On Real-Time Guarantees in Intel SGX and TDX

Peterson Yuhala
Computer science department,University of Neuchâtel, Switzerland

Christian Göttel
Corporate Research Center, ABB Schweiz AG, Baden-Dättwil, Switzerland

Jämes Ménétrey
Computer science department, University of Neuchâtel, Switzerland

Valerio Schiavoni
Computer science department, Universiy of Neuchâtel, Switzerland

David Kozhaya
Corporate Research Center, ABB Schweiz AG, Baden-Dättwil, Switzerland

Pascal Felber
Computer science department, University of Neuchâtel, Switzerland

Abstract

Trusted execution environments (TEE) represent a major technological breakthrough that provide strong confidentiality and integrity guarantees for code and data running on potentially vulnerable or untrustworthy computing systems, such as cloud, edge, embedded, mobile, or even blockchain systems. However, the performance overhead associated with TEEs still poses a limitation on the extent to which real-time (RT) sensitive applications can benefit from this technology, e.g., to run on untrusted third-party infrastructures. This work investigates various TEE-based architectures spanning from process-based to virtual-machine-based implementations, for securing RT applications. It offers in addition an in-depth evaluation of these architectures, providing insights into how various TEE deployments influence the temporal compute and communication guarantees of RT systems.

2012 ACM Subject Classification Security and privacy → Trusted computing; Computer systems organization → Real-time system architecture

Keywords and phrases Trusted execution environments, Real-time systems, Intel SGX, Intel TDX, WebAssembly

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2025.8

Supplementary Material *Software (Source Code)*: `https://github.com/Yuhala/intel-sgx-real-time-ecrts`, archived at `swh:1:dir:00367a4d3e6a3598bee0f2ffcefd7342c124e243`

1 Introduction

The ubiquity of computing, sensing, and communication devices has led various areas like industrial control, smart grids and sensor networks to increasingly rely on such devices to control and coordinate system operations. More specifically, the future electric grid, whose landscape is drastically changing with the proliferation of distributed energy resources, is forced to transition the use, management and control of energy to become more distributed, scalable and dynamic. However, the only non-changing constant in this transition is in the system's ability to be "always" available and maintain predictable low latency response times.

The current trend in standardization bodies and communities (e.g., E4S and SEAPATH [1, 4]) governing this transition promotes for open microservice-like designs capable of hosting multi-tenant/multi-vendor control and monitoring applications. Despite the advocated flexibility of such designs, one has to be careful not only about the varying loads of hosted applications and their need for resources, but also about the security challenges that might arise. We further elaborate on this in what follows.

On the one hand, the multi-tenant architecture like cloud platforms, where users share physical hardware, introduces security vulnerabilities as malicious or vulnerable tenants could breach data confidentiality and integrity, as well as intellectual property of code relevant to the protection, control and monitoring of applications. Collected data often includes detailed information about energy usage patterns at individual or business levels. Additionally, unauthorized access to application code could allow competitors or attackers to gain insights into the energy provider's operations or infrastructure, leading to competitive or security risks. On the other hand, the control and monitoring providers would have reduced trust in the hardware owner as well as privileged software like the operating system (OS) and hypervisor, as they have almost complete control over it. This situation underscores the need for techniques that enhance security guarantees and return control and monitoring capabilities to application providers.

Trusted execution environments (TEEs) are hardware-based security techniques that provide strong confidentiality and integrity guarantees to applications running on untrusted cloud, edge, embedded, or mobile platforms. TEE technologies provide different levels of isolation: either at the application or process level or at the virtual machine (VM) level. Popular TEE technologies that have been introduced in recent years include: Intel Software Guard Extensions (SGX) [18] and Arm TrustZone [51] for process-level isolation, and Intel Trust Domain Extensions (TDX) [17], AMD Secure Encrypted Virtualization (SEV) [23], and Arm Confidential Compute Architecture (CCA) [38] for VM-level isolation. Despite the strong security guarantees provided, TEEs introduce non-negligible overhead in applications, which could be critical when dealing with real-time (RT) systems where tasks must complete within specified deadlines. In that light, our work explores multiple TEE-based architectures for deploying RT applications using Intel SGX and TDX, emphasizing their implications on security, performance, and the temporal guarantees as required by RT applications. Our choice to focus on Intel-based TEEs in this paper is motivated by two main reasons highly correlated with current industrial offerings of RT applications: (1) Intel TDX being the only technology to-date that offers confidential computing on a virtual machine level, which is the virtual form in which some RT applications are sold and (2) the dependence of many of those RT application offerings on Intel hardware. Furthermore, dedicated Intel processors are today the only platform that is simultaneously offering application-level and virtual-machine-level TEEs on a single chip.

Most TEE technologies, including SGX, SEV, TDX, are built on threat models where system software runs on host systems controlled by an untrusted cloud service providers (CSPs) [9, 12, 13, 22, 45]. As a result, these threat models cannot inherently guarantee availability of the code executing in TEEs. For example, TEE technologies are vulnerable to denial-of-service (DoS) attacks by a malicious host OS or hypervisor. Furthermore, since the host OS and hypervisor are inherently untrusted in most TEE threat models, one cannot trust the correctness of the timing information from the underlying privileged software [8, 26, 48]. From a RT perspective, this implies that achieving strict RT guarantees is theoretically infeasible within popular TEEs like SGX or TDX. Nevertheless, in practical scenarios, it is often reasonable to relax these threat models to assume that an untrusted CSP has a strong incentive to maintain availability due to service level agreements with cloud clients [45], but may seek to access and learn sensitive client data or code. To this end, this paper focuses primarily on investigating the ability of various TEE-based architectures for offering industrial software to adhere to the predictability and timing guarantees necessary for the operation of RT systems. The investigation includes insights into the computing (timing and isolation) and the communication capabilities of those architectures. To the best of our knowledge, this work is the first to provide such an in-depth exploration. In summary, our work provides the following contributions:

- Multiple flavors for offering secure RT applications using Intel SGX and TDX.
- A thorough discussion on security, performance (compute and communication), and usability trade-offs of the proposed designs.
- An extensive performance evaluation of the proposed designs with popular RT benchmarking tools, including open-source contributions for execution in Intel SGX.

The rest of this paper is organized as follows. We provide background information on TEEs and RT systems in §2 as well as their threat model in §3. In §4 we describe different TEE-based architectures for securing RT applications. The performance evaluations of these designs are outlined in §5. We discuss the current state of confidential computing for RT systems and highlight various challenges in §6. In §7, we survey prior work, after which we conclude and discuss future work in §8.

2 Background

This section provides essential background information on TEEs and an overview on RT systems.

2.1 Trusted execution environment (TEE)

A TEE is a secure isolated environment provided by the processor to ensure confidentiality and integrity of code and data at runtime. The isolation provided by a TEE can either be at the *process-level*, where a single application is shielded from privileged software, or *virtual-machine-level*, where an entire virtual machine (VM) is shielded from an untrusted hypervisor. Popular processor platforms, such as Intel, Arm, AMD, RISC-V, have introduced TEE capabilities into their central processing unit (CPU) designs, such as Intel SGX [18], RISC-V physical memory protection (PMP) [62], and Arm TrustZone [51] for process-level isolation, or Intel TDX [17], AMD SEV [23], and Arm CCA [38] for VM-level isolation.

Trusted computing base. An important factor when building TEE-based applications is the size of the trusted computing base (TCB), which represents the set of hardware and software components that are critical to the system's security. In general, a large TCB is bad for security because, empirically, there is a strong correlation between the number of security vulnerabilities in a given code base and the size and complexity of the code base [46, 66]. As a result, minimizing the TCB of applications is central in TEE-based development. However, minimizing the TCB usually comes at a cost of increased complexity at the level of development and deployment. This introduces a security vs usability trade-off which should be carefully considered when adopting a specific security architecture.

Intel SGX. SGX is an extension to the x86 instruction set architecture (ISA) which enables user applications (i.e., processes) to create secure memory regions called *enclaves* which are inaccessible to privileged software like the OS or hypervisor. All enclave memory is backed by an encrypted region of physical memory called the enclave page cache (EPC) whose pages are transparently decrypted/encrypted when copied in/out of the CPU's cache lines respectively. Previous Intel SGX versions provided limited EPC memory, typically 128 MiB or 256 MiB. However, more recent versions provide larger EPC memory regions, up to 1 TiB [33], but this comes with weaker integrity guarantees for enclave memory pages [34, 39, 68].

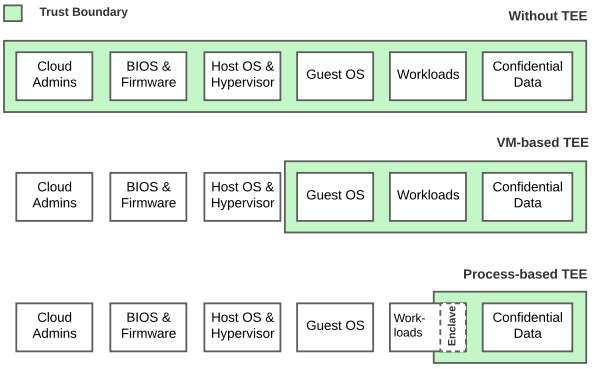

Figure 1 TCBs for different TEE designs.

Intel TDX. This is an architectural extension in the 4th and later Generation Intel Xeon Scalable processors that offers cryptographic isolation for VMs, called *trust domains*, in the presence of an untrusted (or adversarial) hypervisor or host OS. TDX guarantees confidentiality and integrity of the trust domain's memory pages and virtual CPU states, ensuring that they cannot be accessed or tampered with by other tenant VMs executing on the same machine, or a malicious cloud administrator [17]. Similar to SGX, TDX provides remote attestation possibilities to verify the software integrity in the VM, as well as the authenticity of the TDX-enabled platform.

2.2 Real-time system

A RT system is one where correctness not only depends on the logical results but also on the time at which the results are produced [55]. Essentially, a RT program must guarantee a response within a specified time constraint or *deadline* [64]. RT systems are generally classified into *hard* or *soft* RT systems, and are invoked either periodically or in response to interrupts. Hard RT systems have very strict deadlines which must be met, and missing the deadline could lead to severe consequences. Common examples include aircraft and medical monitoring systems. In soft RT systems, meeting the deadline is important for performance but not critical, and missing the deadline can generally be tolerated. Examples include multimedia applications like media streaming.

To ensure the operational deadlines of RT applications are met, the underlying OS must provide predictable scheduling and mechanisms to manage system resources effectively so as to minimize jitter, i.e., the delay between the time a task is expected to start, and the time when the task is actually started. The Linux kernel provides this possibility via the `PREEMPT_RT` patch [53] which converts Linux into a fully preemptible kernel by making sections of the latter preemptible [42].[1] As of Linux v6.12, the `PREEMPT_RT` branch was integrated into the mainline kernel. This permits the scheduler to preempt tasks in favor of a RT tasks, ensuring more predictable scheduling latencies for such tasks.

[1] While the set of `PREEMPT_RT` patches seek to minimize preemption latency, they are not necessarily sufficient for hard real-time applications. In those scenarios, real-time operating systems are typically employed.

3 Threat model

Our threat model considers different isolation designs: process-level isolation and VM-level isolation. We consider an adversary who controls the entire system software, including the OS and the hypervisor, and has physical access to all server hardware.

An adversary with physical access to the server can mount snooping attacks [37] on the host memory bus or is capable of setting and manipulating the system time to mess with the scheduler and execution of RT tasks. However, the CPU package and firmware is considered trusted, and we assume that the adversary cannot physically open the CPU package to extract secrets.[2] We assume that the software being secured, i.e., process or VM, is trusted and does not intentionally leak sensitive information. In the VM-based isolation design, this entails that the guest OS is equally trusted.

Availability. Most server-end TEE technologies, including Intel SGX and TDX, do not provide availability guarantees to code running in enclaves; consequently, availability guarantees are considered out-of-scope. This stems from the fact that privileged software like the host OS and hypervisor are under adversarial control [36] in a TEE threat model, and thus can (in theory) induce resource unavailability. As a result, a RT system secured in a TEE like Intel SGX or TDX must assume an adversary reluctant to provoke resource unavailability or denial of service (DoS). That is, the adversary, e.g., an untrusted CSP, is more interested in learning sensitive information than interrupting enclave execution. This assumption is reasonable in practice because CSPs operate under service level agreements (SLAs) with clients, giving them strong incentives to maintain system availability [45].

Nonetheless, client-side TEE technologies like Arm Trustzone provide mechanisms to isolate kernel-level components, including drivers and peripheral hardware, from unauthorized access by other privileged system software [61, 67]. This can improve availability guarantees as compared to the server-end TEEs like Intel SGX which lack isolation for kernel-level components.

Side channel attacks. Side-channel attacks, e.g., based on speculative execution [16, 35], memory access patterns [11, 65] and cache timing [31] for which mitigations exist [30, 40, 48] are considered out of scope.

4 TEE-based architectures for real-time applications

In this section, we explore different architectures for deploying a RT application in a TEE: (1) Intel SGX software development kit (SDK); (2) library operating system (libOS); (3) WebAssembly (Wasm); and (4) VM.

4.1 Intel SGX SDK-based design

A C/C++ based SGX SDK is provided by Intel to facilitate the development and deployment of Intel SGX applications. It requires the developer to manually partition their program into trusted and untrusted parts, as shown in Figure 2, which execute in or out of an enclave respectively. To allow communication between both parts, the Intel SGX SDK provides *ecalls* and *ocalls* which enable application threads to enter or exit enclave mode respectively. The program's address space is split into a trusted region – the enclave (accessible only to enclave code) – and the untrusted region (accessible to both the enclave and non-enclave

[2] This could involve using specialized equipment to extract data from on-chip memory.

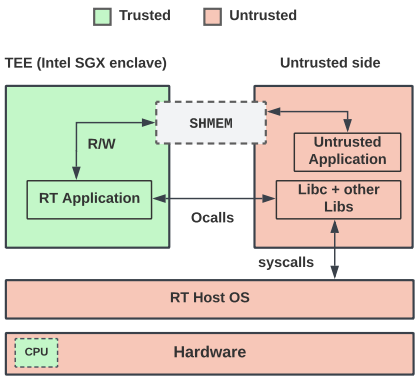

Figure 2 RT application deployed with SGX SDK.

code). Shared-memory between the enclave and non-enclave parts is typically implemented as a region of non-enclave memory used by both parts of the Intel SGX program. This architecture results in the smallest TCB, but requires significant engineering effort to refactor the program to comply with the limited in-enclave library support of the Intel SGX SDK. These refactoring operations typically involve reimplementing unsupported system calls/libc application programming interfaces (APIs), e.g., read, write, *etc.* as ocalls. Such calls may require expensive enclave transitions (up to 14 000 CPU cycles [63, 69]); such overheads can significantly affect RT applications. Several strategies have been proposed to mitigate these transitions by leveraging worker threads [58, 69]. Nevertheless, using such strategies may adversely impact latencies in the RT system since worker threads consume CPU resources as well. To prevent such scenarios, CPU isolation strategies [41] can be used to ensure critical RT threads have exclusive access to CPU resources.

Take-away 1 : *To ensure RT performance in enclave-deployed applications, the use of system calls and unsupported libc routines should be minimized or entirely avoided.*

4.2 LibOS-based design

Partitioning an application for Intel SGX can be very challenging, especially when dealing with complex applications written in unsupported programming languages. These difficulties around SDK-based development have led to the rise of library operating system (libOS)-based designs. In the context of TEEs, a libOS can be viewed as a lightweight OS abstraction designed to run entire applications inside enclaves. The libOS intercepts unsupported operations like system calls, and either emulates them or securely relays them to the underlying OS using mechanisms like remote procedure calls (RPCs) [60]. Popular examples of libOS technologies include Gramine [60], Occlum [54], Haven [9], *etc.* which allow legacy programs to be secured inside Intel SGX enclaves with minimal modifications. This architecture, depicted in Figure 3, enhances usability and enables easier deployment of legacy programs inside the TEE. However, this design leads to a larger TCB when compared to the partitioned approach, and hence larger attack surface due to the use of a libOS.

In the context of RT, [42] highlights special aspects of the Linux RT kernel which should be taken into account for optimal performance of RT applications. Some of these aspects which may be of importance in the context of TEEs and libOSes are: memory locking,

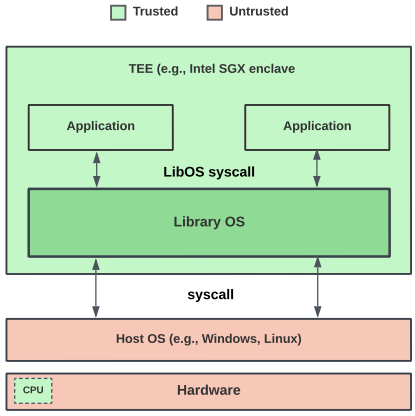

Figure 3 RT application deployed with a library OS.

page faults, and high-resolution timers. Memory locking ensures that the pages of the process are resident in memory, and the Linux kernel provides a the `mlock` system call to achieve this. However, these cannot be used for enclave/EPC pages; libOS designs should consider introducing some degree of support for these. Nevertheless, since enclaves store their data only in the EPC, external (i.e., untrusted) processes/threads in regular dynamic random-access memory (DRAM) will not contend for memory with enclaves.

Also, expensive page swapping operations of enclave pages into regular DRAM can significantly impact the latencies of RT tasks. However, more recent SGX-enabled platforms support enclave memory up to 1 TiB; this substantially minimizes the likelihood of such expensive paging operations.

Lastly, high-accuracy timers are essential to ensure that periodic operations step through time correctly [42]. At the time of this writing, SGX enclaves still rely on the underlying OS to provide time [8], which exposes the enclave to timing-based attacks for threat models requiring trusted time sources. The integration of high-accuracy, secure timers into TEEs can further improve security, enabling time-based attack detection mechanisms [10] and mitigating such vulnerabilities.

Take-away 2 : *TEE technologies like Intel SGX lack trusted time sources and so cannot ensure time integrity. This leaves the system susceptible to timing attacks by malicious privileged software.*

LibOS scheduling. Many libOS implementations employ user-level scheduling for task context switching. However, actual CPU resource management is controlled by the underlying OS, resulting in a dual-layer scheduling model. This layered structure means that while the libOS handles its own scheduling decisions, they are ultimately subject to the host OS's scheduling policies. This dual scheduling can lead to increased latency, as the host OS might not prioritize the libOS process or could frequently preempt it, preventing the libOS from executing its scheduling operations promptly and, in turn, delaying application execution.

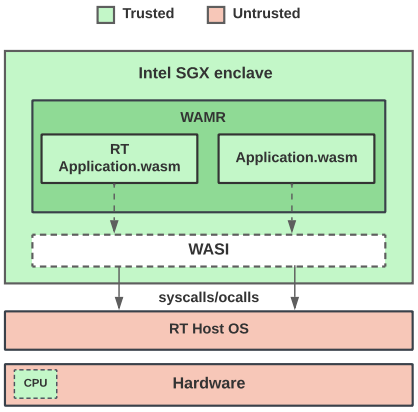

Figure 4 RT application deployed as a Wasm application in an SGX enclave.

4.3 Wasm-based design

WebAssembly (Wasm) is a portable compilation target initially designed for web environments but has since been extended to support standalone execution. Traditional applications can be written in many modern programming languages, e.g., C/C++, Rust, Go, and compiled into Wasm binaries for execution in constrained environments across multiple hardware platforms, such as x86, Arm, and RISC-V. Trusted runtimes like Twine [43, 44] integrate Wasm runtimes, e.g., WAMR, into Intel SGX enclaves. Similar to libOSes, the entire applications are hosted in the enclave, as shown in Figure 4, abstracting away the complexity of the TEE implementation. The resulting TCB size for a Wasm-based runtime is much smaller (a few kilobytes) as compared to a libOS (up to 6.5 GiB for Occlum containers). The WebAssembly system interface (WASI) standardizes the calls to the underlying OS for Wasm applications, offering a means to interact with OS services, such as time retrieval and input/output (I/O), via `ocalls`. Wasm runtimes support two compilation modes: ahead of time (AOT) compilation, where applications are precompiled into native binaries at build time, and just-in-time compilation (JIT), where compilation to machine code occurs at runtime. AOT compilation minimizes runtime overhead, making it ideal for time-sensitive applications, but sacrifices Wasm's flexibility once compiled for a given CPU architecture.

4.4 VM-based design

In this setup, RT applications are deployed within a secure VM, such as Intel TDX, AMD SEV, or Arm CCA. The two important aspects here from a RT perspective are the virtualized environment and the TEE security primitives. Firstly, the additional layer of abstraction introduced in virtualized systems introduces overhead. While server environments for VMs typically comprise optimizations aimed at throughput, these often compromise on aspects like predictable RT behaviour [20]. Also, the inherent resource-sharing (i.e., CPU, memory, timers, *etc.*) in virtualized environments means RT VMs are more impacted from external workloads. Furthermore, VM schedulers typically offer a fair quota of CPU resources to VMs with no regard to the nature of the workloads running within the VM [27]. This inevitably results in increased scheduling and processing latencies for RT applications deployed in such VMs. Secondly, TEE-technologies like Intel TDX introduce an additional layer of

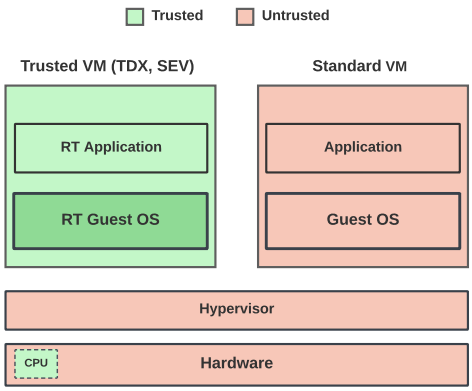

Figure 5 RT application deployed in a trusted virtual machine, e.g., TDX, SEV.

overhead due to the underlying cryptographic operations necessary to ensure confidentiality and integrity of VM data [17,32]. This overhead impacts the completion time of associated RT tasks. From a security perspective, the architecture depicted in Figure 5 provides the largest TCB and hence the largest attack surface.

Take-away 3 : *VM schedulers are oblivious to the nature of workloads running within the VM (and thus the temporal constraints of RT VMs), meaning RT workloads may not be prioritized.*

4.5 Startup times

A key consideration in these architectures is the *startup time*, which represents the time interval from issuing a command to start an entity–such as an enclave, container, or VM–until the entity is fully initialized and capable of executing an application. In TEEs like SGX and TDX, the initialization process includes security-critical operations, such as code integrity verification and remote attestation, which introduce additional startup overhead and may affect real-time performance guarantees.

Notably, startup times become irrelevant if the enclave, container, or VM remains active between invocations, eliminating the need for repeated initialization.

5 Evaluation

This section evaluates RT systems across the different architectures we explored, focusing on addressing the following key questions:
Q1: What are the variations in scheduling latencies of RT tasks with and without a TEE?
Q2: What is the effect of external stress workloads on these scheduling latencies?
Q3: What is the performance of a simple RT application in different TEEs?

Our primary interest in this work is to get a better understanding of the scheduling delays introduced by TEEs in RT applications, as well as the behaviour of shared memory data exchange primitives for RT systems using TEEs. We do not focus much on evaluating the execution times of RT applications in enclaves as these are highly dependent on the use case. For example, industrial control systems execute multiple threads concurrently

each performing their own safety or RT task periodically. Due to differences in nature among safety and RT tasks and in order to make effective use of available compute resources and to reduce product costs, the RT threads are best executed at individual cycle times specifically tailored to their operation. By relying on a simplified setup we expect that our results emphasize more on the impact that trust boundary transitions have when securing RT applications and that any other disturbances in our measurements can be minimized. We believe that the results can be useful for taking decisions when designing and implementing secure RT systems.

Enclave server setup. The SGX evaluations were conducted on a server equipped with an 8-core Intel Xeon Gold 5515+ CPU clocked at 3.20 GHz, 22.5 MiB L3 cache, and 128 GiB of DRAM. The server runs Ubuntu 20.04.6 LTS and Linux RT version 6.9.5-rt5 with a fully preemptible kernel. We used Gramine SGX version 1.8 and WebAssembly Micro Runtime (WAMR) from commit `#0e4dffc4` for running workloads in SGX enclaves. The server was configured with 64 GiB of usable EPC.

VM server setup. Evaluations for Intel TDX were carried out on a 28-core Intel Xeon Gold 5520+ CPU, 52.5 MiB L3 cache, and 128 GiB of DRAM using Ubuntu TDX 2.2 and Linux RT version 6.8.2-rt11 patches. Technologies such as Turbo Boost, Speed Shift, Speed Select, and Hyper-Threading were disabled to operate the processor in a deterministic way. At the OS-level the C-state was fixed to 0, cores 13 to 27 were isolated as well as excluded from read-copy-update (RCU) callbacks, interrupt request (IRQ), and IRQ affinity was set for cores 0 to 12. By isolating the cores from the OS, the Linux scheduler excludes these cores from its load balancing algorithm. Tasks need then to be assigned manually to the isolated cores using `taskset`. Furthermore, we temporarily disabled the network time protocol (NTP) service during evaluations and kept the `sched_rt_runtime_us` at 950 000 µs.

For the VM we allocated 8 virtual CPUs (vCPUs) and pinned them to physical CPUs in the range of 13 to 27. Priorities of the vCPU threads were set to RT priority 98. The VM kernel was setup similar to the host kernel, with the only difference that vCPU cores 4 to 7 were isolated and `sched_rt_runtime_us` was set to 900 000 µs.

Methodology. The Linux Foundation provides an open source benchmarking suite called `rt-tests` which contains programs to test various RT kernel features [29]. A popular program in this suite is `cyclictest`, which evaluates the kernel's scheduling latency and jitter;[3] this helps assess how predictable the system's response to RT tasks is. The most important value in the `cyclictest` results is the maximum detected latency [28], which gives an idea of the worst case latency of RT threads on the evaluated system. Because an ideal RT kernel typically exhibits low scheduling latencies, it is essential to apply some load to the system to obtain worst case latencies. To achieve this, we use `stress-ng`, a popular program used in stress testing, and `hackbench`, a stress tool provided by the `rt-tests` suite. A description of these stress tools can be found in Table 1. We evaluated the different systems first in idle mode, i.e., no stressing, and with the stress programs running concurrently on the same cores. The `cyclictest` evaluations were done following best practices in the field [21, 24]: the RT tasks were spawned by `cyclictest` with a priority of 90 (`-p90`) and intervals (`-i#`) were set slightly higher than measured during a 1 min run. Four cores of the server were used,

[3] Cyclictest measures the difference between a thread's intended wake-up time and the time at which it actually wakes up [28].

Table 1 Stress workloads used alongside cyclictest.

Stressor	Description
Hackbench	Stresses CPU with IPC
Stress-ng VM	Performs memory allocations and paging ops.
Stress-ng IRQ	Generates timer interrupts

and all tests were run for a duration of 1 hour (`-D 60m`). We do not enable memory locking (`-mlockall`) primarily because the `mlock` system call cannot be used on SGX enclave pages. Nevertheless, during the benchmarking process, only `cyclictest` workloads are executed in enclaves.

Lastly, we measured the latency of a RT thread in SGX enclave and TDX modes performing processing operations on data obtained via shared memory. The RT thread is executing a simple safety function were the data is compared to a threshold value. Shared memory provides inter-process communication (IPC) that allows for very fast data exchange at low latency between RT threads by avoiding costly context switches. Furthermore, messaging patterns, such as publish-subscribe for example, can be built on top of shared memory to facilitate data exchanges between two or more RT threads. They allow the threads to operate at individual cycle times and decouple the threads from each other by using asynchronous communication that provides greater flexibility in the design of a RT system.

5.1 Cyclictest

We ran cyclictest for a duration of 1 hour on 4 cores of the experimental server in different setups: (1) no SGX with and without external stress workloads, (2) using Intel SGX (using WAMR and Occlum) with and without external stress-ng workloads, and (3) using Intel TDX with and without external stress workloads. We slightly modified the source code of `cyclictest` to leverage implemented WASI calls by WAMR. We further extended WAMR to provide some POSIX scheduler primitives to `cyclictest` compiled in Wasm. The modifications have been open-sourced and are available in a GitHub repository.[4] Furthermore, cyclictest is compiled AOT for running in WAMR, ensuring optimal performance. We ran `cyclictest` on the different systems (under stress workloads) with the RT tasks executing only on isolated CPUs.

Interpreting cyclictest histogram plots. The majority of latency counts typically form a peak at the lower end of the horizontal axis, indicating that most of the measured latencies are short and within acceptable limits. The position and width of the main peak provide information regarding the average latency/response time and the variability around it. The right-hand side of the graph, i.e., the "tail" indicates higher latencies. In RT systems, long tails are a concern because they show instances where the system experienced significant delays. Also, any noticeable points or clusters in the high-latency region (right side) suggest occasional scheduling delays or system interruptions, and typically occur when the system is under stress. NB: the vertical axes of all `cyclictest` plots in this work are in logarithmic scale.

[4] Repository: `https://github.com/Yuhala/intel-sgx-real-time-ecrts`

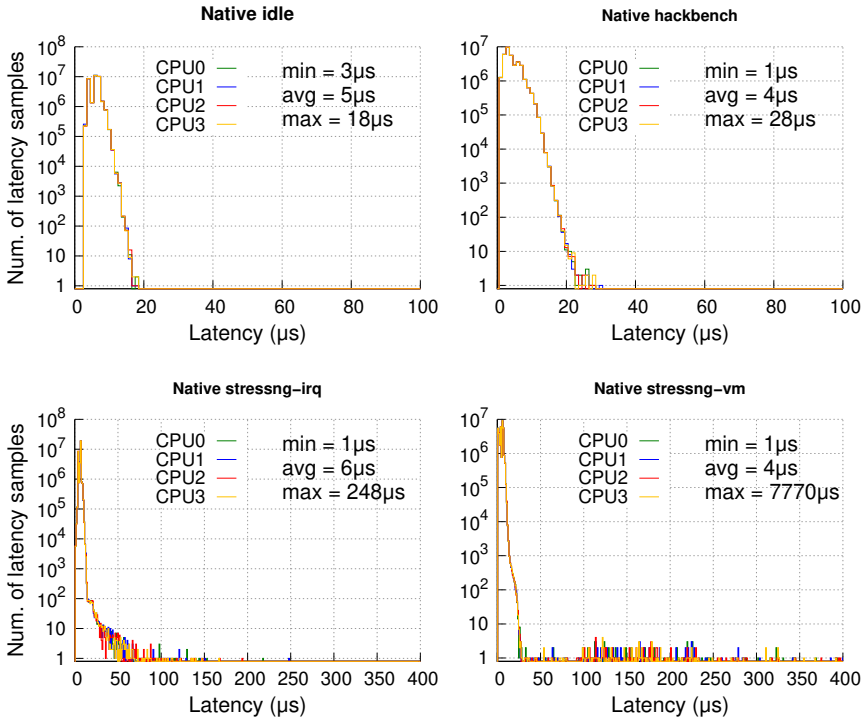

Figure 6 Cyclictest on native system (i.e., no SGX): idle and with external stress workloads.

Native. Figure 6 outlines the results of `cyclictest` on a native (i.e., non SGX) environment. These results serve as a baseline to evaluate latencies in the TEE-enabled systems. The maximum observed latency is 19 µs on an idle system, but increases when different stress workloads are applied, which is mainly due to contention for CPU resources.

WAMR SGX. In an idle WAMR SGX environment, most scheduling latencies remain low, with minimum and average latencies recorded 11 µs and 17 µs, respectively. However, the maximum latency is significantly higher, reaching 1656 µs, about 87 × greater than that observed the native environment. This increase can be attributed to multiple factors: First, *system call overheads*: cyclictest relies on APIs like `clock_nanosleep` and `clock_gettime` for accessing timers, and system calls like `sched_setscheduler` to adjust RT scheduling parameters. In an SGX enclave, these calls trigger context switches, introducing timer slop that inflates the measured scheduling latencies. Previous research on SGX enclaves report `ocall` latencies between 8000 and 14 000 CPU cycles [49, 58, 63, 69]. Second, *runtime abstractions*: runtimes like WAMR introduce an additional abstraction layer, such as the WASI interface, between the application and the host system. These layers act as proxies, introducing API boundary costs which contribute to the overhead during system call invocation.

Take-away 4 : *The additional abstraction layer introduced by WAMR, combined with the context switching overhead required for system calls in enclaves, contributes to increased scheduling latency for RT threads in WAMR SGX.*

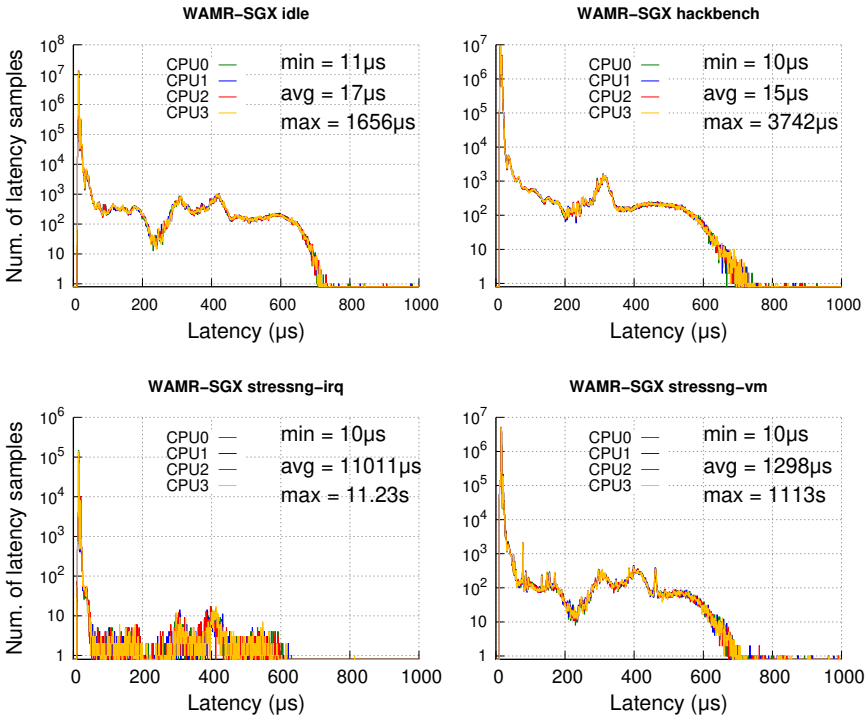

Figure 7 Cyclictest results for WAMR SGX: idle and with external stress workloads.

To mitigate the overhead due to enclave context context switches, RT threads can leverage switchless mechanisms [49, 58, 69] to relay unsupported enclave calls to the untrusted runtime without triggering an expensive context switch. Furthermore, RT threads can leverage techniques like shared memory between the enclave and untrusted which allow triggering the RT thread's processing function based on the value of a shared variable instead of relying on timers or other interrupt mechanisms that invoke costly system calls [48].[5]

Take-away 5 : *Techniques like shared memory and multi-threading can be leveraged to mitigate expensive context switches in RT threads.*

Introducing external stress factors increases these overheads as observed by Figure 7. For compute-heavy stress loads like `hackbench`, the observed latencies increase, reaching a maximum of 3742 µs; this is expected because high IPC activity on the same cores introduces significant CPU contention. For stress-ng IRQ, the highest observed latency is 11.23 s. Since IRQ handling typically has higher priority,[6] increased timer IRQ activity can deprive enclave threads of CPU time, especially for threads waiting to resume after a system call or context switch. Additionally, the stress-ng IRQ workload introduces jitter into the timers, disrupting their reliability and skewing the accuracy of the measurements. Enclave-specific context switches can further amplify the timing discrepancies. The stress-ng virtual memory stressor produces the maximum observed latency, 1113 s (i.e., 18.55 min). This suggests

[5] This can be implemented by keeping the RT thread in a busy-wait loop which periodically reads the value of the shared variable in shared memory and invokes a handler function accordingly.
[6] It is worth exploring the effect on scheduling latencies for cyclictest threads with the highest priority, 99.

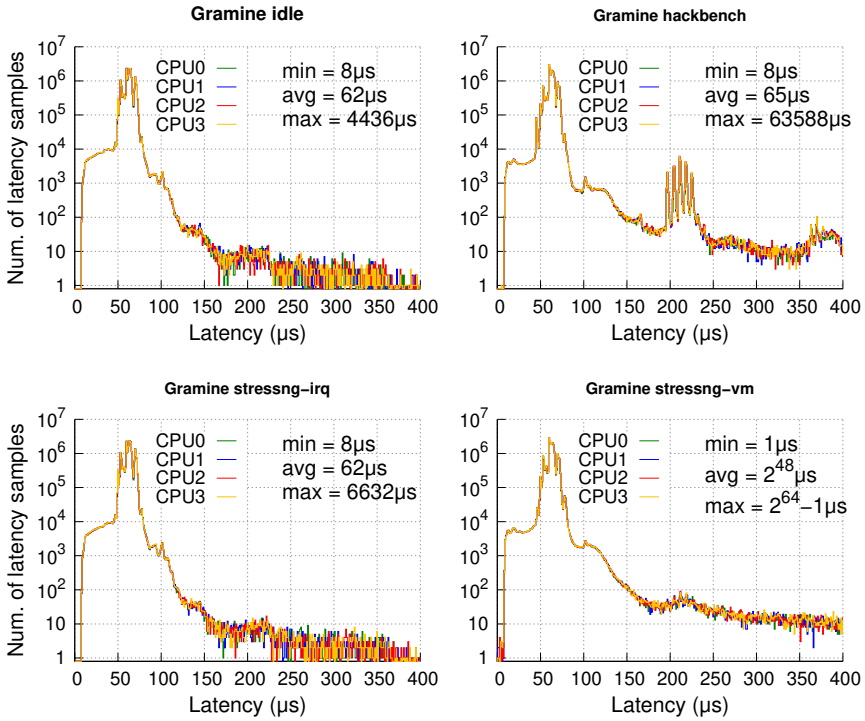

Figure 8 Cyclictest results for a Gramine SGX LibOS.

memory-intensive workloads have the most significant impact on the scheduling latencies of RT threads. This observation is equally true for the native environment, where the maximum latency reaches 7770 µs. Furthermore, memory pressure in the EPC due to excessive virtual memory allocations can cause EPC pages to be evicted to regular DRAM. These paging operations are very expensive [48] and thus inevitably impact the scheduling latencies of RT threads.

Gramine SGX LibOS. Figure 8 illustrates the results for running cyclictest in Gramine LibOS. For an idle system, we observed a maximum latency of 4436μs, which is about 246× and 2.7× higher than the native system and WAMR-SGX respectively. Gramine SGX LibOS binaries generate a significant number of ocalls, with a trivial helloworld example generating over 200 ocalls/asynchronous exits (AEX) [2], a large number of them being scheduler interrupts. These enclave switches lead to additional delays which negatively impact the maximum latency of the RT threads. However, the ocall count is much lower in the sandboxed WAMR-SGX environment. A particularly high maximum latency of $2^{64} - 1\mu$s was observed with the stress-ng VM stressor. This value corresponds to the largest possible unsigned integer (`UINT64_MAX`). Further investigation revealed that this occurred due to missed wake-up events in the cyclictest thread. That is, the scheduler never rescheduled the cyclictest thread after its timer expired. The root cause is the extreme strain on system resources caused by the 64GB virtual memory allocations in the enclave by stress-ng VM, which causes very expensive EPC paging operations, introducing delays that lead to missed wake-up events. To confirm this, we reran the experiment with 4GB virtual memory allocations (less than the 64GB EPC), reducing/eliminating the paging overhead. This resulted in a maximum latency of 2902μs as the paging operations and EPC exhaustion were avoided.

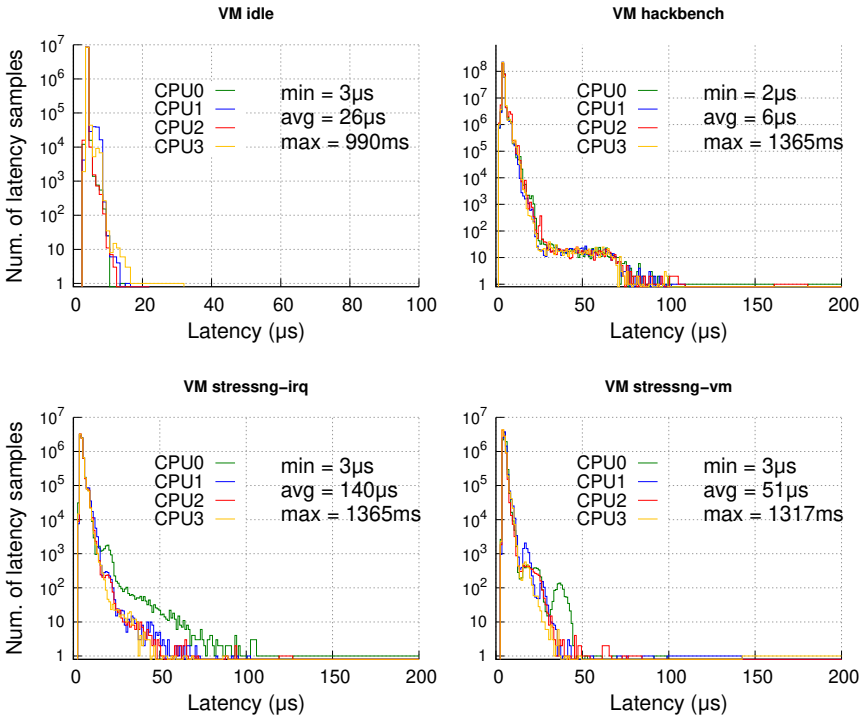

Figure 9 Cyclictest results for a regular VM (without TDX).

Take-away 6 : *Virtual memory allocations within an enclave that exceed the EPC size increase the likelihood of entirely missing real-time events. Therefore, to maintain real-time guarantees, the EPC limit must not be exceeded.*

We equally compiled and executed cyclictest within an Occlum libOS container (in root mode). However, runtime errors occurred due to the `sched_getparam` system call issued by `cyclictest`. While such errors are often caused by Docker restricting certain system calls, the issue persisted even when running the Occlum container with additional capabilities, e.g., `-cap-add=sys_nice`. We defer further investigation of this problem to future work.

Intel TDX. We ran `cyclictest` in a regular VM (no TEE) and one secured with TDX. Similarly, the experiment with the regular VM serves as a baseline against which to compare those in a TDX-enabled VM. In idle mode on the regular VM shown in Figure 9, the minimum, average and maximum observed latencies are 3 µs, 26 µs, and 990 ms respectively. Despite the stringent setup, the system experienced periodic peak latencies on every core of about 1 s between every 1 s to 1.7 s. We observe that enabling TDX increases latencies from 26 µs avg by up to 6 × when idle, and latencies double when under load. Except for the IRQ load, we notice that the cores seem to experience characteristic latency patterns, independent of their identical workload. For example, for stress-ng VM in Figure 10, we clearly identify latencies accumulating for CPU0 at 17 and 210 µs, for CPU1 at 57 and 210 µs, and for CPU2 & 3 at 156 and 210 µs. Further investigation is needed to identify the root cause on these two observations, such as tracing VM events. In particular the combination of TDX with RT patches does not mix well, as both patch sets are developed independently. This situation may improve from Linux kernel version 6.12 on with the integration of RT patches into the mainline kernel.

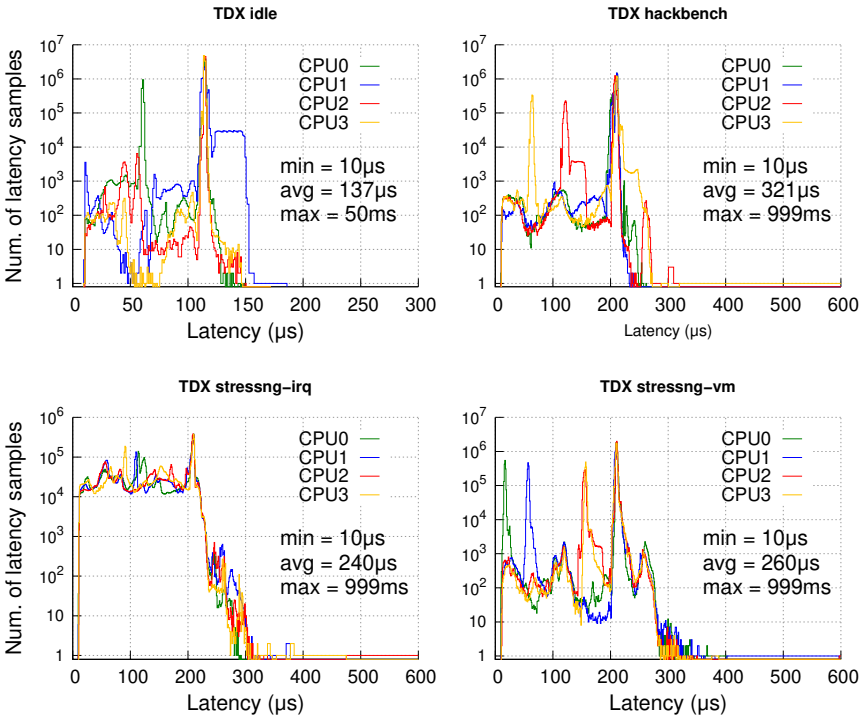

Figure 10 Cyclictest results for a TDX-enabled VM.

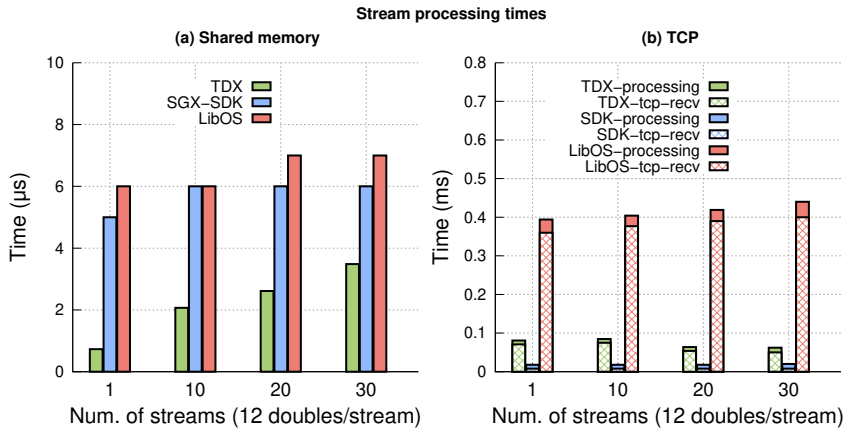

Figure 11 Processing times of RT threads on data streams read from shared memory or transmitted over a TCP connection.

5.2 Real-time application performance

The evaluations with `cyclictest` give us an idea of the scheduling latency associated with RT threads, but not the latency of operations performed by the RT threads. Here we measure the run time of a RT TDX and SGX thread performing data processing cycles. Each data processing cycle involves reading a stream of 96, 960, 1920 and 2880 B of industrial input

data from shared memory or over a TCP connection into enclave memory and processing the data securely. We plot the median times for 1000 runs (after 50 warm-up runs) for each data point. Figure 11 outlines the results obtained.

For shared memory, TDX performs best as the data never has to cross the TEE boundary. Our TDX setup leverages inter-VM shared memory (ivshmem), which is memory mapped at both ends. The total processing times (including the cost of reading from shared memory) [7] for the SGX SDK based system remains relatively consistent across all stream sizes and is marginally better than the Gramine SGX libOS version across the different sizes of data streams. The libOS, however, shows a slight increase in processing time compared to the SDK version for larger streams; this gap points to a potential latency increase when reading/writing shared memory in the libOS as opposed to the SGX SDK version where the enclave has access to the entire application address space with minimal overhead. Nevertheless, both systems provide small overall processing times (within 8 µs) for all data sizes, but these trends are expected to evolve with more complex workloads. When the data is fed to the enclave RT thread via a TCP connection, the total processing time increases, about 42 ×, 3.3 × and 57.7 × larger for the TDX-, SDK- and libOS-based versions, respectively. This is because reading data over a TCP connection incurs higher overhead than accessing shared memory. The increased cost in the libOS is primarily due to the execution of the TCP `recv` system call inside the enclave, as opposed to the SDK-based version where it is performed out of the enclave (i.e., code partitioning), and the data read directly by the enclave thread.[8]

6 Discussion

Leveraging nowadays confidential computing for RT workloads is very challenging. The driving force behind this challenge lies in the nature of the forms of computing: while confidential computing is offering strong confidentiality and integrity guarantees, RT computing is ensuring availability guarantees. Thus, neither form of computing has anything in common with the other form. On the contrary, the confidentiality and integrity guarantees introduce additional overhead that is of opposing nature to the availability guarantees. More specifically, cryptographic operations (i.e., data encryption and decryption) and context switches increase latencies for read and write operations. This is particularly detrimental to RT applications that operate at cycle times in the low millisecond or microsecond range.

A cycle can be broken down into a sequence of read-compute-write phases. Not every RT application is necessarily making use of each of the three phases. Some may require only a read phase while others may only need a write phase, but in general the compute phase is common to all of them. This means that certain RT applications are less impacted by the overhead than others when executing within a TEE. In particular RT applications that make use of read and write phases are being punished twice by the overhead yielding from the decryption and encryption primitives when data is crossing the trust boundary of the TEE. One approach could be to sacrifice on the compute margin, which could be achieved either through software by optimising the code performance or through hardware by using faster processors.

While a hierarchical scheduler would provide availability guarantees to both TEE modes, as for example designed and implemented in [61], the TCB would have to include all TEE-external components comprising the hierarchical scheduler. Extending the TCB would

[7] We include the cost of reading from shared memory because it is a simple variable assignment: `double val = *(pointer)`.

[8] SGX SDK-based enclave threads have access to trusted and untrusted memory.

increase the attack surface that RT applications executing within an Intel SGX and TDX TEE would be exposed to. In order to make use of a hierarchical scheduler with components external to the TEE, those components would need to be executed themselves within a TEE. Furthermore, runtime attestation capabilities would ensure that the integrity of the external hierarchical scheduler components are guaranteed. However, the runtime attestation mechanism could affect or introduce additional latencies as RT tasks are being scheduled. Another important aspect of the hierarchical scheduler would be the time source. As the threat model considers a powerful attacker, there are several way in which time can be manipulated. For example, the attacker could alter the system time by means of root privileges or network protocols such as NTP or precision time protocol (PTP) that is often times being used for synchronizing industrial control systems. This means that there is a need for secure timers being made available on platforms that implement secure RT systems. For example, Intel SGX can provide access to a trusted time service through the Converged Security and Management Engine (CSME) [14], while Intel TDX can ensure access to a TEE-internal time stamp counter (TSC) timer. Having to rely on a platform service as in the case of Intel SGX introduces latency and the time resolution is also not as high as with an TEE-internal timer. For these reasons, we believe that many challenges remain both on the software and hardware side in order to be able to implement secure RT systems based on x86 platforms. On Arm platforms there are far less challenges as new application-grade processors provide secure timers with interrupts routed entirely within Arm TrustZone, thus completely isolated from the normal world.

However, first the software support and tooling ecosystem has to be improved before a stage can be reached where performance decisions can be taken. A major issue we faced while combining confidential with RT computing is the current lack of compatibility and integrity testing. In the case of the Linux kernel, neither the Intel TDX patches nor the `PREEMPT_RT` patches are widely available on current Linux distribution kernels. Both patch sets have been developed over the past years on separate branches and have only recently been merged into the mainline Linux kernel [59]. This means that during the development process there has never been an easy or low effort way to test and validate the compatibility and integrity of the features with others. Following such an approach, after the introduction of a new feature, there is still the need for a phase to ensure compatibility and integrity with other features. The Linux kernel is making use of release candidates for such purposes, but whether these are sufficient for covering all compatibility and integrity cases remains debatable. Despite recent efforts it is still necessary for the foreseeable future to manually apply these patch sets. Although integration of both patch sets often times goes smoothly, enabling both Intel TDX and `PREEMPT_RT` features at the same time can break user space. For that reason we were not able to make use of the latest Canonical TDX releases and needed to revert back to Canonical TDX release 2.2.

We summarize our findings of the proposed architecture and designs in Table 2. Our evaluations have shown that in general all confidential computing approaches suffer from extremely high tail latency. In application-based TEEs such as SGX SDK, WASM, and Gramine, the issue is due to the lack of RT mechanisms such as, for example, preemption. As enclaves are self-contained, there is the need for a protocol, e.g., Compute Express Link [3], which allows not only secure use of resources external to the enclave, but also availability guarantees on these resources. VM-based TEEs, such as Intel TDX, are less affected by this problem, because of the virtualization of resources and the support of RT mechanisms. For this reason, their tail latency, while still not ideal, is in a range that could with sufficient improvements become acceptable in the near future.

Table 2 Comprehensive summary of TEE design evaluations for RT applications.
Legend: 🟢 Good, 🔴 Bad (Colours in-between show the progression from good to bad), ❓ Not available.

Architecture	TCB (i.e., Security)	Avg Latency	Max Latency	Usability
SGX SDK	🟢	❓	❓	🔴
WASM	🟢	🟢	🔴	🟢
Gramine	🟢	🟢	🔴	🟢
TDX	🟢	🟢	🔴	🟢
Native	🔴	🟢	🟢	🟢

As our evaluation indicates, congested resources are particularly problematic, as they are the main reason for high tail latency. This issue can only be resolved by raising awareness of the problem and encouraging both the confidential and RT computing communities to join their research efforts in addressing common challenges. One of the main challenges will be the introduction of RT mechanisms into TEEs for enabling availability guarantees to RT application executed inside them. However, the introduction of these mechanisms should not come at the cost of usability.

7 Related work

Several exiting works in literature have attempted to assess and understand how to provide RT performance of various software and hardware platforms. We distinguish three main classes of such works and elaborate on what has been done in what follows.

7.1 Real-time Capabilities of Operating Systems

The work in [24] establishes a first step towards realizing a measurement-based methodology for assessing the RT performance and capabilities of RT operating systems. In particular, the paper investigates the RT capabilities of RT Linux with the `PREEMPT_RT` patch on the Raspberry Pi 5. The authors use Cyclictest in addition to other various stressors to simulate extreme operational conditions. They compare the performance and determinism of Linux kernels with and without the `PREEMPT_RT` patch, and observe that RT Linux has a 294 × shorter maximum latency than on regular Linux.

Another work [15] evaluates scheduling latency under `LITMUS_RT`, a RT extension of the Linux kernel, based on data obtained with a ported version of cyclictest. The authors' main effort was aimed at finding a mechanism to compare `LITMUS_RT` against stock Linux and `PREEMPT_RT`. Such a comparison was not possible before, since scheduling latency of `LITMUS_RT` is typically evaluated using Feather-Trace while that of Linux and its principal RT variant, the `PREEMPT_RT` patch, are typically measured using cyclictest, two tools that have fundamentally different outputs. The paper's main conclusion was that `LITMUS_RT` introduces only minor overhead itself, but it inherits mainline Linux's severe limitations in the presence of I/O-bound background tasks.

7.2 Real-time Capabilities of Virtual Environments

A second class of literature investigated the real-time capabilities of virtual environments. For example, works in [6, 25, 27, 52] analyze the performance of virtualized RTOS environments, mainly showcasing challenges in achieving hard real-time guarantees, and pointing out issues that should be addressed for this to be a reality.

Another line of work [5, 19, 56] investigated container-based designs for deploying RT systems. Some major shortcomings and immature aspects of real-time container virtualization that prevents the expansion of the technology have been identified by [56]. Among the shortcomings highlighted by the authors are:

1. Tools support: shortcomings on this end were identified mainly in the lack of orchestration tools that can schedule real-time containers based on pre-configured capabilities, as well as frameworks to expose the runtime requirements of the containerized real-time applications and ensure optimal allocation of containers to system resources.
2. Real-time communication: the main limitation here was the lack of inter-container real-time communication as well as real-time data management across containers.

7.3 Real-time Capabilities of Confidential Computing

A third line of existing works have investigated TEE-based security in the context of RT applications. Literature in [7, 20, 47, 50, 57, 61] provide techniques, such as for example hierarchical scheduling, for building security-oriented RT systems based on TEEs. Most of these works, however, focus on user-end TEE technologies like Arm TrustZone or RISC-V Keystone. While some interesting work and results have been provided on how to ensure availability of resources for RT processes inside those TEEs, it is not clear how such results can be used on Intel systems, specifically for Intel-based TEEs.

In that sense, our work provides a systematic study of RT capabilities of security architectures based on server-end TEE technologies like Intel SGX and TDX, which resonates well with industrial offerings that currently have high dependency on Intel hardware and/or virtual machine technologies. Our work also provides recommendations on practical deployment of RT applications in cloud and edge settings.

8 Conclusion

This work represents the first in-depth study of TEE-based architectures for deploying RT applications. It evaluates scheduling and processing latencies for RT threads in Intel SGX and TDX across various workload types and provides practical recommendations for optimizing the deployment of TEE-based RT applications. This work can be further extended by varying system parameters such as thread priorities and RT scheduling algorithms, as well as by applying different levels of external stress workloads during the cyclictest runs. These varied parameters would provide deeper insights into the behavior of the various architectures under investigation.

References

1 Edge for Smart Secondary Substations (E4S) Alliance. `https://www.e4salliance.com/`.

2 Gramine SGX Performance tuning and analysis. URL: `https://gramine.readthedocs.io/en/stable/performance.html`.

3 Homepage – Compute Express Link. `https://computeexpresslink.org/`.

4 Linux Foundation Energy SEAPATH. URL: `https://lfenergy.org/projects/seapath/`.

5 Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the Linux kernel. *SIGBED Rev.*, 16(3):33–38, November 2019. `doi:10.1145/3373400.3373405`.

6 Mehdi Aichouch, Jean-Christophe Prevotet, and Fabienne Nouvel. Evaluation of an RTOS on top of a hosted virtual machine system. In *2013 Conference on Design and Architectures for Signal and Image Processing*, pages 290–297. IEEE, 2013.

7 Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling open systems through strong availability guarantees for enclaves. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 1357–1372, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460120.3484782.

8 Fatima M. Anwar and Mani Srivastava. Applications and challenges in securing time. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association. URL: https://www.usenix.org/conference/cset19/presentation/anwar.

9 Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015. doi:10.1145/2799647.

10 Nicolas Bellec, Simon Rokicki, and Isabelle Puaut. Attack detection through monitoring of timing deviations in embedded real-time systems. In *ECRTS 2020-32nd Euromicro Conference on Real-Time Systems*, pages 1–22, 2020. doi:10.4230/LIPICS.ECRTS.2020.8.

11 Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

12 Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2988336.2988350.

13 Tiago Brito, Nuno O. Duarte, and Nuno Santos. Arm TrustZone for secure image processing on the cloud. In *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 37–42, 2016. doi:10.1109/SRDSW.2016.17.

14 Shanwei Cen and Bo Zhang. Trusted Time and Monotonic Counters with Intel Software Guard Extensions Platform Services, 2017.

15 Felipe Cerqueira and Björn Brandenburg. A comparison of scheduling latency in Linux, PREEMPT_RT, and LITMUS_RT. In *9th Annual workshop on operating systems platforms for embedded real-time applications*, pages 19–29. SYSGO AG, 2013.

16 Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 142–157, 2019. doi:10.1109/EuroSP.2019.00020.

17 Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. *ACM Comput. Surv.*, 56(9):238:1–238:33, 2024. doi:10.1145/3652597.

18 Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016. URL: http://eprint.iacr.org/2016/086.

19 Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Reducing temporal interference in private clouds through real-time containers. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 124–131, Los Alamitos, CA, USA, July 2019. IEEE Computer Society. doi:10.1109/EDGE.2019.00036.

20 Janis Danisevskis, Michael Peter, and Jan Nordholz. Minimizing event-handling latencies in secure virtual machines. *arXiv preprint arXiv:1806.01147*, 2018. arXiv:1806.01147.

21 Raimarius Delgado and Byoung Wook Choi. New insights into the real-time performance of a multicore processor. *IEEE Access*, 8:186199–186211, 2020. doi:10.1109/ACCESS.2020.3029858.

22 Sen Deng, Mengyuan Li, Yining Tang, Shuai Wang, Shoumeng Yan, and Yinqian Zhang. CipherH: Automated detection of ciphertext side-channel vulnerabilities in cryptographic implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6843–6860, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/deng-sen.

23 Advanced Micro Devices. AMD SEV-TIO: Trusted I/O for secure encrypted virtualization. https://www.amd.com/content/dam/amd/en/documents/developer/sev-tio-whitepaper.pdf. Accessed on 11-07-2023.

24 Wannes Dewit, Antonio Paolillo, and Joël Goossens. A preliminary assessment of the real-time capabilities of real-time Linux on Raspberry Pi 5. In *18th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*. Alexander Zuepke and Kuan-Hsun Chen, 2024.

25 Tao Ding, Qin-fen Hao, Bing Zhang, Tie-gang Zhang, and Li-ting Huai. Scheduling policy optimization in kernel-based virtual machine. In *2010 International Conference on Computational Intelligence and Software Engineering*, pages 1–4. IEEE, 2010.

26 Gabriel Fernandez, Andrey Brito, and Christof Fetzer. Triad: Trusted timestamps in untrusted environments. In *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 169–176, 2023. doi:10.1109/CloudCom59040.2023.00037.

27 Nils Forsberg. Evaluation of real-time performance in virtualized environment, 2011.

28 Linux Foundation. Cyclictest. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start. Accessed on 11-11-2024.

29 Linux Foundation. Rt-tests. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests. Accessed on 11-11-2024.

30 Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: Long live KASLR. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, volume 10379 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2017. doi:10.1007/978-3-319-62105-0_11.

31 David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011. doi:10.1109/SP.2011.22.

32 Intel. Intel® trust domain extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-final9-17.pdf. Accessed on 20-11-2024.

33 Intel. Intel Xeon Scalable Processors. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions-processors.html, 2021. Accessed on 16-10-2024.

34 Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. Supporting Intel SGX on multi-socket platforms. *Intel Corp*, 2021.

35 Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020. doi:10.1145/3399742.

36 David Kohlbrenner, Shweta Shinde, Dayeol Lee, Krste Asanovic, and Dawn Song. Building open trusted execution environments. *IEEE Security & Privacy*, 18(5):47–56, 2020. doi:10.1109/MSEC.2020.2990649.

37 Dayeol Lee, Dongha Jung, Ian T. Fang, Chia che Tsai, and Raluca Ada Popa. An Off-Chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 487–504. USENIX Association, August 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol.

38 Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 465–484, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/osdi22/presentation/li.

39 Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, Santa Clara, CA, July 2017. USENIX Association. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

40 Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, 2016. doi:10.1109/HPCA.2016.7446082.

41 Canonical Ltd. Tuning a real-time kernel. https://ubuntu.com/blog/real-time-kernel-tuning. Accessed on 02-12-2024.

42 Michael M Madden. Challenges using Linux as a real-time operating system. In *AIAA Scitech 2019 Forum*, page 0502, 2019.

43 Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for WebAssembly. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 205–216. IEEE, 2021. doi:10.1109/ICDE51399.2021.00025.

44 Jämes Ménétrey, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, Giovanni Mazzeo, Arne Hollum, and Darshan Vaydia. A comprehensive trusted runtime for WebAssembly with Intel SGX. *IEEE Trans. Dependable Secur. Comput.*, 21(4):3562–3579, 2024. doi:10.1109/TDSC.2023.3334516.

45 Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. Confidential vms explained: An empirical analysis of amd sev-snp and intel tdx. *Proc. ACM Meas. Anal. Comput. Syst.*, 8(3), December 2024. doi:10.1145/3700418.

46 Subhas C. Misra and Virendra C. Bhavsar. Relationships between selected software measures and latent bug-density: guidelines for improving quality. In *Proceedings of the 2003 International Conference on Computational Science and Its Applications: PartI*, ICCSA'03, pages 724–732, Berlin, Heidelberg, 2003. Springer-Verlag.

47 Anway Mukherjee, Tanmaya Mishra, Thidapat Chantem, Nathan Fisher, and Ryan Gerdes. Optimized trusted execution for hard real-time applications on cots processors. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, RTNS '19, pages 50–60, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3356401.3356419.

48 Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 227–240, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/atc18/presentation/oleksenko.

49 Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 238–253, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3064176.3064219.

50 Sandro Pinto, Tiago Gomes, Jorge Pereira, Jorge Cabral, and Adriano Tavares. IIoTEED: An enhanced, trusted execution environment for industrial IoT edge devices. *IEEE Internet Computing*, 21(1):40–47, 2017. doi:10.1109/MIC.2017.17.

51 Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A comprehensive survey. *ACM Comput. Surv.*, 51(6), January 2019. doi:10.1145/3291047.

52 M Ramachandran. Challenges in virtualizing real-time systems using KVM/QEMU solution. In *Proceedings of the 14th Real-Time Linux Workshop*, 2013.

53 Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time Linux kernel: A survey on PREEMPT_RT. *ACM Computing Surveys (CSUR)*, 52(1):1–36, 2019. doi:10.1145/3297714.

54 Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 955–970. ACM, 2020. doi:10.1145/3373376.3378469.

55 John A Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, 1990.

56 Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V Papadopoulos. Real-time containers: A survey. In *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

57 Alex Thomas, Stephan Kaminsky, Dayeol Lee, Dawn Song, and Krste Asanovic. Ertos: Enclaves in real-time operating systems. *Woodstock*, 2018.

58 Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, pages 22–27, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3268935.3268942.

59 Linux Torvalds. Merge tag "sched-rt-2024-09-17". https://web.git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=baeb9a7d8b60b021d907127509c44507539c15e5.

60 Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 645–658, Santa Clara, CA, USA, 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

61 Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 352–369, 2022. doi:10.1109/SP46214.2022.9833604.

62 Andrew Waterman and Krste Asanovi. The RISC-V instruction set manual. https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf. Accessed on 16-10-2024.

63 Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. *SIGARCH Comput. Archit. News*, 45(2):81–93, June 2017. doi:10.1145/3140659.3080208.

64 Shuhao Wu. Real-time programming with linux. https://shuhaowu.com/blog/2022/02-linux-rt-appdev-part2.html. Accessed on 04-11-2024.

65 Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, USA, 2015. IEEE Computer Society. doi:10.1109/SP.2015.45.

66 Peterson Yuhala, Pascal Felber, Hugo Guiroux, Jean-Pierre Lozi, Alain Tchana, Valerio Schiavoni, and Gaël Thomas. SecV: Secure code partitioning via multi-language secure values. In *Proceedings of the 24th International Middleware Conference, Bologna, Italy, December 11-15, 2023*, Middleware '23, pages 207–219. ACM, 2023. doi:10.1145/3590140.3629116.

67 Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. Fortress: Securing iot peripherals with trusted execution environments. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, SAC '24, pages 243–250, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3605098.3635994.

68 Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi. Montsalvat: Intel SGX shielding for GraalVM native images. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, pages 352–364, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3464298.3493406.

69 Peterson Yuhala, Michael Paper, Timothee Zerbib, Pascal Felber, Valerio Schiavoni, and Alain Tchana. SGX switchless calls made configless. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 229–238, Los Alamitos, CA, USA, June 2023. IEEE Computer Society. doi:10.1109/DSN58367.2023.00032.