's correct operation and maintain our security invariants, we exclude enclave shared memory and SM-related pages from being swapped. In order to swap an enclave page, the OS provides a temporary page to the SM. The SMre-encrypts the page-to-swap to the temporary page. Next, the SMinvalidates the original physical enclave page in order to uphold $\mathcal{IT}$. The involved metadata (authentication tag, nonce, virtual address, range information, page permissions) are saved on a page only accessible to the SM. This metadata pins the page's version and prevents any roll-back attacks. Swap-in of enclave pages is analogous.

5 SERVAS Implementation Details

In this section, we detail our SERVAS implementation, shown in Fig. 2a. We discuss our instruction set changes with the tweak construction and page types and caching considerations and an encryption bypass option.



(a) Blocks with black corners show components that are part of RVAS.

(b) SERVAS tweak composition, including replay-protection counters, tweak select (TS), privilege mode (PRV), etc.

Fig. 2. Overview of RVAS and the tweak construction for the MEE.

5.1 Instruction Set Extension

RVAS adds minimal changes to the RISC-V Instruction Set Architecture (ISA): We add Control and Status Registers (CSRs) to set RVAS' tweak in software. Moreover, we add an authentication exception that is raised whenever decryption fails with an integrity check error during a read, write, or fetch operation.

We add the segmentation registers `MRange`, `SRange`, and `URange` for the **m**achine-, **s**upervisor-, and **u**ser-mode, respectively, as seen in Fig. 2b. Each segment is defined by a base and a size in the virtual address space. The SMuses `MRange` to declare an enclave's memory. `SRange` and `URange` can be used for different purposes (e.g., shared memory). To include software-controllable Session Identifiers (SIDs) for page coloring, we add `xSID0` and `xSID1` CSRs.

Tweak Override. We provide additional tweak override registers that allow the SMto cryptographically initialize a page without trusting the OS-supplied page mapping. These registers can override any tweak parameters used by RVAS, except for the RVAS-managed integrity counters. Thus, any memory accesses by lesser-privileged modes must adhere to the same tweak used for initialization.

5.2 Tweak

RVAS incorporates CPU state information into the MEE via the tweak, including integrity counters, page mapping, and privilege information, range checks, and SIDs (cf. Fig. 2b). SERVAS uses a tweak size of 192 bits, as follows:

Counter. Similar to SGX's 56-bit counters, we reserve 58 tweak bits for *integrity counters* to protect against replay attacks, which are not exposed to software.

xRange. We use a bitmap with three bits to encode whether the accessed address is within `URange`, `SRange`, or `MRange`, respectively. If an address matches `xRange` segments, their tweak bits are set. The bits act as a domain separation and influence the SID and `voffset` selection. The most unprivileged matching `xRange` (e.g., `URange`) then determines the `voffset` calculation and the choice of the `xSID` registers.

voffset is the virtual address offset computed from the trusted base address of the effective matching `xRange` register at cache line granularity. Intel SGX protects the page mapping by linking the virtual- and physical address in the EPCM. For SERVAS enclaves, the page mapping is protected with the `voffset`, which is relative to the base of a segment. Additionally, the segment is uniquely identified with a SID. Combined, this ensures that the virtual mapping is correct, *i.e.*, as initialized by the SM. For 48-bit virtual addresses [56] and 64 B cache lines [17], this results in 42 bit.

PRV encodes the current privilege level of the CPU in two bits and ensures the memory can only be accessed at a specific level.

Page Table Entry (PTE). We include seven page table attribute bits from the page table entry (PTE) in the tweak to ensure they represent their initialized configuration. These attributes cover the user mode (U), global mapping (G), read, write and execute privileges (RWX), and software-defined reserved tweak select (TS) for selecting the effective SID register.

Session Identifier (SID). We allocate 80 bit for SIDs useable for defining memory colors and ensure a certain execution context. The active `xRange` determines whether `MSID`, `SSID`, or `USID` is used. TS determines whether one or both `xSID0` and `xSID1` registers are used. When using, both the effective SID is truncated to 80 bit.

5.3 Page Types

SERVAS defines five page types via a specific tweak combination (cf. Table 1):

PT_NORMAL marks untrusted memory outside of all `xRange` segments. It adheres to the PTE and can optionally be encrypted (cf. our encryption bypass).

PT_ENCLAVE denotes private enclave pages within the `MRange`, having any suitable combination of PTE page permissions. The TS bits specify the use of the MSID0 register, holding the unique SM-defined RTID identifier.

PT_SHCODE shares non-writable pages between different instances of the same enclave to reduce memory and TLB pressure. This type adheres to the `MRange` and uses the MSID1 CSR, which holds the unique EncID of the enclave.

PT_SHDATA shares non-executable data between enclaves and resides in the enclave-configured `URange`. The *virtual address offset* is calculated relative to `URange` and ensures cross-enclave accessibility. The TS bits select the USID0 and USID1 CSRs, into which enclaves load an 80-bit shared secret before accessing the shared memory. The SMhelps in establishing the shared secret.

PT_MONITOR stores dynamic metadata for each enclave and thread (cf. SECS and TCS [17]) and is only accessible by the SMin M-mode. Note, that enclave code is not protected via PT_MONITOR but instead via Physical Memory Protection (PMP).

5.4 Security Monitor (SM)

The SMruns in machine mode (which is loosely comparable to CPU microcode used for Intel SGX logic) and protects itself using the RISC-V PMP. Our prototype has a tiny code size of 1253 Lines of Code (LoC), of which 381 LoC are used by the ASCON implementation (which is used for computing, e.g., the EncID and the sealing key). Therefore, it is small enough to be stored in on-chip SRAM, without the need for encryption.

In principle, the SMcan run completely stateless and only requires a small (approx. 1KiB) stack. Enclave management data is stored in dynamically allocated PT_MONITOR pages. The SMmaintains a 64 bit monotonic counter to allocate a unique RTID per enclave. One could also sample the RTID from the RISC-V hardware performance counters, e.g., the elapsed CPU cycles `mcycle`.

5.5 Caching

In our so-called *inline variant*, we cache the RVAS tweaks in the data and instruction caches. This caching allows the match of the currently active tweak with the cache line and ensures that the entire tweak can be reconstructed for any write-back operations. We store $b_{SERVAS} = 134$ tweak bits in each of the N_{cache} cache lines, covering the `xRange`-, PTE-, PRV-, SID-, and voffset bits. Since the MEE manages the integrity counters, they need no caching. This totals $S_{Data}, S_{Instr} = b_{SERVAS} \cdot N_{cache}$ additional cache bits.

Cache Optimization. For real-world scenarios, tweaks can be deduplicated into a separate Tweak Cache (TC) [34] to shrink the caching overhead in the *main caches*. The TC is linked with the main caches via a tweak index of $b_{tweakidx}$ bits. We propose a set-associative TC whose set index is derived from the tweak via a pseudorandom non-linear function, e.g., a lightweight cryptographic primitive [50,58]. This allows for efficient identification of already inserted tweaks, and only one additional cycle for the tweak comparison may be required. If a TC entry is removed, all associated cache lines need to be flushed.

Table 1. Tweak decision table: • denotes arbitrary values.

`MRange`	`SRange`	`URange`	PRV	PTE	TS	SID	Label
0	0	0	•	•	•	•	PT_NORMAL
1	0	0	U	•	01	MSID0	PT_ENCLAVE
1	0	0	U	!W	10	MSID1	PT_SHCODE
0	0	1	U	!X	11	USID0+1	PT_SHDATA
•	•	•	M	rw	•	0	PT_MONITOR

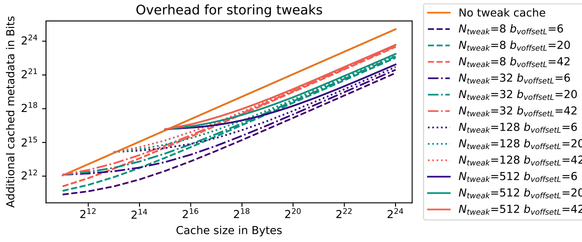


Fig. 3. Tweak cache compared to the inline variant (512*bit* cache lines) for different TC configurations (N_{tweak}, $b_{voffsetL}$). Lower is better.

The exact TC parameterization depends on the expected workload (*i.e.*, the number of tweaks required in parallel). As another optimization, the $b_{voffset}$ can be split between main caches ($b_{voffsetL}$) and the TC ($b_{voffsetH}$), resulting in a number of *tweak zones* per enclave. Moreover, we must consider two constraints: (1) more ways in the TC require additional parallel comparator logic, and (2) the overall cached tweak bits must be less than for the inline variant. To handle (1) there should be no more ways in the TC as in the main cache. Addressing (2) depends on the size of tweak zones, the size of the main caches, and the TC's size. The number of entries N_{tweak} in the TC determines the width of the index $b_{tweakidx}$. We also include a valid bit b_{valid}:

$$S_{DataOpt}, S_{InstrOpt} = (b_{voffsetL} + b_{tweakidx}) \cdot N_{cache} \quad (1)$$

$$S_{tweakcache} = (b_{valid} + b_{SERVAS} - b_{voffsetL}) \cdot N_{tweak} \quad (2)$$

We evaluate the storage overhead as a function of N_{cache} with 512 bit cache lines for different N_{tweak} and $b_{voffsetL}$ in Fig. 3. We observe that each N_{tweak} has the same break-even point for all $b_{voffsetL}$. After that, smaller $b_{voffsetL}$ reduce the overhead more. With larger caches $b_{voffsetL}$, becomes the dominating factor and clusters them into groups. This effect is less pronounced for $b_{tweakidx}$ due to its smaller size. However, as an example, a TC with $N_{tweak} = 128$ entries could reduce the storage overhead from 137 216 bit to 42 368 bit by about 69% for the CVA6 CPU with its 64 kB main cache and 1 GB tweak zones ($b_{voffsetL} = 20$).

5.6 Encryption Bypass Optimization

Our prototype encrypts the whole system's physical memory. To improve performance, one could also apply RVAS encryption only to pages (e.g., enclaves) that require this protection. This can be achieved by using the `xRange` registers to decide if a request has to go through the MEE or access the memory directly.

A limitation of the encryption bypass is that the inherent overhead of the integrity protection trees introduced by the MEE persists. Making the trees sparse could address this problem, but no such open-source memory encryption schemes have been proposed to the best of our knowledge.

6 Security Analysis

This section analyzes how RVAS's tweak (cf. Sect. 2) cryptographically enforces the challenges of memory isolation $\mathcal{C}_1$-$\mathcal{C}_6$. Attacks aim at breaking the *attribute-* ($\mathcal{IA}$), *spatial-* ($\mathcal{IS}$), or *temporal invariant* ($\mathcal{IT}$).

6.1 Attacks on Physical Memory

An OS or physical attacker can access the physical memory via software or by mounting bus probing or cold-boot attacks [29]. However, RVAS's encryption prevents data access. Performing roll-back attacks or move encrypted data around to violate the invariant $\mathcal{IT}$ is mitigated by RVAS' integrity counters.

6.2 Attacks on Virtual Memory

Memory Isolation. Enclaves run in the virtual memory of a host application. Hence a rogue enclave, host, or OS could try to access enclave data. SERVAS supplies unforgeable (e.g., `MRange`, MSID) or enclave-private data to the memory color field of RVAS. Without a correct tweak, RVAS fails and traps to the SM.

Page Mapping Attacks. The OS has full control over the page table entries (PTEs) (challenge $\mathcal{C}_4$, cf. Sect. 2), allowing for a range of attacks:

Downgrade or Remapping Attacks. The OS can map an unprotected page or another enclave's page to an address in the enclave's `MRange` to leak data or divert the control flow and violate $\mathcal{IA}$, $\mathcal{IT}$ or $\mathcal{IS}$. However, the combination of SM-controlled `MRange` registers, PTE bits, and the session identifier (e.g., the Runtime Identifier (RTID) or the Enclave Identifier (EncID)) ensure that any rogue pages in the `MRange` fail RVAS' integrity check.

Swapping Attacks. The OS can create a copy of an enclave page via the swapping mechanism to attempt to violate the temporal invariant $\mathcal{IT}$. SERVAS mitigates this attack by invalidating the original page before the swapped-out copy is given to the OS. A similar attack attempts a roll-back by swapping-out a page twice and providing the older copy during swap-in. This is prevented by the latest authentication tag of the swapped out copy on a PT_MONITOR page.

Shared Data Page Attacks. Enclave shared memory faces two attacks from the OS: (1) Replace an enclave page with a shared memory page to trick the enclave into leaking its secrets. (2) The OS and malicious enclave cooperate to brute-force the 80 bit key. To prevent (1), the enclave must explicitly set the `URange` CSR and the key before shared memory is active. Scenario (2) is an online attack on shared memory only. It can be made practically infeasible by (A) terminating the attacker enclave, as loading the enclave acts as dynamic rate-limiting, or (B) performing explicit rate-limiting in the SMexception handler.

Shared Code Page Attack. To deduplicate non-writable pages between different instances of the same enclave, the Enclave Identifier (EncID) is used as the memory color. An attacker can run an offline attack to create an enclave with a colliding EncID, which refers to finding a second pre-image to a cryptographic authentication code. Using a 128 bit EncID could completely eliminate this attack. However, SERVAS only supports an 80 bit SID. To achieve practical security, the EncID could be truncated using a key derivation function that involves a secret CPU key.

7 Evaluation

Our prototype is based on the CVA6 [63] platform, a 64-bit RISC-V CPU. For SERVAS, we extended this platform with the RVAS ISA extensions, the storage of tweaks in its write-through cache tag, and an MEE for RVAS. We increased the default cache line size from 16 B to 64 B, a common choice for many CPUs. We use MEMSEC [59], an open-source framework supporting various encryption schemes for the MEE. To fulfill our requirements (cf. Sect. 3), we configured MEMSEC to use ASCON-128 and extend it to process the tweak as Ascon's associated data. We use ASCON-128 for RVAS as it is the only cipher that is supported by the MEMSEC framework in TEC-Tree mode. MEMSEC is placed between the cache and the memory controller to encrypt all data transparently.

7.1 Performance Overhead

We ran a set of macrobenchmarks on a Linux 5.10 kernel on the CVA6 CPU to evaluate the performance. We use BEEBS [46] and CoreMark [22] as benchmarks. We excluded the `crc32`, `ludcmp`, `st`, `matmult-float`, and `rijndael` benchmarks from BEEBS, since they caused lockups on the unmodified CPU. We use the geometric mean to aggregate all BEEBS benchmarks into a single metric in Fig. 4. Detailed results are given in Appendix A. For the fast-running CoreMark, we plot the mean over 1 000 runs (with 10 internal iterations), while for the slower BEEBS, we average over 25 runs (with 4 internal iterations). We cannot run more heavyweight benchmarks due to the prototype's resource constraints (256 MB accessible DRAM and 50 MHz CPU frequency).

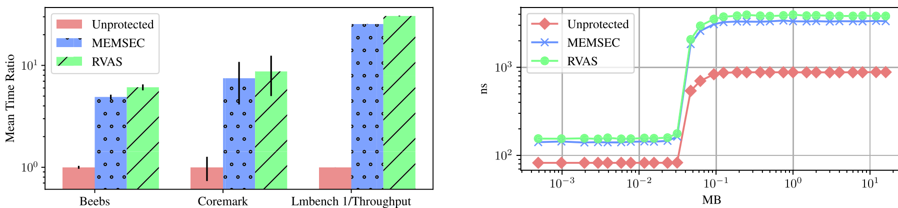


Fig. 4. Left: Benchmarking the CVA6 core using no memory encryption, MEMSEC, or RVAS. Right: Memory read-write latency.

Table 2. Micro-benchmarking results for SERVAS.

	Cycles median	Relative to getpid
Syscall getpid	10 403	1.0x
SERVAS SM Call “null”	9 030	0.9x
SERVAS Enter	18 865	1.8x
SERVAS Exit	17 391	1.7x
SERVAS Create	280 052	26.9x
Context Switch Sem.	238 781	23.0x

Figure 4 depicts our performance results normalized to an unprotected baseline, *i.e.*, CVA6 without any memory encryption. The overhead stems primarily from MEMSEC in TEC-tree mode. RVAS adds two calls to Ascon's permutation function to process the tweak in the MEE. RVAS' overhead is 16.7% for CoreMark, 20.0% for LMbench [44] and 24.5% for BEEBS compared to MEMSEC. Figure 4 shows the results of the read-write latency test of LMbench. This benchmark measures the latency for differently sized data chunks and visualizes the impact of CVA6's 32 kB L1 data cache and the latency of the external DDR3 memory. MEMSEC increases the average latency for a memory access from 850 ns to 3300 ns. The two additional rounds in RVAS only increase the latency by another 290 ns on average. These results are encouraging, given that we instantiated our RVAS prototype with the general-purpose MEMSEC encryption framework. We discuss possible optimizations in Sect. 7.3.

We evaluate SERVAS using the microbenchmarks shown in Table 2. We repeat each test 10 000 times to reduce scheduling- and cache-related differences. We compare the number of cycles for `eenter/eexit` with a simple “getpid” system call to get the switching overhead. The “null SM call” is the equivalent of a getpid system call that forms the baseline overhead for SM calls, but instead of calling into the OS, we invoke the SM. Calling an enclave function only takes 3.5x the time of a simple system call. This call includes the time for entering (18 865 cycles), executing a function with a fixed return value, and exiting (17 391 cycles) the enclave. Process-based context switching, using a semaphore and shared memory for synchronization, takes 23.0x of a simple system call. For comparison, entering and exiting an Intel SGX enclave takes 71x the time of a system call [37], thus being significantly slower than invoking a SERVAS enclave.

7.2 Hardware Overhead

We synthesize our modified CVA6 for a Xilinx Kintex-7 series FPGA. The hardware overhead of RVAS consists of the MEE, the ISA extension, and the augmented cache. The design's lookup tables (LUTs) increase by 20.27% and the flip-flops by 19.13%. 61.84 s% of the additional LUTs result from the MEE, 37.26% from the extended cache, and the rest from the ISA extension. Each

cache line is tagged with 125 bit for the memory encryption tweak. The overhead in the cache is 25% for 512-bit cache lines. However, this overhead could be reduced by using the optimizations in Sect. 5.5.

7.3 Prototype Limitations

Our RVAS prototype is not optimized for performance. Due to a lack of openly available high-performance MEEs with authentication, we used the MEMSEC [59] framework. Its storage overhead is analyzed in [59]. The MEE significantly impacts the overall performance (cf. Fig. 4). According to ARM, full memory encryption has a runtime overhead of 7.5% to 25% and a storage overhead of 7.8% to 26.7% [48]. Intel SGX memory encryption is good for medium workloads but might cause worst-case throughput penalties of 400% [28]. Given recent advances in RISC-V, we expect open-source, high-performance MEEs in the future. The bypass optimization Sect. 5.6 could also improve the system performance.

8 Related Work

Intel SGX [17,43] is a set of x86 instructions to interact with enclaves. Unlike SGX, SERVAS dynamically reuses the whole physical memory instead of being limited to the statically allocated 128 MB *Processor Reserved Memory*. Furthermore, SERVAS does not require a trusted metadata storage (*i.e.*, SGX's *Enclave Page Cache Map*) but instead feeds this metadata directly into the encryption.

CrypTag [45] feeds the upper pointer bits into an authenticated encryption engine to achieve memory safety. In contrast, RVAS supports various policies and incorporates information on the CPU state as specified by an SM(cf. Sect. 3) and adds the necessary logic to enforce these policies. The cache area overhead of CrypTag is up to 20%, which is comparable with RVAS.

VAULT [53] makes Intel SGX's EPC available to the full system memory to reduce paging overhead. Unlike SERVAS, VAULT does not overcome SGX's limitation regarding efficient shared memory.

SMARTS [60] implements a Memory Protection Unit as a framework that partially encrypts the memory and partitions the physical DRAM into three regions. In contrast, SERVAS is not bound to a static boot-time memory configuration.

AMD Secure Encrypted Virtualization (SEV) [5,6] describe CPU extensions to run virtual machines in untrusted environments. Unlike RVAS, SEV's memory encryption does neither provide integrity protection nor authentication.

Intel MKTME [32] transparently encrypts memory pages based on one of 64 different encryption keys indicated by the PTE. It does not provide cryptographic authentication and relies on a trusted hypervisor.

Other Systems. Sanctum [18], Keystone [39], and CURE [11] are other recent enclave and TEE designs tackling unique challenges. However, in contrast to SERVAS, these designs do not explicitly protect the external memory from physical attacks using memory encryption.

9 Future Work

We see usage scenarios of RVAS beyond traditional enclaves to provide, for example, fine-grained intra-enclave isolation and system-level enclaves. SERVAS could be used to supersede other protection mechanisms such as memory protection keys [31], pointer authentication [41], pointer tagging [10], and memory coloring [45]. SERVAS specifies configuration registers on each privilege level. These registers can allow for additional protection in the kernel by creating kernel-level enclaves. Our current prototype implementation uses ASCON as it is a lightweight cryptographic primitive already available in MEMSEC. However, realizing RVAS with other encryption primitives, such as AES, would be possible but requires additional analysis, which we leave open for future work.

Remote Attestation is a method to ensure the authenticity of the enclave [25] before provisioning secrets to it. Similar to SecureBlue++ [14], SERVAS loads already encrypted enclaves. Thus, they can embed their secrets directly in the code without the need for a remote attestation service. Based on these secrets, one can easily establish a remote attestation protocol.

10 Conclusion

This paper presented an innovative isolation primitive called authenticryption shield that unifies traditional and advanced isolation policies and offers potential for future security applications. This primitive is built on top of an authenticated memory encryption scheme, thus giving cryptographic isolation guarantees. We demonstrated our design and prototype for RISC-V called SERVAS, which allows for native and secure sharing between enclaves. We show how a small Security Monitor with only 1253 LoC can manage all enclaves throughout their life-cycle with our ISA extension. We prototyped and thoroughly assessed SERVAS's performance on the CVA6 RISC-V hardware and showed that entering or exiting takes only about 3.5x of a getpid syscall.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402) and by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center (grant number 844595), which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria. Furthermore, this work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia.

A Detailed Evaluation Results

See Fig. 5.

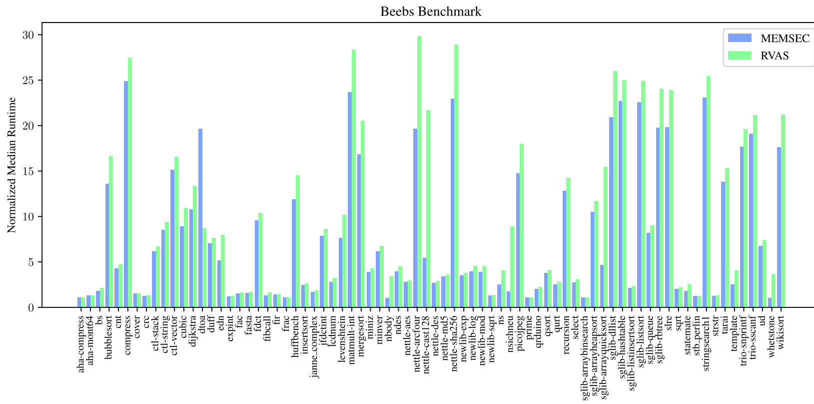


Fig. 5. RVAS performance on the BEEBS benchmark suite compared to MEMSEC, both normalized to an unprotected implementation.

References

1. USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, 10–12 July 2019 (2019)
2. 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, 14–16 August 2019 (2019)
3. 29th USENIX Security Symposium, USENIX Security 2020, 12–14 August 2020 (2020)
4. ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, 14–18 June 2014 (2014)
5. Advanced Micro Devices Inc.: AMD secure encrypted virtualization (SEV) (2020). https://developer.amd.com/sev/
6. Advanced Micro Devices Inc.: AMD SEV-SNP: strengthening VM isolation with integrity protection and more (2020). https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf
7. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: HASP 2013, vol. 13, p. 7 (2013)
8. Andzakovic, D.: Extracting BitLocker keys from a TPM (2019). https://pulsesecurity.co.nz/articles/TPM-sniffing
9. Arm Limited: ARM security technology, building a secure system using TrustZone technology (2009). http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. Ref. no. PRD29-GENC-009492C

10. Arm Limited: Armv8.5-a memory tagging extension (2020). https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf
11. Bahmani, R.: CURE: a security architecture with customizable and resilient enclaves. CoRR abs/2010.15866 (2020)
12. Beer, I.: An iOS zero-click radio proximity exploit odyssey (2020). https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html
13. Biondo, A., Conti, M., Davi, L., Frassetto, T., Sadeghi, A.: The guard's dilemma: efficient code-reuse attacks against Intel SGX. In: USENIX Security 2018, pp. 1213–1227 (2018)
14. Boivie, R.: SecureBlue++: CPU support for secure execution (2020). https://dominoweb.draco.res.ibm.com/reports/rc25287.pdf
15. Bourgeat, T., Lebedev, I.A., Wright, A., Zhang, S., Arvind, Devadas, S.: MI6: secure enclaves in a speculative out-of-order processor. In: MICRO 2019, pp. 42–56 (2019). https://doi.org/10.1145/3352460.3358310
16. Busi, M., et al.: Provably secure isolation for interruptible enclaved execution on small microprocessors. In: CSF 2020, pp. 262–276 (2020). https://doi.org/10.1109/CSF49147.2020.00026
17. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptol. ePrint Arch. **2016**, 86 (2016)
18. Costan, V., Lebedev, I.A., Devadas, S.: Sanctum: minimal hardware extensions for strong software isolation. In: USENIX Security 2016, pp. 857–874 (2016)
19. Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., Adve, V.S.: Nested kernel: an operating system architecture for intra-kernel privilege separation. In: ASPLOS 2015, pp. 191–206 (2015). https://doi.org/10.1145/2694344.2694386
20. Dessouky, G., Frassetto, T., Sadeghi, A.: HybCache: hybrid side-channel-resilient caches for trusted execution environments. In: USENIX Security 2020 [3], pp. 451–468 (2020)
21. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2. Submission to the CAESAR Competition (2016). https://ascon.iaik.tugraz.at/files/asconv12.pdf
22. EEMBC: Coremark (2020). https://www.eembc.org/coremark/
23. Elbaz, R., Champagne, D., Lee, R.B., Torres, L., Sassatelli, G., Guillemin, P.: TEC-Tree: a low-cost, parallelizable tree for efficient defense against memory replay attacks. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 289–302. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74735-2_20
24. Five, H.: MultiZone security for RISC-V (2020). https://hex-five.com/multizone-security-sdk/
25. Francillon, A., Nguyen, Q., Rasmussen, K.B., Tsudik, G.: A minimalist approach to remote attestation. In: DATE 2014, pp. 1–6 (2014). https://doi.org/10.7873/DATE.2014.257
26. Gjerdrum, A.T., Pettersen, R., Johansen, H.D., Johansen, D.: Performance of trusted computing in cloud infrastructures with Intel SGX. In: CLOSER 2017, pp. 668–675 (2017). https://doi.org/10.5220/0006373706680675
27. Goodin, D.: Attackers exploit 0-day vulnerability that gives full control of Android phones (2019). https://arstechnica.com/information-technology/2019/10/attackers-exploit-0day-vulnerability-that-gives-full-control-of-android-phones/

28. Göttel, C.: Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In: SRDS 2018, pp. 133–142 (2018). https://doi.org/10.1109/SRDS.2018.00024
29. Halderman, J.A., et al.: Lest we remember: cold boot attacks on encryption keys. In: USENIX Security 2008, pp. 45–60 (2008)
30. Hedayati, M., et al.: Hodor: intra-process isolation for high-throughput data plane libraries. In: USENIX ATC 2019 [1], pp. 489–504 (2019)
31. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3 (3A, 3B & 3C): System Programming Guide (325384) (2016)
32. Intel Corporation: Intel Architecture Memory Encryption Technologies Specification. Ref: # 336907-002US. Rev: 1.2 (2019)
33. Jang, Y., Lee, J., Lee, S., Kim, T.: SGX-bomb: locking down the processor via rowhammer attack. In: SysTEX 2017, pp. 5:1–5:6 (2017). https://doi.org/10.1145/3152701.3152709
34. Joannou, A., et al.: Efficient tagged memory. In: ICCD 2017, pp. 641–648 (2017). https://doi.org/10.1109/ICCD.2017.112
35. Jomaa, N., Nowak, D., Grimaud, G., Hym, S.: Formal proof of dynamic memory isolation based on MMU. In: TASE 2016, pp. 73–80 (2016). https://doi.org/10.1109/TASE.2016.28
36. Kim, Y., et al.: Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In: ISCA 2014 [2], pp. 361–372. https://doi.org/10.1109/ISCA.2014.6853210
37. Koning, K., Chen, X., Bos, H., Giuffrida, C., Athanasopoulos, E.: No need to hide: protecting safe regions on commodity hardware. In: EUROSYS 2017, pp. 437–452 (2017). https://doi.org/10.1145/3064176.3064217
38. Kossifidis, N.: Secure boot notes (2020). https://lists.riscv.org/g/tech-tee/message/288. E-mail #288 from the tech-teelists.riscv.org group from 2 June 2020
39. Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K., Song, D.: Keystone: an open framework for architecting trusted execution environments. In: EUROSYS 2020, pp. 38:1–38:16 (2020). https://doi.org/10.1145/3342195.3387532
40. Lee, J., et al.: Hacking in darkness: return-oriented programming against secure enclaves. In: USENIX Security 2017, pp. 523–539 (2017)
41. Liljestrand, H., Nyman, T., Wang, K., Perez, C.C., Ekberg, J., Asokan, N.: PAC it up: towards pointer integrity using ARM pointer authentication. In: USENIX Security 2019 [2], pp. 177–194 (2019)
42. McKeen, F., et al.: Intel Software Guard Extensions (Intel SGX) support for dynamic memory management inside an enclave. In: HASP 2016, pp. 1–9 (2016)
43. McKeen, F., et al.: Innovative instructions and software model for isolated execution. In: HASP 2013, p. 10 (2013). https://doi.org/10.1145/2487726.2488368
44. McVoy, L.W., Staelin, C.: lmbench: portable tools for performance analysis. In: USENIX ATC 1996, pp. 279–294 (1996)
45. Nasahl, P., Schilling, R., Werner, M., Hoogerbrugge, J., Medwed, M., Mangard, S.: CrypTag: thwarting physical and logical memory vulnerabilities using cryptographically colored memory. In: ASIA CCS 2021: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, 7–11 June 2021, pp. 200–212 (2021). https://doi.org/10.1145/3433210.3453684
46. Pallister, J., Hollis, S.J., Bennett, J.: BEEBS: open benchmarks for energy measurements on embedded platforms. CoRR abs/1308.5174 (2013)
47. Park, S., Lee, S., Xu, W., Moon, H., Kim, T.: libmpk: Software abstraction for intel memory protection keys (Intel MPK). In: USENIX ATC 2019 [1], pp. 241–254 (2019)

48. Roberto-Maria, A.: Memory protection for the ARM architecture (2020). https://rwc.iacr.org/2020/slides/Avanzi.pdf. Presented at Real World Crypto 2020
49. Schrammel, D., et al.: Donky: domain keys - efficient in-process isolation for RISC-V and x86. In: USENIX Security 2020 [3], pp. 1677–1694 (2020)
50. Seznec, A., Bodin, F.: Skewed-associative caches. In: Bode, A., Reeve, M., Wolf, G. (eds.) PARLE 1993. LNCS, vol. 694, pp. 305–316. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56891-3_24
51. Steinegger, S., Schrammel, D., Weiser, S., Nasahl, P., Mangard, S.: SERVAS! secure enclaves via RISC-V authenticryption shield. CoRR abs/1802.09085 (2021)
52. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: eternal war in memory. In: S&P 2013, pp. 48–62 (2013). https://doi.org/10.1109/SP.2013.13
53. Taassori, M., Shafiee, A., Balasubramonian, R.: VAULT: reducing paging overheads in SGX with efficient integrity verification structures. In: ASPLOS 2018, pp. 665–678 (2018). https://doi.org/10.1145/3173162.3177155
54. Unterluggauer, T., Werner, M., Mangard, S.: MEAS: memory encryption and authentication secure against side-channel attacks. J. Cryptogr. Eng. **9**(2), 137–158 (2018). https://doi.org/10.1007/s13389-018-0180-2
55. Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N.O., Sammler, M., Druschel, P., Garg, D.: ERIM: secure, efficient in-process isolation with protection keys (MPK). In: USENIX Security 2019 [2], pp. 1221–1238 (2019)
56. Waterman, A., Asanović, K.: The RISC-V instruction set manual, volume II: privileged architecture, document version 20190608-priv-msu-ratified (2019). https://riscv.org/specifications/privileged-isa/
57. Weiser, S., Werner, M., Brasser, F., Malenko, M., Mangard, S., Sadeghi, A.: TIMBER-V: tag-isolated memory bringing fine-grained enclaves to RISC-V. In: NDSS 2019 (2019)
58. Werner, M., Unterluggauer, T., Giner, L., Schwarz, M., Gruss, D., Mangard, S.: ScatterCache: thwarting cache attacks via cache set randomization. In: USENIX Security 2019 [2], pp. 675–692 (2019)
59. Werner, M., Unterluggauer, T., Schilling, R., Schaffenrath, D., Mangard, S.: Transparent memory encryption and authentication. In: FPL 2017, pp. 1–6 (2017). https://doi.org/10.23919/FPL.2017.8056797
60. Wong, M.M., Haj-Yahya, J., Chattopadhyay, A.: SMARTS: secure memory assurance of RISC-V trusted SoC. In: HASP 2018, pp. 6:1–6:8 (2018). https://doi.org/10.1145/3214292.3214298
61. Woodruff, J., et al.: The CHERI capability model: revisiting RISC in an age of risk. In: ISCA 2014 [4], pp. 457–468 (2014). https://doi.org/10.1109/ISCA.2014.6853201
62. Wu, H., Preneel, B.: AEGIS: a fast authenticated encryption algorithm v1.1. Submission to the CAESAR Competition (2016). https://competitions.cr.yp.to/round3/aegisv11.pdf
63. Zaruba, F., Benini, L.: 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. **27**, 2629–2640 (2019). https://doi.org/10.1109/TVLSI.2019.2926114
64. Zeldovich, N., Kannan, H., Dalton, M., Kozyrakis, C.: Hardware enforcement of application security policies using tagged memory. In: OSDI 2008, pp. 225–240 (2008)