TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V

Samuel Weiser*, Mario Werner*, Ferdinand Brasser†, Maja Malenko*, Stefan Mangard*, Ahmad-Reza Sadeghi†
* Graz University of Technology: {samuel.weiser, mario.werner, stefan.mangard}@iaik.tugraz.at, malenko@tugraz.at
† TU Darmstadt: ferdinand.brasser@trust.tu-darmstadt.de, ahmad.sadeghi@trust.cased.de.

***Abstract*—Embedded computing devices are used on a large scale in the emerging internet of things (IoT). However, their wide deployment raises the incentive for attackers to target these devices, as demonstrated by several recent attacks. As IoT devices are built for long service life, means are required to protect sensitive code *in the presence of potential vulnerabilities*, which might be discovered long after deployment. Tagged memory has been proposed as a mechanism to enforce various fine-grained security policies at runtime. However, none of the existing tagged memory schemes provides efficient and flexible compartmentalization in terms of isolated execution environments.**

We present TIMBER-V, a new tagged memory architecture featuring flexible and efficient isolation of code and data on small embedded systems. We overcome several limitations of previous schemes. We augment tag isolation with a memory protection unit to isolate individual processes, while maintaining low memory overhead. TIMBER-V significantly reduces the problem of memory fragmentation, and improves dynamic reuse of untrusted memory across security boundaries. TIMBER-V enables novel sharing of execution stacks across different security domains, in addition to interleaved heaps. TIMBER-V is compatible to existing code, supports real-time constraints and is open source. We show the efficiency of TIMBER-V by evaluating our proof-of-concept implementation on the RISC-V simulator.

I. Introduction

With ongoing advances in miniaturization and energy efficiency, computing devices are rapidly penetrating everyday life. Due to long service life, security of such devices becomes decisive. In in the recent past, we have been witnessing attacks on millions of cameras and routers [39], cars [38], cardiac devices [35] and light bulbs [41], to name a few. The high code complexity of these devices fosters programming bugs, making their exploitation only a matter of time. This attenuates potential IoT use cases since a compromise could have immediate monetary, legal or privacy consequences [26]. Also, the protection of intellectual property (IP) in a highly diverse market like the IoT, which integrates code from multiple vendors, requires strong security guarantees.

Isolated execution protects sensitive code and data on devices with compromised or untrusted software, and has been proposed for different systems, with and without virtual memory [2,3,7,9,10,13,21,22,25,28,34,37,40,47]. Especially small resource-constrained devices often suffer from poor memory utilization due to memory fragmentation and inefficient isolation mechanisms. A tighter integration of trusted memory in the limited physical address space would demand fine-grained isolation boundaries, which existing schemes either do not provide at all, or provide only at the expense of high management overhead. Also, more flexible isolation mechanisms are important for dynamically managing trusted memory. A technique that has the potential to offer fine-grained and flexible isolation boundaries is *tagged memory*. Tagged memory transparently associates blocks of memory with additional metadata. It has been used for dynamic information flow tracking [48] as well as access control [53], and is still an active subject of research [31,45]. While tagged memory has been shown to support a variety of security policies like protection of control data [14], pointers [18] or capabilities [52], strong, efficient and flexible isolated execution is still an open problem for small embedded systems. In particular, data flow isolation [45] cannot provide strong isolation since tags can be destructively written by untrusted software. Other existing solutions are not appropriate for low-end embedded devices due to their memory overhead stemming from large tags [54] or fully programmable but expensive tag engines [11,17,19,49]. Hence, currently no existing tagged memory schemes supports efficient isolated execution on small embedded devices.

In this work, we propose TIMBER-V, a tagged memory architecture which brings efficient isolated execution in form of enclaves to low-end devices. Since isolated execution is still not well researched for low-end RISC-V processors, we prototype TIMBER-V on the open RISC-V architecture via a hardware-software co-design. On the hardware side, we achieve fine-grained in-process isolation with only two tag bits. Moreover, we combine tagged memory with a memory protection unit (MPU) to support an arbitrary number of processes while avoiding the overhead of large tags [54]. On the software side, we enforce isolated execution via a small trust manager, called TagRoot. We isolate privileged from unprivileged security domains, supporting both, Intel SGX enclaves [37] and the TrustZone [3] programming model, however, with much finer isolation granularity and more efficient memory utilization. This has several advantages. On the one hand, data locality can be maintained by interleaving trusted and untrusted memory, thus minimizing memory fragmentation. On the other hand, TIMBER-V uses a tag update policy which allows highly flexible dynamic memory management of trusted data. Dynamic memory support has been announced for the upcoming Intel SGXv2 which involves costly interaction with the operating system [29]. In contrast, TIMBER-V enclaves can instantaneously claim memory by

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23068
www.ndss-symposium.org

using a single instruction. To demonstrate these advantages, we show heap interleaving and a novel stack interleaving scheme. That is, we use a single heap and stack across different security domains while maintaining strong isolation. Moreover, we demonstrate highly efficient inter-enclave communication over secure shared memory. We support real-time constraints by making all trusted software interruptible. We implement and benchmark TIMBER-V on the RISC-V Spike simulator, allowing an evaluation under different CPU models which highlights characteristics of TIMBER-V rather than CPU implementation specifics. We show that the runtime overhead of TIMBER-V is 25.2% for naive implementations while tag caching reduces the overhead to 2.6%.

In summary, our main contributions are:

- We propose TIMBER-V, the first efficient tagged memory architecture for isolated execution on low-end processors
- We present a novel concept called stack interleaving that allows for efficient and dynamic memory management
- We propose lightweight shared memory between enclaves
- We propose an efficient shared MPU design
- We extensively evaluated our proof-of-concept implementation[1] on the RISC-V simulator for different CPU models

II. Background

This section gives background information about related security architectures, RISC-V and tagged memory.

Security Architectures. Process isolation is a fundamental security concept which combines hardware and software techniques to isolate the memory of processes from each other. It is usually enforced by the operating system taking advantage of processor's privilege modes. Large systems isolate processes in separate virtual address spaces with the help of a memory management unit (MMU), while resource-constrained devices use a memory protection unit (MPU), suitable for single address space implementations. However, these isolation mechanisms can be circumvented, as modern operating systems are becoming large and complex, and their exploitation becomes easier. Recently, there has been a lot of research towards isolated execution environments which exclude the operating system from the trusted computing base (TCB). Isolated execution protects security-critical code in isolated compartments, ensuring its confidentiality and integrity even in the presence of malicious privileged software. An outside module can call these compartments only at their designated secure entry points. Two widely deployed architectures for isolated execution are ARM TrustZone [2] and Intel SGX [37].

ARM TrustZone [2] is a hardware security extension which partitions computer's resources into a secure and a non-secure world. This world split is orthogonal to the processor's privilege modes and effectively creates a secure virtual CPU. The secure world can access all system resources, while the non-secure world can only access non-secure memory regions. This way, sensitive code can be hidden from the non-secure world. The TrustZone concept demands a security kernel for managing the secure world. This includes process and memory management as well as scheduling, which enlarges the TCB. Non-secure code must use a single entry point to a secure monitor handler to switch worlds. ARM TrustZone-M [3] integrates the TrustZone concept into smaller Cortex-M processors. Non-secure applications can call secure applications through multiple designated entry points, specified via non-secure callable regions.

The central concept of Intel SGX [37] is a hardware-isolated container, called an enclave, in which sensitive parts of an application are placed. Unlike TrustZone, an enclave directly resides in the address space of a user process. SGX does not rely on any privileged software (trusted kernel, hypervisor, etc.) to isolate enclaves, thus reducing the TCB to only the CPU and enclaves themselves. However, SGX's management instructions involve pretty complex microcode.

RISC-V. RISC-V [50] is an open and extensible instruction set architecture and defines three privilege modes [51], namely machine-mode (M), supervisor-mode (S), and user-mode (U). M-mode has the highest privileges and is used for emulating missing hardware features. S-mode and U-mode are meant to run an operating system and user applications, respectively.

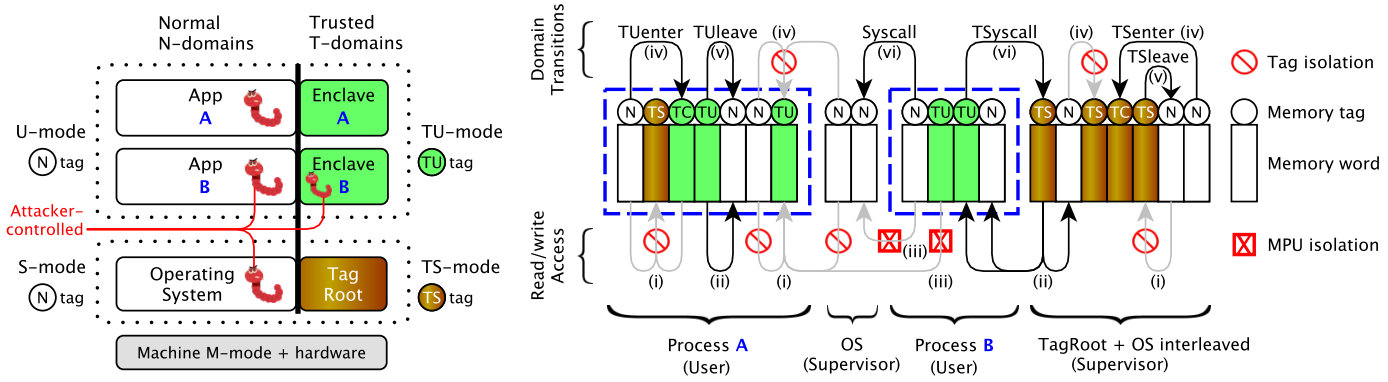
Tagged Memory. The idea behind tagged memory is to extend each memory word with additional bits that store metadata. The general tagged architecture concept is very old and can already be found in numerous early computer designs [23]. There, tag bits were, for example, used for debugging as well as for dynamically tracking the numeric type of data words. Recent commercially available computer architectures hardly support hardware-based tagged memory. Schemes that associate memory with metadata, like for example dynamic analysis tools [43,44,46], rely on software-based solutions instead. However, recent research on tagged-memory architectures in the system security context [8,19,45] hints that re-establishing hardware-support can considerably improve security.

III. Adversary Model and Design Goals

A stakeholder wants to securely execute pieces of code on a small IoT device. However, the stakeholder distrusts the IoT device for various possible reasons. First, the device's operating system might not sufficiently isolate individual tasks to guarantee secure code execution, as is the case for the popular FreeRTOS kernel, for example.[2] Second, even if the operating system provides sufficient task isolation, it might be subject to exploitation, circumventing all isolation guarantees [27]. Third, the operating system might be controlled by a party whom the stakeholder distrusts and wants to protect intellectual property against. We consider the strongest attacker to have complete control over the operating system. Thus, he can not only deny service but also use the system's security features to spawn malicious enclaves in an attempt to subvert benign ones, as depicted in Figure 1a. However, we assume that benign enclaves are properly protected against direct exploitation via runtime attacks using concepts like memory safety [16], for example. A proper tag isolation architecture shall guarantee security of benign enclaves in the presence of such attacks. We assume that cryptographic primitives are secure. We do not address physical attacks. The trusted computing base consists of the hardware, including the hardware emulation mode (M-mode), as well as a small trust manager (TagRoot).

[1] The source code is available at https://github.com/IAIK/timber-v

[2] FreeRTOS allows to elevate privileges via the prvRaisePrivilege syscall.



(a) TIMBER-V supports four security domains.

(b) Security domains are interleaved in flat physical memory.

Fig. 1: (a) TIMBER-V extends user apps running in U-mode and the operating system running in supervisor S-mode with trusted memory, namely TU-mode for enclaves and TS-mode for TagRoot. User processes **A** and **B** integrate trusted enclave memory within untrusted apps. The attacker controls all software in the N-domains and can run malicious enclaves (cf. enclave B). (b) Tag isolation protects T-domains while MPU isolation encapsulates and protects individual processes across domains. T-domains can only be entered at trusted callable entry points (TC-tag), which allows fast domain transitions.

TIMBER-V does not prevent software side-channel attacks. While memory interleaving provides the untrusted software with additional information about enclave's memory allocations, an enclave that follows the constant-time paradigm [5, 12] is secure against any address-based side-channel attack.

We demand that a tagged memory architecture designed for isolated execution shall meet the following design goals:

- **G1 Security.** It shall guarantee that sensitive code can leverage strong isolation to maintain confidentiality and integrity of its sensitive data. This demands (i) strong memory isolation, (ii) secure entry points, (iii) secure communication and (iv) attestation and sealing.
- **G2 Flexibility.** It shall be flexible with respect to fine-grained and dynamically reconfigurable isolation boundaries as well as the programming model.
- **G3 Compatibility.** Untrusted code shall run without modification to support existing operating systems and apps.
- **G4 Low Overhead.** It shall minimize the cost of tagged memory as well as the performance overhead of switching security domains.
- **G5 Real-time.** It shall support hard real-time constraints.

IV. TIMBER-V Design

TIMBER-V is a novel tagged memory architecture achieving lightweight, yet powerful isolated execution on small embedded processors. Specifically, we achieve fine-grained and dynamic in-process isolation. TIMBER-V follows a hardware-software co-design. On the hardware side, TIMBER-V uses tagged memory for enforcing a strong and fine-grained isolation policy and for providing fast domain switches. Tagged memory is augmented with a Memory Protection Unit (MPU) for lightweight isolation between processes. Dedicated tag instructions allow flexible dynamic memory management. For example, we demonstrate memory interleaving across security domains not only for heap memory but also for stack memory. On the software side, TIMBER-V delegates policy enforcement to a small privileged trust manager called TagRoot, which provides various trusted services to the operating system and to enclaves.

A. Isolated Execution

TIMBER-V supports four security domains, as depicted in Figure 1a. The operating system and apps live in the "normal" N-domains, which are considered untrusted. The N-domains support the traditional split between user (U-mode) and supervisor (S-mode) and allow existing code to run without modification (goal **G3**). Sensitive memory is protected via fine-grained memory tagging, which creates islands of trusted memory inside the N-domains. Trusted user mode (TU-mode) can be leveraged for isolated execution environments, called enclaves. Moreover, trusted supervisor mode (TS-mode) allows to run a trust manager like TagRoot, augmenting the untrusted operating system with trusted services. To achieve this, TIMBER-V combines security domain isolation with MPU-based process isolation. The trusted domains are protected by a strict tagged memory policy, which we denote as *tag isolation*. Individual processes or enclaves are protected via *MPU isolation*. Memory accesses are only permitted if *both* mechanisms "agree". This allows a variety of different programming models, as demanded by goal **G2**. For example, we achieve TrustZone's [3] security split via memory tags, however with much finer and highly dynamic isolation boundaries. Also, TIMBER-V can embed enclaves directly in user processes, as done in Intel SGX-like designs [37], however, again with the benefits of tagged memory.

Tag Isolation. TIMBER-V uses a two-bit tag per 32-bit memory word for fine-grained protection of trusted memory (goal **G2**). Having only two tag bits keeps the hardware cost of tagged memory low, achieving goal **G4** while at the same time retaining advantages for fine-grained memory isolation. With two tag bits we encode four different tags, namely N-tag, TU-tag, TS-tag and TC-tag. We use them to identify untrusted memory (N-tag), trusted user memory (TU-tag), trusted supervisor memory (TS-tag) as well as secure entry points via the trusted callable TC-tag. Tag isolation is depicted with arrows in Figure 1b and will be discussed in detail in Section VII. At every memory access, a hardware tag engine ensures that trusted memory cannot be accessed from untrusted code (i). Moreover, trusted supervisor memory (TS-tag) used

for TagRoot cannot be accessed from enclaves (TU-tag). In contrast, trusted domains can access lesser trusted memory (ii), as long as the MPU isolation policy allows it. Finally, tag isolation could be directly applied to other peripherals, e.g., preventing DMA accesses to trusted memory.

MPU Isolation. Tag isolation enforces protection of security domains. However, an embedded system typically runs several independent processes within the *same* security domain. Relying on tag isolation for process isolation would require large tags, which is unacceptable for our goal **G4**. TIMBER-V isolates individual processes via a memory protection unit (MPU) (see dashed boxes and arrows (iii) in Figure 1b). This minimizes tagging overhead while supporting fine-grained in-process isolation.

Fast Domain Transitions. Our system distinguishes horizontal and vertical domain transitions, as shown in the upper half of Figure 1b. Both need to be fast and efficient to achieve goal **G4**. Horizontal transitions switch between N and T-domains while maintaining the current privilege mode. To avoid code-reuse attacks, trusted domains can only be entered at secure entry points (iv) (cf. goal **G1**). Entry points are marked as trusted callable with the TC-tag and are denoted as "TUenter" and "TSenter", depending on the caller's privilege mode (iv). Whenever the CPU fetches an instruction tagged with TC-tag, it switches to the trusted security domains. Likewise, when fetching normal N-tag memory, the CPU switches back to the normal N-domains, leaving trusted execution, denoted as "TUleave" and "TSleave" (v), respectively. More details about how TUenter and TSenter are protected will be discussed in Section VII. Unlike SGX, which involves costly checks in microcode for each domain switch [29], our design imposes zero runtime overhead. Unlike TrustZone-M [3], it allows even faster and very compact transitions, keeping code locality and compatibility to the maximum extent possible.[3]

Vertical transitions are in fact syscalls (vi). In the N-domains, apps can issue syscalls to the operating system. Likewise, in the T-domains, enclaves can request TagRoot services via trusted TSyscalls. When finished, a syscall or TSyscall can return to the calling app or enclave, respectively. To cleanly separate vertical transitions, TIMBER-V adds a separate trusted syscall (trap) handler.

MPU Sharing. TIMBER-V shares a single MPU between the N-domains and the T-domains. That is, the same MPU slots can be used for processes executing in U-mode and in TU-mode. Thus, TIMBER-V not only supports traditional apps and secure enclaves but also mixed processes, as shown in Figure 1. In contrast to using two separate MPUs, our approach reduces hardware and energy costs since fewer MPU slots are required. To maintain compatibility (goal **G3**), the operating system can always update shared MPU slots. Any such updates are detected by the MPU which then prevents enclaves from using the updated slots until TS-mode validates the changes. To do so, we augment the MPU with just two additional flags.

[3]For example, one could split an unmodified program binary into untrusted and trusted parts by mere tagging, that is, without the need for changing code or the memory layout. However, in practice one typically augments the program with secure argument passing, stack handling and register cleanup.

TABLE I: Tag update policy, permitting (✓) or refusing (✗) tag updates from certain security domains.

Can update tag	N-tag	TC-tag	TU-tag	TS-tag
N-domains	✓	✗	✗	✗
TU-mode	✓	✗	✓	✗
TS-mode	✓	✓	✓	✓
M-mode	✓	✓	✓	✓

B. Dynamic Memory Management

TIMBER-V supports highly flexible management of trusted memory. For this, new tag-aware instructions are added which act according to a tag update policy. Using these instructions, we show dynamic memory interleaving as well as a simple but effective code hardening transformation.

New Tag-aware Instructions. TIMBER-V adds new checked memory instructions which allow fine-grained and dynamic management of trusted memory. We call them "checked" instructions, since they augment ordinary memory instructions adhering to tag isolation with one additional programmable tag check. This additional tag check does not bypass our tag isolation policy but tightens it by constraining memory accesses to a specific security domain. For example, when enclaves process untrusted data, they can use checked instructions to prevent accidentally accessing a wrong security domain.

Tag Update. In addition to the tag checks, checked store instructions allow to (de)privilege memory by changing memory tags as follows. Tags can only be updated within the same or a lower security domain but cannot be used to elevate privileges, as shown in Table I. TS-mode (and M-mode) have full access to all tags. TU-mode can only change tags between N-tag and TU-tag to support dynamic interleaving of user memory. We prevent TU-mode from manipulating TC-tags, which are reserved for secure entry points. Our tag update policy makes isolation boundaries flexible during runtime (goal **G2**).

Dynamic Memory Interleaving. Checked memory instructions allow to dynamically claim memory across security domains, thus maintaining data locality and reducing management overhead. For example, an enclave can claim untrusted memory during runtime by setting its tags from N-tag to TU-tag. We show that this allows heap interleaving as well as a novel code transformation that we call stack interleaving. That is, an enclave does not need to maintain a separate secure heap or stack. In general, dynamic memory interleaving can help reduce memory requirements to a single heap and a single stack per execution thread. This has not only operational advantages like reduced memory fragmentation and thus reduced memory consumption but also security gains, since dynamic memory management can be removed from the trusted computing base (TCB).

Code Hardening Transformation. Checked instructions can be used for additional code hardening against code-reuse attacks. In these attacks, one misuses existing code to perform malicious actions, e.g., leak secrets from trusted to untrusted domains. In contrast, normal code execution usually operates in a single security domain and all accessed memory tags are predetermined by this security domain. Our code hardening transformation enforces this property by replacing memory instructions with checked instructions, checking for the correct

TABLE II: TagRoot trusted OS and enclave services.

Trusted OS services	Trusted enclave services
`create-enclave(ecb)`	`get-key(id)`
`add-region(ecb, region)`	`shm-offer(targetEID, region)`
`add-data(ecb, region)`	`shm-accept(ownerEID)`
`add-entries(ecb, entries)`	`shm-release(region)`
`init-enclave(ecb)`	
`load-enclave(ecb)`	
`destroy-enclave(ecb)`	
`resume()`	

tag and restraining code execution to the current security domain. Only code interacting with untrusted memory on purpose is left unmodified. As discussed later, this transformation adds negligible performance overhead and we apply it to enclaves and TagRoot as an additional layer of defense.

C. Trusted Services

We provide a small trust manager, called TagRoot, which serves as trust anchor for bootstrapping secure enclaves and maintaining isolated execution, as demanded by goal **G1**. TagRoot offers trusted OS services to the untrusted operating system as well as trusted enclave services to the enclaves themselves. This includes enclave management, secure entry points, attestation and sealing. Moreover, in contrast to existing solutions, TagRoot supports fast enclave to enclave communication via secure shared memory, which imposes zero copying overhead and allows m:n connectivity. TagRoot and enclaves are fully interruptible, thus meeting goal **G5**.

Enclave Life Cycle. TIMBER-V enclaves are created and loaded within an ordinary user process at the discretion of the operating system but with assistance of TagRoot. Once loaded, enclaves can be directly invoked by user apps to carry out security-critical tasks. For freshly generated enclaves, one typically provisions secret data like cryptographic keys to the enclave via a secure remote channel. This channel is authenticated using enclave attestation with assistance of TagRoot. During its lifetime, enclaves can authenticate and communicate with other enclaves or seal sensitive information for keeping state across reboots, again with the help of TagRoot.

Possible Extensions. Independently of our TagRoot design, TIMBER-V allows other trust manager designs as well. For example, implementing trusted I/O is straight forward by tagging I/O memory as trusted. Also, trusted scheduling services requiring availability guarantees can be easily implemented in TS-mode. However, since these additional services enlarge the TCB, we did not implement them in our current prototype. We discuss different design options in Section XI.

V. TagRoot Trust Manager

We develop a small trust manager for TIMBER-V, called TagRoot. It runs in trusted supervisor mode (TS-mode) and offers privileged trusted services to the untrusted operating system as well as unprivileged trusted services to enclaves. All trusted services are listed in Table II.

A. Trusted OS Services

Trusted OS services can be invoked by the operating system via TSenter (see (iv) in Figure 1b) and provide enclave management like creation and cleanup, loading as well as handling of interruption. Also, trusted OS service calls define the enclave identity used for subsequent trusted enclave services.

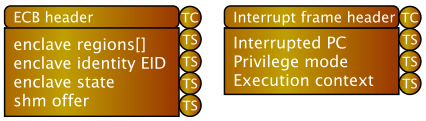


Fig. 2: TagRoot trusted metadata includes enclave control blocks (ECB) and interrupt frames with unforgeable headers.

Creation and Cleanup. Enclaves are identified by a data structure called enclave control block (ECB) which is kept in secure TS-mode memory, as shown in Figure 2. ECBs are created via `create-enclave`. When creating a new enclave, the operating system can add memory regions (contiguous chunks of memory) to it via `add-region`. These enclave regions will be loaded in the MPU when the enclave is about to run. TagRoot ensures that enclave regions will never overlap with other enclaves but are unique to each enclave. However, as mentioned before, an enclave region can cover app memory as well. Thus, a single shared MPU region can hold enclave data and app data. This is achieved by executing `add-data`, which claims enclave memory by setting TU-tag, as long as the claimed memory is within the enclave's regions. `add-data` works on a word granularity, thus supporting fine-grained in-process memory interleaving. All claimed memory (TU-tag) constitutes the actual enclave (TU-mode), while the rest (N-tag) constitutes the untrusted app (U-mode) (cf. processes in Figure 1b). While the enclave can access its app counterpart, the opposite direction is prohibited by the tag isolation policy. Similar to enclave data, entry points are announced by a call to `add-entries`. TagRoot will mark all entry points with TC-tag, given that they belong to the enclave's regions. Finally, a call to `init-enclave` will cause TagRoot to compute a cryptographic identity over the enclave and mark it as runnable. Once the enclave is initialized, it cannot be altered using the above trusted service calls but only loaded, resumed or destroyed. At the end of an enclave's life cycle, a call to `destroy-enclave` will unload and invalidate the ECB, preventing the enclave from further execution, clear all claimed enclave memory, release enclave regions and clear up ECB memory. This also reverts all enclave tags to N-tag.

Loading enclaves. In order to run an enclave, the operating system first loads the enclave regions into the MPU and then calls `load-enclave`. If another enclave is currently loaded, TagRoot unloads it by invalidating stale enclave MPU slots. Next, TagRoot validates the current MPU configuration, as configured by the operating system, by acknowledging all updated MPU slots that correspond to the enclave. Moreover, TagRoot locks the enclave's ECB to prevent further modifications and restores its runnable or interrupted state in a special register, called `STSTATUS`. Now that the enclave is loaded, it can be entered from the app by a simple call to one of its entry points (TUenter), or in case of interruption, it can be resumed.

Interruptibility. Trusted code execution is fully interruptible except for a small trusted interrupt handler. Interruptibility is necessary to support real-time tasks reacting on external I/O events or control loops that need to run periodically in order to meet certain stability criteria, for example. Whenever

an interrupt happens during enclave execution, TIMBER-V raises a special "interrupted" CPU flag that prevents re-entering the enclave and calls the trusted trap vector of TagRoot. TagRoot then saves the current enclave's execution context in a protected interrupt frame (see Figure 2) and erases sensitive CPU registers to avoid accidental leakage of sensitive data. Moreover, it sets the interrupted program counter to a dedicated `resume` function, before giving control to the operating system. When the operating system returns from interrupt handling, `resume` gets executed. TagRoot restores the enclave execution context, clears the "interrupted" CPU flag and resumes enclave execution. This process is completely transparent and requires no changes to the operating system. Moreover, it also supports interruption of TagRoot (TS-mode) while processing trusted service calls.

Enclave Identity. From a functional perspective, enclaves are defined by their code base and initial data as well as their entry points. To capture this, all trusted OS service calls from `create-enclave` to `init-enclave` contribute to a continuous SHA256 computation, called measurement. The measurement involves not only the sequence of trusted service calls but also its parameters, that is, enclave regions, data as well as entry points. The measured data is immutable until `init-enclave` stores the final measurement as enclave identity (EID) inside the ECB and marks the enclave state as runnable (see Figure 2). Thus, the EID reliably identifies enclaves. This concept is similar to MRENCLAVE in SGX [29]. Enclave identities are used for trusted enclave services.

B. Trusted Enclave Services

Enclaves can request trusted enclave services via TSyscalls (see (iv) in Figure 1b). This includes sealing, attestation and inter-enclave communication via shared memory.

Sealing and Remote Attestation. An enclave can call `get-key` to generate enclave-specific cryptographic keys [1], derived from the enclave identity (EID) and a secret platform key K_p, which is only known to TagRoot and remote verifiers. The keys are derived as follows: $k^{id}_{\texttt{EID}} = \mathrm{HMAC}_{K_p}(\texttt{EID}, id)$. By providing an additional key identifier id, the enclave can request keys for different purposes. For example, it can derive sealing keys for encrypting and decrypting sensitive data for secure offline storage. Also, it can derive remote attestation keys to compute a message authentication code (MAC) over a challenge given by a remote verifier. The remote verifier knowing the platform key K_p can then recompute the MAC, thus remotely attesting the enclave. TagRoot can be easily extended to asymmetric remote attestation protocols [1].

Secure Shared Memory. TagRoot supports secure shared memory (shm) as a fast and flexible inter-enclave communication method. An enclave can offer another "target" enclave shared memory access to parts of its own enclave memory regions via `shm-offer`. TIMBER-V creates a special entry in the offering enclave's control block (ECB), covering the offered shm region and the target enclave's EID. For this, the target enclave does not need to exist yet. It can independently accept the shm offer via `shm-accept`, which expects the offering enclave's EID as argument. When accepting shm, TagRoot scans the existing ECBs to find the offering enclave via its EID. In case a valid shm offer exists, TagRoot adds the offered shm region to the target enclave's regions in the ECB and also returns the memory region's pointer back to the enclave to help it use the shared memory. Once an enclave has accepted a new shared memory region, it has to notify the untrusted operating system to load the shm region into the MPU. The target enclave can close an accepted shm by issuing `shm-release`, which removes the shm from the enclave's memory ranges. An offering enclave can withdraw a pending offer by offering the empty region, however it cannot close an accepted offer. This is because TagRoot only manipulates the ECB of the calling enclave but not the one of the communication partner.

Our secure shared memory allows m:n connectivity between enclaves, where m is the number of offers an enclave can make and n is the number of offers a target enclave can accept. m is unlimited and n is only limited by the number of enclave regions that can be stored in the ECB, which is an implementation-defined constant. Moreover, TagRoot's shared memory supports a transitive trust model. An owner enclave could subsequently offer the same shared memory to other target enclaves, thus minimizing memory usage in case of broadcast channels, for example.

Local Attestation. Local attestation is implicitly achieved using shared memory without the involvement of cryptographic secrets. By offering and accepting shared memory, both involved enclaves identify their communication partner via its EID, thus mutually attesting each other.

VI. Dynamic Memory Management

TIMBER-V provides highly flexible and dynamic memory management. Memory can be claimed by different security domains during runtime with fine granularity. Dynamic memory has been an issue for isolated execution before. For example, Intel SGX adds dynamic management of enclave pages in SGXv2 via separate trusted service calls in microcode. In contrast to Intel SGX, TIMBER-V naturally supports much finer grained dynamic memory management by simply updating tags. User software can directly claim or release memory via checked store instruction without the need for trusted service calls. This high flexibility and efficiency enables novel application scenarios like dynamic memory interleaving schemes. Memory interleaving minimizes memory fragmentation by keeping data locality across security domains. For example, when passing large untrusted data structures to an enclave, the enclave could avoid copying the data to enclave memory by just updating tags. Thus, the data structures remain interleaved within the untrusted memory. In the same way memory interleaving can be used for dynamic memory management—the dynamic allocation and deallocation of trusted memory.

In this section, we first explain heap interleaving from which we develop stack interleaving, a novel memory interleaving scheme. We do this for both, TagRoot and enclaves, and show that we can entirely outsource dynamic memory from TagRoot to the untrusted operating system, thus reducing the TCB. Finally, we show that stack interleaving supports interrupts with arbitrary nesting levels.

A. Heap Interleaving

Heap interleaving reuses an untrusted heap to store trusted data. To do so, trusted code first instructs untrusted code to

allocate a chunk of memory on its heap. The precise heap layout is irrelevant as long as the requested memory chunk lies within N-tagged memory. Since the complex task of memory allocation is now outsourced to the untrusted domains, the TCB can be significantly reduced. Next, the trusted code claims the allocated memory chunk. This is done via checked store instructions, which atomically check memory for N-tag and update it to TS-tag or TU-tag, respectively. This protects the newly created trusted heap object against malicious access from the N-domains. However, care must be taken to reliably identify trusted heap objects during their lifetime. To free a trusted heap object, the trusted code simply clears it and reverts its tags to N-tag by means of checked store instructions, and notifies the untrusted code to do the heap cleanup.

User Heap Interleaving. Typically, an enclave actively requests heap space for trusted heap objects, which it uses internally to satisfy its dynamic memory demand. To reliably identify a trusted heap object, enclaves should always keep a pointer to it inside protected enclave memory and only use this pointer to reference the trusted object. If enclaves would interpret untrusted function arguments as trusted heap pointers, memory corruption attacks become possible.

Supervisor Heap Interleaving. When creating a new enclave, the operating system allocates a trusted enclave control block (ECB) on its heap and calls `create-enclave`, which claims the ECB for TS-mode. Since most trusted OS service calls take the ECB as argument from the untrusted OS, TagRoot needs means to verify its validity. This is done in two steps. First, TagRoot accesses an ECB only via checked memory instructions, checking for TS-tag. This prevents misinterpreting untrusted data as ECB. Second, since the ECB argument could point to arbitrary TS-tagged memory, TagRoot identifies valid ECBs via an unforgeable header at the start of each ECB, as will be explained in Section VI-C.

B. Stack Interleaving

Stack protection is crucial for isolated execution. Typically, an execution thread is given individual stacks for every security domain it can exercise. For example, SGX enclaves use separate secure stacks which are isolated from their hosting app. Also, operating systems usually maintain separate kernel stacks for each app. With TIMBER-V we can reuse the same stack across different security domains, thus removing the need for maintaining multiple stacks per execution thread. This reduces memory fragmentation, which is particularly relevant for the limited physical address space of low-end embedded systems.

Stack interleaving is a simple program transformation that inserts additional stack allocation code. Whenever allocating a new stack frame, we claim this memory using checked store instructions, checking memory for N-tag and updating it to TS-tag or TU-tag, respectively. When deallocating the stack frame, we clear it and revert the tags to N-tag via checked stores. As with heap interleaving, one needs means to check validity of dynamic memory, that is, validity of stack pointers. We show stack interleaving (i) horizontally within supervisor mode, (ii) horizontally within user mode, and (iii) vertically across TSyscalls. We implement stack interleaving in a separate compilation step and defer details to Section VII.

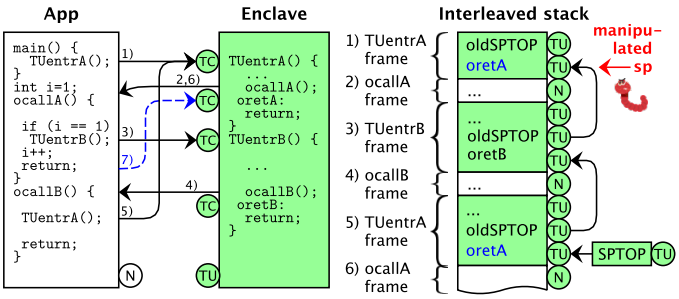


Fig. 3: User stack interleaving with nested TUenter and ocall.

Horizontal Supervisor Stack Interleaving. When receiving trusted OS service calls (TSenter), TagRoot reuses the S-mode stack maintained by the untrusted operating system. Validity of the stack pointer is implicitly checked by our stack interleaving transformation, checking untrusted memory for N-tag before claiming it. This prevents TagRoot from accidentally overwriting trusted memory. If the untrusted operating system provides an invalid `sp`, it can only break system's availability, which it can do anyway. While processing trusted service calls, `sp` cannot be manipulated because TagRoot does not leave TS-mode until the service call is finished (or interrupted).

Horizontal User Stack Interleaving. When transitioning from an untrusted app to an enclave (TUenter), the enclave claims and releases stack frames on the untrusted app's stack. An enclave might call untrusted functions from the outside, e.g., to request dynamic heap memory or file access. Such transitions are named "ocalls" and demand special treatment. First, a finished ocall needs to return to the enclave's call site, denoted as "oret". We achieve this by making the oret sites callable using TC-tag, as depicted in Figure 3. Second, orets need to be protected against misuse, as follows: An attacker could directly jump to an oret without a corresponding ocall and thus perform code reuse attacks. We address this by securely pushing the return address (*i.e.*, the address of oret) onto the stack before doing the ocall and verifying it afterwards. Thus, an attacker can only jump into active orets. However, the attacker could point `sp` to arbitrary trusted data that contains a valid return address. E.g., he could confuse the nesting level of multiple ocalls by returning to a previous ocall rather than the latest one. Consider the code in Figure 3, where both, `TUentrA` and `TUentrB` perform ocalls, leading to a nested call sequence denoted with numbers 1) to 6). When returning from `ocallA` in step 7), an attacker could confuse the context of `oretA` by pointing `sp` to the first `TUentrA` frame instead of the correct fifth one (see upper right corner). We prevent this by verifying the stack pointer `sp` at each enclave oret site against `SPTOP`, which holds the `sp` of the latest ocall in trusted enclave memory. To support nesting, we securely push the previous `SPTOP` onto the stack and restore it afterwards.

Vertical Stack Interleaving. When enclaves request trusted services via a TSyscall, TagRoot reuses the enclave's stack in the same way as outlined before. However, care must be taken since the stack is now interleaved across different privilege modes. Before TagRoot uses the enclave stack, it has to ensure that `sp` points into the current enclave's memory and that it has enough space for processing the TSyscall. The stack requirements of TSyscalls can be statically determined by

means of profiling or static code analysis. In addition, the stack needs to be able to hold one interrupt frame.

Interrupt Handling. Stack interleaving naturally supports interrupt handling. As outlined in Section V-A, on interruption of trusted code TagRoot stores the current execution context in a secure interrupt frame. With stack interleaving, TagRoot can directly store the interrupt frame on the current stack. As with ocalls, care must be taken since the untrusted operating system can manipulate the stack pointer before resuming from interruption. However, unlike before, we cannot keep a copy of the last valid `sp` in secure memory (like `SPTOP`) because the operating system might resume a different interrupted enclave first or resume an interrupted TagRoot service call. To allow TagRoot distinguishing valid interrupt frames from other TS-tagged data, we introduce an unforgeable header, which TagRoot can check on every `resume` call.

C. Unforgeable Headers

Trusted metadata such as ECBs or interrupt frames are protected via unforgeable headers (see Figure 2). To make headers unforgeable, they are tagged with TC-tag which only TagRoot can set. ECB headers and interrupt frame headers contain two distinct magic values which TagRoot can use to identify valid ECBs and valid interrupt frames. TagRoot takes care not to accidentally set the TC-tag on any other data containing these magic values. Since headers are callable via TC-tag, they could be misused as malicious entry points. To prevent this, the magic values have to fulfill the following property: When interpreted as assembler instruction, they shall divert control flow to some form of secure error handling (e.g., an endless loop "`j .`" or a jump to an error handler).

VII. TIMBER-V Implementation Details

We implemented TIMBER-V on the RISC-V Spike simulator and used it to run our TagRoot implementation. Subsequently, we give more details about tag isolation and the disambiguation of TUenter and TSenter, our tag-aware instructions, the proposed code transformations, required efforts for enclave developers, our MPU design and additional CPU registers.

Tag Isolation Policy. Our tag isolation policy is given in Table III. N-domains can only access N-tagged memory. The only way to enter T-domains is by fetching code tagged with TC-tag. Depending on the current privilege mode, TIMBER-V performs a TUenter or a TSenter. When fetching N-tagged memory, the CPU leaves trusted execution and switches back to the N-domains. This is denoted as TUleave and TSleave. Enclaves in TU-mode cannot write TC-tags to prevent manipulation of secure entry points. TS-tagged memory is exclusive to TS-mode and protects trusted metadata against malicious enclaves and the operating system. For security reasons, we also prevent TS-mode from fetching TU-tagged memory. This technique is well known and implemented as supervisor mode execution prevention (SMEP) in Intel x86 CPUs [29], for example. M-mode has full access to all tags, as it is commonly used to emulate missing hardware features.

TUenter vs TSenter Disambiguation. Both TU-mode and TS-mode use the same TC-tag to specify secure entry points. If not cleanly separated, this would allow confusion attacks between TUenter and TSenter. For example, an attacker could spawn a malicious enclave (TU-mode). While this malicious enclave normally cannot access other benign enclaves, the attacker could invoke the enclave via a TSenter from S-mode rather than a TUenter from U-mode. Hence, the malicious enclave would execute in higher-privileged TS-mode, thus undermining all of TagRoot's security guarantees. We prevent such attacks by constraining horizontal transitions to MPU regions of the same privilege mode: TUenter is only allowed for user mode MPU slots, while TSenter can only target MPU slots marked for TS-mode. TS-mode slots cannot be manipulated from the untrusted OS. Again, this resembles supervisor mode execution prevention [29].

TABLE III: Tag isolation policy for the memory accesses read (`r`), write (`w`), fetch or execute (`x`) as well as the horizontal transitions TUenter/TSenter (`e`) and TUleave/TSleave (`l`).

Access permitted	N-tag	TC-tag	TU-tag	TS-tag
N-domains	`rwx`	`--e`	`---`	`---`
TU-mode	`rwl`	`r-x`	`rwx`	`---`
TS-mode	`rwl`	`rwx`	`rw-`	`rwx`
M-mode	`rwx`	`rwx`	`rwx`	`rwx`

TABLE IV: TIMBER-V tag-aware instructions.

Checked Loads	Checked Stores
`lbct etag, dst, src`	`sbct etag, ntag, src, dst`
`lbuct etag, dst, src`	`shct etag, ntag, src, dst`
`lhct etag, dst, src`	`swct etag, ntag, src, dst`
`lhuct etag, dst, src`	Load Test Tag
`lwct etag, dst, src`	`ltt etag, dst, src`

MPU Design. Each MPU slot not only holds base and bound information together with rwx access permissions but also a TU and a TS flag. Slot marked as TU are shared between enclaves and untrusted apps. Slots marked as TS cannot be manipulated from untrusted code and are used to distinguish TSenter from TUenter, as outlined before. Only TagRoot can enable these flags. While the untrusted operating system cannot manipulate TS slots, it can overwrite TU slots, which will automatically clear the TU flag. This prevents enclave execution until TagRoot validates changes and reenables TU.

Tag-aware Instructions. We add new instructions for checking and manipulating tags, as listed in Table IV. We duplicate existing RISC-V load and store instructions to checked variants with the suffix `ct`. Checked loads preserve semantics of loading memory from address `src` into the register `dst`. Likewise, checked stores transfer the content of the `src` register to the memory address `dst`. Unlike normal memory accesses, the checked instructions trigger a trap if the memory tag of the memory address being accessed does not match the expected tag, encoded in `etag`. In addition, checked stores overwrite the tag at `dst` with a new tag, encoded in `ntag`. The accessed memory address is determined by a base address, stored in a register, and a 12-bit signed address offset, encoded as immediate. Also, `etag` and `ntag` are encoded as immediate, stripping the upper bits of the address offset to 10 bits and 8 bits for checked loads and checked stores, respectively.

For cases where memory tags are unknown, we add a separate load and test tag (LTT) instruction.[4] Similar to a checked load, LTT verifies the tag of a memory location

[4]Cf. the Test Target (TT) instruction of TrustZone-M [3].

```
1  lw    t0,24(sp)
2
3  lw    t1,2048(a1)
4
5  add   t0,t0,t1
6  sw    t0,24(sp)
```

(a) Original code.

```
1  lwct  ts,t0,24(sp)
2  addi  a1,a1,2040
3  lwct  ts,t1,8(a1)
4  addi  a1,a1,-2040
5  add   t0,t0,t1
6  swct  ts,ts,t0,24(sp)
```

(b) Transformed code.

Fig. 4: Code hardening for TS-mode with overflow correction.

```
1  function:
2    addi sp,sp,-8
3
4
5    ...
6
7
8    addi sp,sp,8
9    ret
```

(a) Original code.

```
1  function:
2    addi sp,sp,-8
3    swct n,ts,zero,4(sp)
4    swct n,ts,zero,0(sp)
5    ...
6    swct ts,n,zero,4(sp)
7    swct ts,n,zero,0(sp)
8    addi sp,sp,8
9    ret
```

(b) Transformed code.

Fig. 5: Stack interleaving for TS-mode.

(src) against a given expected tag (etag). However, instead of trapping, LTT stores the result in a register (dst), thus allowing subsequent code to take appropriate action. We utilize LTT for enclave cleanup, since this discharges TagRoot from keeping track of the exact enclave layout.

Code Transformations. We implement code transformations in a separate compilation step, where we compile source code to assembler code which we then transform using a custom awk script [24]. The code hardening transformation simply replaces all memory accesses with their checked instruction pedants, as shown for TS-mode in Figure 4. In some cases address overflows occur, namely when the encoding space of memory addresses is insufficient for a direct 1:1 transformation due to the additional etag and ntag encoding. In these cases, we insert correcting instructions which shift the overflowing part to the instruction's base register (lines 2–4). For stack interleaving the script detects stack allocations and deallocations by searching for manipulations of the stack pointer sp. It then claims or unclaims the stack frame by inserting checked store instructions accordingly, as seen in Figure 5 lines 3–4 and 6–7.

Developer Effort. From a developer's perspective, writing enclaves boils down to placing memory into distinct linker sections, for which we provide macros. One can mix enclave and non-enclave code in the same source file via annotations. Entry points are specified via a simple array. Ocalls in addition require to invoke an assembler macro. Code transformations are fully integrated in the macros and the build system. For memory accesses across security domains we provide dedicated macros setting etag accordingly. Edge routines could further reduce efforts, as done in the SGX SDK [30].

Additional CPU registers. We add new control and status registers (CSRs) for TS-mode (and M-mode). STSTATUS configures TIMBER-V and controls enclave execution. It holds a flag indicating the current security mode (normal or trusted). Moreover, whenever a running enclave traps due to an interrupt or exception, STSTATUS will raise a flag that prevents enclave execution until resumed by TS-mode. To allow TS-mode to intercept traps, we add a separate trap vector, called STTVEC. Whenever the CPU is in trusted mode, traps are redirected to a trusted trap handler pointed to by STTVEC. Traps happening in normal mode are forwarded to the standard trap handler, stored in STVEC. This is implemented in a small M-mode trap delegation code. To help the trusted trap handler in setting up scratch space, we duplicate the supervisor scratch register for the trusted mode, called STSCRATCH. In addition, we add a register denoted as SECB to hold a pointer to an enclave control block, which identifies the currently loaded enclave. This helps TS-mode in processing trusted enclave service calls.

VIII. Security Analysis

Shielded execution systems like TIMBER-V build upon various components to protect sensitive data from being leaked (confidentiality) or corrupted (integrity). In the following, we discuss how TIMBER-V protects enclaves against direct and indirect accesses. Furthermore, we discuss security of enclave shared memory, TagRoot and dynamic memory interleaving.

Direct Access. During runtime, the tag isolation policy prevents N-domains from directly accessing or tampering enclave memory. Also, our tag update policy does not allow elevating the current privilege mode. To prevent (malicious) enclaves from accessing other enclave's memory, TagRoot ensures that (i) enclave regions do not overlap upon enclave initialization, and (ii) the MPU only holds regions of a single enclave at a time. (i) ensures exclusiveness, *i.e.*, the only way for having enclave regions overlap is via shared memory, as discussed later. (ii) ensures that enclaves cannot misuse stale MPU entries of other enclaves. Also, our shared MPU design prevents forging of MPU entries. Whenever the untrusted operating system updates an MPU slot, an enclave cannot use it until TagRoot acknowledges these changes (cf. Section VII).

Indirect Access. Indirect security violations are prevented by (i) load-time attestation, (ii) secure entry points and (iii) secure interruption. During enclave loading, the operating system could manipulate an enclave's code to divulge secret information later on. To prevent this, enclave loading is measured using a cryptographically strong hash function (SHA256). Thus, whenever the untrusted operating system manipulates the loading procedure, this will yield a different enclave identity (EID) leading to different cryptographic keys, and subsequent attestation or unsealing of secrets will fail. To prevent direct code-reuse attacks from leaking sensitive enclave data to an attacker, TIMBER-V enforces secure entry points via the TC-tag. Since TC-tag can only be set by TagRoot, they are tamper-proof. Of course, this does not prevent code-reuse attacks in case of memory safety vulnerabilities in the enclave code itself. Achieving memory safety is an ongoing field of research [16]. If memory safety cannot be guaranteed, our code hardening transformation can make potential code-reuse attacks harder by preventing the attacker from misusing memory instructions to leak sensitive information. Finally, indirect information leakage due to enclave interruption is prevented by TagRoot, which clears sensitive register content before giving control to the operating system.

Shared Memory. In general, enclave regions cannot be modified during runtime except for shared memory, where enclaves willingly accept memory region overlaps with other enclaves. Since this process involves mutual authentication, it cannot be

misused to open bogus shared memory. Shared memory (shm) also demands temporal isolation to prevent time-of-check vs time-of-use (TOCTOU) attacks in two directions. First, if a shm-offering enclave gets destroyed, a target enclave still has access to the shm. As long as the target enclave does not release it, the shm cannot be given to a newly created offering enclave because TagRoot prevents enclave region overlaps. Thus, TagRoot supports temporal authenticity of the offering enclave. Second, if the target enclave gets destroyed after having accepted a shm offer, it might get reinstantiated and accept the same shm offer again without the knowledge of the offering enclave. This allows TOCTOU attacks. To avoid this, the offering enclave needs to close the shm offer after being accepted and employ a simple handshake to verify aliveness of the target enclave. For example, both enclaves could agree on a session identifier that changes for each enclave restart.

TagRoot. All of the aforementioned analysis critically depends on the integrity of TagRoot. We assume loading of TagRoot itself is protected using secure boot [42]. Once loaded, TagRoot can protect itself in an isolated execution container similar to enclaves by using tag isolation via TS-tag and secure entry points protected via TC-tag together with TS-mode MPU slots.

Dynamic Memory Interleaving. Here, untrusted code offers N-tag memory to trusted code. To be secure, untrusted arguments need to be validated by trusted code. In particular, one needs to ensure (i) validity of the memory when claiming it, and (ii) validity during usage. By claiming dynamic memory with checked store instructions (`etag` = N-tag) one can ensure (i), namely that trusted code does not accidentally overwrite trusted data in case of bogus memory pointers, for example. In addition, vertical stack interleaving crosses privilege modes and, thus, requires additional enclave region checks, as explained in Section VI-B. Point (ii) is different for the various interleaving schemes we presented before. In general, whenever pointers to trusted memory objects can be manipulated by untrusted code, one needs means to validate them. For supervisor heap and stack interleaving we introduced unforgeable headers, uniquely identifying valid ECBs and interrupt frames. This voids the need for tracking valid objects. In contrast, for user heap interleaving we recommended to track pointers to trusted heap objects inside the enclave. Also, horizontal user stack interleaving with ocalls needs additional checks of the stack pointer `sp` when re-entering the enclave. Here, we store the last valid stack pointer inside the enclave. By maintaining (i) and (ii), dynamic memory interleaving is secure against corruption and direct information leakage.

IX. Evaluation

A. Methodology

We evaluate TIMBER-V by running various macro- and micro-benchmarks in the Spike simulator, which we extended to support TIMBER-V. We configure Spike for the `RV32IMAFD` ISA and use it to record histograms of all executed instructions. To estimate the runtime in CPU cycles, we map executed instructions to actual CPU cycles using different pipelined CPU models. We first define a simple baseline model, against which we then compare two possible realizations of TIMBER-V, namely TIMBER-V Model A, capturing unoptimized implementations, and TIMBER-V Model B, representing optimized designs with tag caching. It should be noted that Model B is by no means an upper bound on the maximum performance achievable. Rather, it presents a conservative performance estimate based on related work about tagged memory architectures [31,45,48]. We outline these CPU models in the following and summarize them in Table V.

TABLE V: Expected CPU cycles per instruction.

CPU model	ld	st	lct	sct	reg	mul	div	other	stall
Baseline Model	1	1	-	-	1	1	1	1	3
TIMBER-V Model A	2	2	2	3	1	1	1	1	4
TIMBER-V Model B	1.1	1.1	1.1	1.1	1	1	1	1	3.1

Baseline CPU Model. As a baseline we assume that all register (`reg`) or memory instructions (`ld`/`st`) take one CPU cycle. This is reasonable for a load/store architecture as RISC-V with on-chip SRAM commonly used for embedded processors. When instructions stall the execution pipeline we assume additional latency to refill the pipeline. This applies to conditionally taken branches for indirect jumps as well as to syscalls and returns and is indicated by the column `stall`. We assume that multiplication (`mul`) and division (`div`) instructions also complete within one CPU cycle, which keeps our evaluation results pessimistic. That is, comparing against this baseline will show higher overhead than observed in practice, where multiplication and division typically take multiple cycles.

TIMBER-V CPU Models. Each instruction fetch requires one additional tag fetch. For the unoptimized Model A, we assume that this tag fetch can be effectively hidden by the prefetcher. Thus, linear code fetches do not exhibit overhead and all non-memory instructions (`reg`, `mul`, `div` and `other`) take one cycle. However, when the execution pipeline stalls, the tag fetching overhead gets visible for the first instruction after the stall. Thus, we add one extra cycle for stalls. For memory loads and stores we assume one extra cycle to load and check the tag on the accessed data. A checked memory load (`lct`) does not experience additional overhead since the data's memory tag is already loaded for enforcement of the tag isolation policy and can be readily used for the additional tag check. On the other hand, for checked memory stores (`sct`) we assume one additional cycle to store the new tag. This model does not make use of tag caching, which could significantly improve performance. A tag cache can serve tags in parallel to ordinary memory accesses and thus, hide the tag checking latency for all cached tags. By comparing state-of-the-art literature on tagged memory architectures, we observe that tag caching can reduce the average overhead of tag accesses into the low single digit range [31,45,48]. Considering that our work utilizes two tag bits per word, we conservatively estimate the expected performance impact of the tag operations with 10%, which is reflected in Model B. The resulting costs for the individual instruction classes is depicted in the last line of Table V. Again, the prefetcher hides tag checking latency for instructions, while a stall is prolonged by 10% of a cycle. Likewise, memory loads and stores experience 10% overhead. We assume that checked stores (`sct`) are not slower than ordinary stores (`st`) because the additional tag update latency can be absorbed by the parallel tag cache.

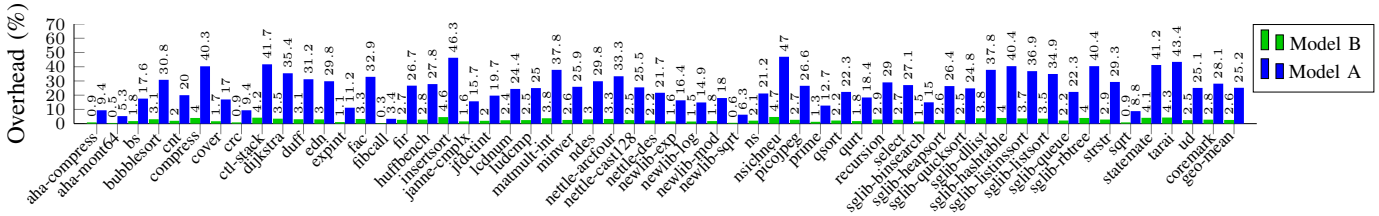


Fig. 6: Runtime overhead of TIMBER-V tag isolation for simple CPUs (Model A) and tag caching (Model B).

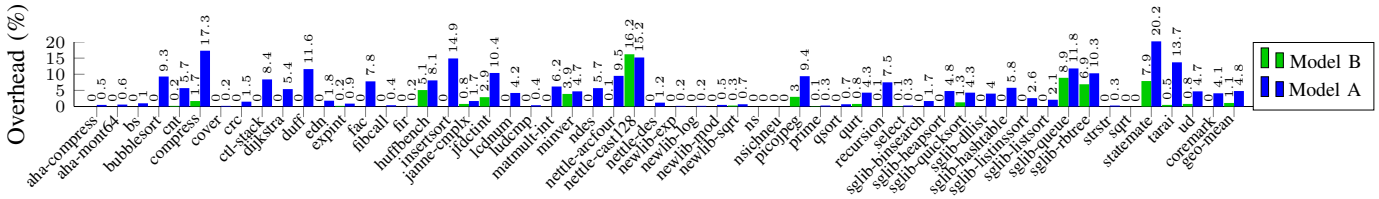


Fig. 7: Runtime overhead of the additional code hardening transformation.

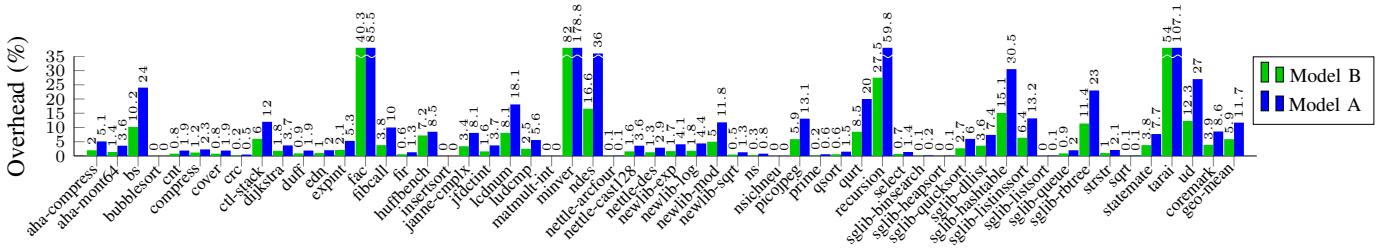


Fig. 8: Runtime overhead of the additional stack interleaving transformation.

B. Macrobenchmarks

To benchmark raw CPU performance, we used the beebs benchmark suite [4] as well as CoreMark [20]. We compiled them with GCC version 7.3.0 with "`-O1`". We excluded beebs benchmarks with external dependencies. Also, we filtered `nettle-md5` and `fdct` due to verification mismatches. For `newlib-log` and `ns` we had to prevent the compiler from optimizing out essential code by adding `volatile` and `noinline` statements. We ran beebs and CoreMark with one iteration since our evaluation does not need warm-up iterations to fill CPU caches but precisely captures all instructions.

Tag Isolation. Our tag isolation policy causes overhead of code execution for both, N-domains and T-domains. Figure 6 shows an average runtime overhead of 25.2% for TIMBER-V Model A with a peak of 47% for `nsichneu`, which uses frequent lookup table accesses. `insertsort` frequently swaps memory locations, which causes higher overhead. `statemate` implements a state machine with frequent state updates and `tarai` uses recursion, causing stack accesses to dominate over other operations. Interestingly, `aha-compress` shows significantly less overhead than `compress`, because it benchmarks four different CPU intensive compression algorithms with relatively few memory accesses. The `fibcall` benchmark shows least runtime overhead (3.4%) because the recursive Fibonacci computation can be kept entirely within the CPU registers. For the optimized TIMBER-V Model B, the average overhead is as little as 2.6% with a minimum of 0.3% for `fibcall` and a maximum of 4.7% for `nsichneu`. Our results indicate that even for memory intensive benchmarks Model B incurs small runtime overhead.

Code Hardening Transformation. Our code hardening transformation adds only negligible overhead, as shown in Figure 7. This is because the checked instructions are almost a drop-in replacement for ordinary memory instructions. Since ordinary memory instructions are subject to tag isolation causing memory tags to be loaded from memory (this overhead is included in Figure 6), the additional tag checks of checked instructions do not incur additional performance penalty. Few benchmarks show noticeable overhead because the code hardening transformation in some cases inserts correcting instructions to handle address overflows, as discussed in Section VII. By integrating the transformation directly into the compiler, one could leverage compiler optimization to avoid overflow behavior.

Stack Interleaving. To benchmark the additional overhead induced by stack interleaving, we compare each TIMBER-V model without stack interleaving against a compilation with enabled stack interleaving. The results are shown in Figure 8. The overhead is highly dependent on good compiler optimization and the used stack space. Many benchmarks (e.g., the memory-intensive `nsichneu`) show zero overhead for stack interleaving, since stack frames are optimized out in favor of CPU registers. The highest overhead (178.8%) occurs for `minver`, which allocates a temporary stack buffer of 500 words for computing matrix inverses. The average runtime overhead of stack interleaving is acceptable with 11.7% for Model A and 5.9% for Model B. Yet, we see potential for improvements in several directions: First, large stack allocations should be avoided. This is bad practice anyway since there exists no generic way of handling out-of-memory behavior on stack allocations. We manually adapted `minver` to preallocate a large stack buffer in the data segment and observed

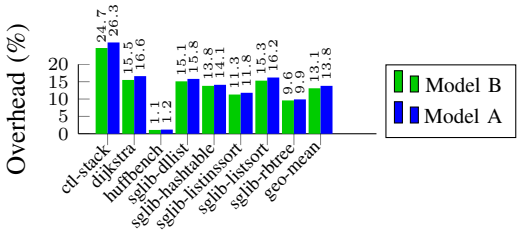


Fig. 9: Runtime overhead of additional heap interleaving.

that the runtime overhead drops from 82% and 178.8% to negligible 2.5% and 5.6% for Model B and Model A, respectively. Second, since stack interleaving implicitly erases new stack frames, one can avoid potential double clearing. We evaluated this for `huffbench` by manually removing the calls to `memset` on stack buffers. This reduces overhead from 7.2% and 8.5% down to 3.9% and 5% for Model B and Model A, respectively. This task could be automated by a compiler. Third, one could optimize frequent stack frame allocation and deallocation in favor of less frequent pre-allocation. For example, when having frequent calls to the same subfunction inside a loop, one could pre-allocate the subfunction's stack frame at the call site, thus reducing the stack interleaving overhead from N loop iterations to one.

Heap Interleaving. For heap interleaving we only evaluate benchmarks that use heaps. We use a simple heap implementation provided with FreeRTOS, namely `heap-4`. For heap interleaving we wrapped (de)allocation routines to claim and unclaim allocated memory using checked store instructions. The runtime overhead of heap interleaving is slightly below 14%, as shown in Figure 9, which is comparable to stack interleaving. We believe further improvements are possible since our `realloc` wrappers currently do not reuse existing allocations but always request new memory with `malloc`. The `huffbench` test shows negligible overhead because it allocates only one out of many buffers on the heap. Our secure `malloc` wrapper acts like `calloc`, clearing the whole buffer while changing tags. Likewise, our secure `free` automatically erases all data, while restoring the original tags. Thus, for security critical code that demands such zeroing functionality anyway, heap interleaving comes virtually for free.

C. Microbenchmarks

In the following, we discuss performance of trusted services as well as horizontal transitions between apps and enclaves. The performance numbers are summarized in Table VI. We depict SHA256 hashing costs in a separate column.

Trusted OS services. Trusted OS services are invoked like ordinary functions, hence, the transition denoted as TSenter has minimal overhead. When returning from a TSenter via TSleave, all callee-saved registers are cleared to avoid information leakage. The following results show the performance of the individual trusted OS service calls *without* TSenter and Tleave overhead. The base cost of `create-enclave` is dominated by claiming the ECB. We show the runtime when creating the first enclave. The runtime slightly increases when adding more enclaves since we chain all ECB's in a linked list. For `add-region` we show the runtime for adding the first region. The runtime grows with the number of regions as well as the number of enclaves due to the region overlap

TABLE VI: Enclave performance in expected CPU cycles.

	Functionality	TIMBER-V Model B		TIMBER-V Model A	
		Base cost	Hash cost	Base cost	Hash cost
Trusted OS Services	TSenter	7.1	0.0	9.0	0.0
	TSleave	27.4	0.0	32.0	0.0
	create-enclave	527.5	5647.1	759.0	7175.0
	add-region	396.3	5821.6	606.0	7483.0
	add-data	212.0	11309.4	340.0	14365.0
	add-entries	206.4	5616.2	348.0	7127.0
	init-enclave	123.5	5236.6	208.0	6397.0
	load-enclave	315.6	0.0	437.0	0.0
	destroy-enclave	733.5	0.0	1057.0	0.0
Trusted Encl. Services	TSyscall	68.3	0.0	71.0	0.0
	TSyscall dispatch	71.1	0.0	88.0	0.0
	TSyscall return	49.2	0.0	66.0	0.0
	get-key	337.5	12216.6	457.0	15686.0
	shm-offer	1045.6	0.0	1560.0	0.0
	shm-accept	1455.0	0.0	2062.0	0.0
	shm-release	231.7	0.0	317.0	0.0
	interrupt-enclave	107.7	0.0	175.0	0.0
	resume-enclave	103.0	0.0	200.0	0.0
App-Encl.	TUenter	1.0	0.0	1.0	0.0
	TUleave	4.1	0.0	5.0	0.0
	ocall	16.9	0.0	29.0	0.0
	ocall return	28.4	0.0	45.0	0.0

checks `add-region` performs. This variability is acceptable since overlap checks are cheap. Also, overlap checks are only performed at enclave initialization but not during runtime. For `add-data` and `add-entries` runtime increases with the size of the added data blob or the number of added entries, respectively. This is because changing memory tags as well as computing the hash measurement depends on the amount of data. Also, performance slightly depends on the position of the associated enclave region in the ECB. The base costs are shown in Table VI when adding one data word or one entry to the first enclave region. `init-enclave` has constant overhead. In contrast, `destroy-enclave` unclaims all enclave memory with a linear sweep over the enclave. We show the runtime of destroying an empty enclave. `load-enclave` validates the MPU configuration against the loaded enclave's ECB, hence the moderate overhead. Once an enclave is loaded, horizontal transitions between U-mode (app) and TU-mode (enclave) experience no principled overhead, as discussed later.

Trusted enclave services. Trusted enclave services are implemented as TSyscalls, which experience slight overhead due to M-mode trap delegation. TSyscall dispatching includes validation of the MPU configuration for vertical stack interleaving and jumping to the correct service routine. A return from a TSyscall unwinds the dispatcher context, clears all caller-saved registers and returns back to TU-mode using the RISC-V `sret` instruction. In the following we exclude TSyscall, dispatch and return overhead. `get-key` computes an HMAC using two SHA256 computations, hence the overhead. `shm-offer` needs to check validity of the arguments—not only their memory tags but also whether the arguments belong to the calling enclave. Apart from that, the performance is constant and independent of other enclaves. `shm-accept` traverses the linked list of ECB's to find a matching SHM offer. For our benchmarks, the first enclave in the linked list has a corresponding SHM offer. `shm-release` only erases the accepted SHM region from the enclave's ECB, in our case, the fifth enclave region. Interruption and resumption of enclaves (and TagRoot) is quite fast and mainly consists of saving and restoring the execution context in the interrupt frame. As before, the performance numbers of `interrupt-enclave`

and `resume` exclude TSyscall latency due to trap delegation, while TSyscall dispatch and return overhead do not apply here.

TUenter and TUleave. As shown in the last rows of Table VI, TUenter has no overhead, showing only one jump instruction into the enclave. TUleave only takes longer because of an assumed pipeline stall of the `ret` instruction. When enclaves call untrusted functions on the outside, these ocalls need to securely store and verify the stack pointer, as discussed in Section VI-B. Moreover, an enclave must clear sensitive CPU registers on TUleave as well as ocalls, which can be automated, e.g., via so-called edge routines in the SGX SDK [30].

D. Memory Overhead

TIMBER-V adds two tag bits to each 32-bit memory word, thus introducing 6,25% hardware memory overhead. Our TIMBER-V architecture directly runs unmodified code and, thus, does not introduce software memory overhead. Likewise, our code hardening transformation does not introduce memory overhead, since memory instructions are replaced 1:1 with checked instructions. Slight overhead only occurs if additional instructions are inserted for fixing offset overflows, as discussed in Section IX-B. Heap interleaving needs small constant-sized code memory for the allocation hooks but in turn voids the need for secure heap implementations, which in total reduces code size. We do not give actual numbers since this strongly depends on the heap implementation. Stack interleaving needs additional code for stack frame allocation and deallocation. Currently, we insert checked store instructions for each allocated word, thus showing 43% overhead in assembler code lines for the expensive `minver` benchmark. However, when optimizing for code size, one could easily achieve constant overhead per stack (de)allocation by embedding checked stores in a loop. We manually optimized stack interleaving for `minver` and reduced the code overhead to 1%.

TagRoot Code Size. We used `sloccount` to count the number of source code lines as an estimate of TagRoot's complexity. TagRoot consists of 369 lines of assembler code and 1686 lines of C-code, from which 313 lines are used by HMAC and SHA256. This code base is fairly small, which is desirable for a trusted computing base as it reduces the risk of programming bugs. Also, the small size is beneficial for formal verification techniques that could help certify our TagRoot implementation [33]. As a comparison, the used FreeRTOS operating system has approximately 12 500 lines of code.

X. Related Work

In this section, we compare TIMBER-V against related work on isolated execution as well as tagged memory.

A. Isolated Execution

Hardware-based isolated execution can be classified into virtual and physical address-based systems, of which Maene et al. [36] give an extensive overview. Many schemes target mid and upper-class processors with virtual memory support, among which are AEGIS [47], Intel TXT [28], ARM TrustZone [2], Bastion [10], IBM SecureBlue++ [7], Intel SGX [37], ISO-X [22]. Sanctum [13] implements the SGX enclave model on RISC-V with virtual memory, adding additional side-channel protection. In contrast, we bring enclaves to smaller RISC-V featuring only limited physical memory.

Physical Address-based Systems. SMART [21], Sancus [40], Soteria [25], TyTAN [9], and TrustLite [34] implement program counter-based memory access control for isolating secure tasks. Secure task's memory regions are only accessible when the program counter is in its code region. Sancus has a hardware-only TCB and isolates a fixed number of small uninterruptible secure tasks stored in pre-defined memory locations. TyTAN and TrustLite use an execution aware MPU (EA-MPU) with multiple code and/or data regions per secure task. TrustLite loads all secure tasks at boot time, while TyTAN allows dynamic loading and unloading of secure tasks at runtime. The EA-MPU makes context switches faster but limits the number of concurrently loaded secure tasks. In contrast, TIMBER-V supports an arbitrary number of enclaves with fine-grained, dynamic isolation and multiple entry points.

Secure communication in TrustLite is done via a simple handshake protocol, where two secure tasks first attest each other and then use cryptographic session tokens to authenticate messages. In TIMBER-V local enclave attestation and communication is done implicitly via shared memory, without using any cryptographic secrets. TyTAN uses a dedicated IPC proxy task which forwards messages between secure tasks, introducing copying overhead (1324 CPU cycles). In contrast, our secure shared memory is a fast alternative for exchanging bulk data between enclaves.

TrustZone-M [3] supports four security domains like TIMBER-V. Horizontal and vertical domain transitions require special instructions, while in TIMBER-V domain switches are direct, thus imposing zero runtime overhead. TrustZone-M only supports secure and non-secure tasks, while our architecture supports mixed processes, where enclaves are directly embedded in untrusted processes via tagged memory, thus achieving fine-grained isolation. TrustZone-M optionally supports two separate MPUs, one for the secure and one for the non-secure world. We reuse the same MPU across security domains, thus saving hardware costs. Also, our dynamic memory interleaving allows for stack (and heap) reuse, while TrustZone-M requires separate stacks for each domain.

B. Tagged Memory Architectures

The availability of metadata is the foundation for a multitude of run-time monitoring techniques like various sanitizers [44,46], as well as dynamic information flow tracking (DIFT) (a.k.a. taint tracking) [43]. Subsequently, many hardware-based tagged memory architectures have been developed. In particular, for DIFT, implementations range from single tag bit schemes with fixed policy (e.g., Minos [14] and CHERI [52]), over multi-bit schemes with partially configurable policy (e.g., Raksha [15], DIFT [48], DIFT with coprocessor [32]), to schemes with configurable bit width and fully programmable policy and enforcement (e.g., FlexiTaint [49], instruction-grain lifeguards [11], Harmoni [17], PUMP [19]).

Compared to DIFT architectures, TIMBER-V has notably different characteristics. Firstly, DIFT schemes have a strong focus on performing tag/taint propagation during ALU operations. TIMBER-V, on the other hand, does not perform any tag

propagation but utilizes tags for isolation purposes. Abusing a DIFT architecture solely for isolation, while possible in some schemes like Raksha [15] and PUMP [19], is needlessly wasteful. Secondly, TIMBER-V introduces a new trusted security domain, and the isolation and update policies depend on the currently active domain. Partially configurable DIFT architectures typically do not support such a domain switch. Finally, even fully programmable DIFT architectures are not necessarily suited for implementing TIMBER-V. Namely, architectures that perform tag operations asynchronously to the main processor [11,17,32] introduce a TOCTOU gap that can potentially be used to exfiltrate data from the trusted domain.

Besides DIFT-based architectures, other architectures use tagged-memory for enforcing various kinds of memory protection. HardBound [18] implements fat pointers to prevent spatial memory safety violations. HDFI [45] uses a single tag bit to protect sensitive data words. However, in HDFI, tag checks are only performed when reading the data which means that destructive write operations on sensitive data can not be prevented but only detected. This property corresponds to the weak *low-watermark policy for objects* of the Biba integrity model [6]. In contrast, TIMBER-V follows the stronger strict integrity policy of the Biba model by refusing untrusted modifications of trusted data. Compared to that, Mondrian Memory Protection [53], which uses two tag bits, and Loki [54], using up to 32 tag bits per word, are more similar to TIMBER-V. However, both concepts solely use tagged memory to implement word-wise access permissions which is not sufficient to implement efficient isolated execution. Additionally, when different permissions are tightly interleaved, Loki's tag size is simply too large for low-end devices that we target.

XI. Possible Extensions

The concept of TIMBER-V can be directly applied to other system components. Together with secure interrupts, flexible safety-critical systems can be implemented.

Secure Components and Peripherals. One can easily extend CPU caches with our two tag bits and propagate them to main memory on cache eviction. Also, memory-mapped I/O peripherals can benefit from TIMBER-V's tag isolation policy by pinning their tag bits in a tag cache. That way, TIMBER-V can facilitate secure I/O, that is, secure interaction with end users, sensors, actuators or other networked devices.

Secure Interrupts. Most embedded systems react upon regular timer or irregular I/O interrupts. TIMBER-V supports secure interrupts by modifying the M-mode trap delegation mechanism to route interrupts directly to the trusted trap handler, which is *not* callable to prevent fake interrupts from S-mode.

Safety-critical Systems. TagRoot is a compact implementation of isolated execution on top of TIMBER-V. Extending TagRoot for safety-critical systems with availability guarantees is an interesting field of research and should be straight forward. We denote safety-critical enclaves as safeclaves. To guarantee real-time behavior, safeclaves must be protected against denial-of-service attacks (DoS). Safeclaves are not triggered by untrusted code but by external I/O events or recurring timer periods. TagRoot can intercept safeclave interrupts as discussed before in order to assuredly trigger safeclave execution. Obviously, one cannot use dynamic memory interleaving for safeclaves. Normal enclaves, however, can still benefit from interleaving. Also, by slightly adapting our shared MPU design, one can exclude safeclave MPU slots from being shared, making safeclaves safe against DoS from the OS.

XII. Conclusion

We presented TIMBER-V, the first efficient tagged memory architecture for isolated execution of enclaves. TIMBER-V minimizes memory overhead of tagged memory by augmenting tag isolation with MPU isolation. The flexibility of TIMBER-V enables fine-grained and dynamic management of trusted memory, enabling novel schemes like stack interleaving. This reduces memory fragmentation, which is particularly relevant for low-end devices. A small trust manager provides trusted services, including secure shared memory. We implemented and evaluated TIMBER-V to demonstrate its practicality.

Acknowledgments

This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments" and by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Furthermore, this research was co-funded by the German Science Foundation, as part of project S2 and P3 within CRC 1119 CROSSING, and Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

References

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing, 2013. White Paper.

[2] ARM Security Technology: Building a Secure System using TrustZone Technology, 2009. Ref. no. PRD29-GENC-009492C.

[3] TrustZone technology for ARMv8-M Architecture, 2017. Ref. no. 100690_0200_00_en.

[4] J. Bennett, A. Burgess, S. Cook, K. Eder, S. Hollis, and J. Pallister. Bristol/embecosm embedded benchmark suite. http://beebs.eu/ (Accessed 2018/06/18).

[5] D. J. Bernstein. Cache-Timing Attacks on AES, 2005. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. (Accessed 2018/05/29).

[6] K. J. Biba. Integrity Considerations for Secure Computer Systems, 1977. The MITRE Corporation. Tech. Report ESD-TR-76-372.

[7] R. Boivie and P. Williams. SecureBlue++: CPU Support for Secure Executables, 2012. IBM research report no. RC25369.

[8] A. Bradbury, G. Ferris, and R. Mullins. Tagged memory and minion cores in the lowRISC SoC, 2014. lowRISC-MEMO 2014-001.

[9] F. F. Brasser, B. E. Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: tiny trust anchor for tiny devices. In *Design Automation Conference – DAC'15*, pages 34:1–34:6. ACM, 2015.

[10] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *High Performance Computer Architecture – HPCA'10*, pages 1–12. IEEE Computer Society, 2010.

[11] S. Chen, M. Kozuch, P. B. Gibbons, M. P. Ryan, T. Strigkos, T. C. Mowry, O. Ruwase, E. Vlachos, B. Falsafi, and V. Ramachandran. Flexible Hardware Acceleration for Instruction-Grain Lifeguards. *IEEE Micro*, 29:62–72, 2009.

[12] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Security and Privacy – S&P'09*, pages 45–60. IEEE Computer Society, 2009.

[13] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security'16*, pages 857–874. USENIX Association, 2016.

[14] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *TACO*, 3:359–389, 2006.

[15] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *International Symposium on Computer Architecture – ISCA'07*, pages 482–493. ACM, 2007.

[16] A. A. de Amorim, C. Hritcu, and B. C. Pierce. The Meaning of Memory Safety. In *Principles of Security and Trust – POST'18*, volume 10804 of *LNCS*, pages 79–105. Springer, 2018.

[17] D. Y. Deng and G. E. Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Dependable Systems and Networks – DSN'12*, pages 1–12. IEEE Computer Society, 2012.

[18] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. In *Architectural Support for Programming Languages and Operating Systems – ASPLOS'08*, pages 103–114. ACM, 2008.

[19] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. K. Jr., B. C. Pierce, and A. DeHon. Architectural Support for Software-Defined Metadata Processing. In *Architectural Support for Programming Languages and Operating Systems – ASPLOS'15*, pages 487–502. ACM, 2015.

[20] EEMBC. CoreMark. https://www.eembc.org/coremark/ (Accessed 2018/06/18).

[21] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *Network and Distributed System Security Symposium – NDSS'12*. The Internet Society, 2012.

[22] D. Evtyushkin, J. Elwell, M. Ozsoy, D. V. Ponomarev, N. B. Abu-Ghazaleh, and R. Riley. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *Symposium on Microarchitecture – MICRO'14*, pages 190–202. IEEE Computer Society, 2014.

[23] E. A. Feustel. The Rice research computer: a tagged architecture. In *American Federation of Information Processing Societies – AFIPS*, volume 40 of *AFIPS Conference Proceedings*, pages 369–377. AFIPS, 1972.

[24] The GNU Awk User's Guide. Edition 4.2. https://www.gnu.org/software/gawk/manual/gawk.html,(Accessed 2018/08/06).

[25] J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. C. Freiling, and I. Verbauwhede. Soteria: Offline Software Protection within Low-cost Embedded Devices. In *Annual Computer Security Applications Conference – ACSAC'15*, pages 241–250. ACM, 2015.

[26] Helpnetsecurity. The cost of IoT hacks: Up to 13% of revenue for smaller firms, 2017. https://www.helpnetsecurity.com/2017/06/05/iot-hacks-cost/ (Accessed 2018/07/27).

[27] B. Igal. Bits, please! exploring Qualcomm's TrustZone implementation, 2015. http://bits-please.blogspot.com/2015/08/exploring-qualcomms-trustzone.html (Accessed 2018/08/01).

[28] Intel Trusted Execution Technology (Intel TXT), Software Development Guide. Reference no. 315168-012.

[29] Intel 64 and IA-32 Architectures Software Developer's Manual, 2016. Reference no. 325462-061US.

[30] Intel Software Guard Extensions SDK for Linux OS. Developer Reference, 2016. Rev. 1.5.

[31] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. D. Son, and A. T. Markettos. Efficient Tagged Memory. In *International Conference on Computer Design – ICCD'17*, pages 641–648. IEEE Computer Society, 2017.

[32] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *Dependable Systems and Networks – DSN'09*, pages 105–114. IEEE Computer Society, 2009.

[33] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53:107–115, 2010.

[34] P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan. TrustLite: a security architecture for tiny embedded devices. In *European Conference on Computer Systems – EUROSYS'14*, pages 10:1–10:14. ACM, 2014.

[35] S. Larson. FDA confirms that St. Jude's cardiac devices can be hacked. https://money.cnn.com/2017/01/09/technology/fda-st-jude-cardiac-hack/ (Accessed 2018/07/18).

[36] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. C. Freiling, and I. Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Trans. Computers*, 67:361–374, 2018.

[37] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Hardware and Architectural Support for Security and Privacy – HASP*, page 10. ACM, 2013.

[38] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle, 2015. http://illmatics.com/Remote%20Car%20Hacking.pdf (Accessed 2018/07/18).

[39] NJCCIC. Mirai Botnet. https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet (Accessed 2018/07/18).

[40] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. C. Freiling. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.*, 20:7:1–7:33, 2017.

[41] E. Ronen, A. Shamir, A. Weingarten, and C. O'Flynn. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *Security and Privacy – S&P'17*, pages 195–212. IEEE Computer Society, 2017.

[42] X. Ruan. *Boot with Integrity, or Don't Boot*, pages 143–163. Apress, Berkeley, CA, 2014.

[43] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Security and Privacy – S&P'10*, pages 317–331. IEEE Computer Society, 2010.

[44] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-Sanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference – USENIX ATC'12*, pages 309–318. USENIX Association, 2012.

[45] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *Security and Privacy – S&P'16*, pages 1–17. IEEE Computer Society, 2016.

[46] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Symposium on Code Generation and Optimization – CGO'15*, pages 46–55. IEEE Computer Society, 2015.

[47] G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *International Conference on Supercomputing – ICS'03*, pages 160–171. ACM, 2003.

[48] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Architectural Support for Programming Languages and Operating Systems – ASPLOS'04*, pages 85–96. ACM, 2004.

[49] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *High Performance Computer Architecture – HPCA'08*, pages 173–184. IEEE Computer Society, 2008.

[50] A. Waterman and K. Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.2. Technical report, SiFive Inc., EECS Department, University of California, Berkeley, 2017.

[51] A. Waterman and K. Asanović. The risc-v instruction set manual, volume ii: Privileged architecture, version 1.10. Technical report, SiFive Inc., EECS Department, University of California, Berkeley, 2017.

[52] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. D. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Security and Privacy – S&P'15*, pages 20–37. IEEE Computer Society, 2015.

[53] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Architectural Support for Programming Languages and Operating Systems – ASPLOS'02*, pages 304–316. ACM Press, 2002.

[54] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Operating Systems Design and Implementation – OSDI'08*, pages 225–240. USENIX Association, 2008.