Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves

Fritz Alder
fritz.alder@acm.org
imec-DistriNet, KU Leuven
Leuven, Belgium

Jo Van Bulck
jo.vanbulck@cs.kuleuven.be
imec-DistriNet, KU Leuven
Leuven, Belgium

Frank Piessens
frank.piessens@cs.kuleuven.be
imec-DistriNet, KU Leuven
Leuven, Belgium

Jan Tobias Mühlberg
jantobias.muehlberg@cs.kuleuven.be
imec-DistriNet, KU Leuven
Leuven, Belgium

ABSTRACT

Embedded Trusted Execution Environments (TEEs) can provide strong security for software in the IoT or in critical control systems. Approaches to combine this security with real-time and availability guarantees are currently missing. In this paper we present Aion, a configurable security architecture that provides a notion of guaranteed real-time execution for dynamically loaded enclaves. We implement preemptive multitasking and restricted atomicity on top of strong enclave software isolation and attestation. Our approach allows the hardware to enforce confidentiality and integrity protections, while a decoupled small enclaved scheduler software component can enforce availability and guarantee strict deadlines of a bounded number of protected applications, without necessarily introducing a notion of priorities amongst these applications. We implement a prototype on a light-weight TEE processor and provide a case study. Our implementation can guarantee that protected applications can handle interrupts and make progress with deterministic activation latencies, even in the presence of a strong adversary with arbitrary code execution capabilities.

CCS CONCEPTS

• **Security and privacy** → **Trusted computing**; **Operating systems security**; *Embedded systems security*; • **Computer systems organization** → *Real-time systems*; **Availability**.

KEYWORDS

trusted computing, availability, open systems, resource sharing

ACM Reference Format:
Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. 2021. 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), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3460120.3484782

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484782

1 INTRODUCTION

With the increased connectivity of devices all across the computing spectrum comes an increasing demand for systems that are not locked down but are more dynamic and open to changes after they are deployed in the real world. An *open* system runs software components (tasks, processes, …) from several stakeholders that do not necessarily trust each other. The resources of such system, including memory, devices, and the CPU, must be shared among these software components without introducing security vulnerabilities that would allow a malicious component to violate the security expectations of another component. Traditionally, Operating System (OS) kernels have the responsibility of enforcing appropriate isolation between components, and, hence, the OS kernel has been part of the Trusted Computing Base (TCB).

However, experience has shown that operating system kernels can have vulnerabilities too, and several approaches have been explored to reduce the amount of trust in the OS kernel:

First, there is a long line of work in reducing the *size* of kernels (e.g., move to microkernels), or relying on simpler hypervisors or security monitors for enforcing isolation [6, 23, 36]. The key idea is that the trusted layer of software gets smaller, but all software components still need to fully trust the system software for any of their security properties.

Second, formal verification of system software has been proposed as a mechanism to reduce the likelihood of vulnerabilities, and, hence, to better justify the level of trust in system software[17, 19].

Third, work in the trusted computing research area has developed the idea of Trusted Execution Environments (TEEs) or enclaves [1, 5, 7, 20, 21, 26, 30]. These approaches make it possible to remove most (if not all) system software from the TCB, but they cannot guarantee all desired security properties. More specifically, while integrity and confidentiality of enclaves can be guaranteed with a TCB consisting of just the enclave software itself and the hardware, no availability guarantees can be provided. More generally, these systems can provide strong guarantees for resources (like memory) that are *spatially* shared, but not for resources (like CPU time) that are *temporally* shared. In the best case (for instance, in Intel SGX), the operating system kernel can *preempt* temporally shared resources from misbehaving enclaves, at the cost of having

to trust the kernel for availability properties. In other cases, there are no availability guarantees in the presence of malicious enclaves.

The objective of this paper is to improve the state-of-the-art in this third approach. We propose a hardware/software co-design that supports classic enclave-like isolation of software components in an open system, and that improves on that classic isolation by also providing availability guarantees. Our system supports the secure *temporal* sharing of resources (including CPU and I/O devices) among mutually distrusting software components with a small TCB. More specifically, a given enclave software component needs to trust: *(i)* its own code and the hardware for confidentiality and integrity properties, and *(ii)* its own code, the hardware, the drivers of the shared devices it requires access to, and a small, trusted scheduler enclave for availability properties. Crucially, since the scheduler is only trusted for availability, our design protects the confidentiality and integrity of vital enclave applications even against a misbehaving scheduler. Furthermore, when the scheduler is well-behaved, our design can provide strong availability guarantees (including real-time guarantees) to software components in the presence of arbitrary malicious software on the platform outside the TCB (including malicious enclaves, malicious drivers for devices not used by this specific component, and system software besides the trusted scheduler).

Our design targets small embedded systems (specifically, our prototype is based on a TI MSP430 16-bit processor running the RIOT OS), both because these can benefit most from availability and real-time guarantees, and because this allows us to focus on the essence of our design: building on preemption combined with a safe bounded atomicity primitive. Extensions to larger systems, such as for instance Intel SGX-scale processors, are not in the scope of this paper, and are left for future work.

In summary, the contributions of this paper are:

- a novel hardware-software co-design of a security architecture for open systems that extends the strong security properties of modern hardware TEEs with strong guarantees on enclave availability, even in the presence of powerful software adversaries on the same platform.
- a prototype implementation built by extending an existing TI MSP430-based TEE and by extending the existing RIOT IoT operating system.
- a case-study driven evaluation of the security and availability provisions and the costs of the design.

2 PROBLEM AND ASSUMPTIONS

To illustrate the problem and our platform requirements, we first discuss the base platform that we use as a starting point for our work. We then describe a simple application scenario with specific security and availability needs that cannot be realized with classic TEE implementations. Finally we generalize this to derive platform requirements and discuss these in the context of related work.

In general, we aim to support *open* systems, which are systems that allow multiple distrusting stakeholders to dynamically load arbitrary applications at runtime. While it is obviously possible to combine an open system with priority-based scheduling, the interesting and most difficult case is dealing with mutually distrusting stakeholders executing code with the same priority. Only in this case resources have to be divided fairly.

2.1 Generalized Base Platform

The base platform we start from is an embedded TEE that provides an enclave-like isolation mechanism. This base platform supports the creation of enclaves that offer the following security guarantees. First, the software in an enclave is isolated from all other software on the same platform, including system software such as the operating system. Second, enclaves support (local and remote) attestation: they can provide cryptographic evidence about their identity (characterized by a cryptographic hash of the binary code of the enclave). These security guarantees rely on a small trusted computing base, sometimes even only the hardware.

More specifically, in terms of isolation, the base platform guarantees that: *(i)* the data section of an enclave is only accessible while executing code from the code section of that same enclave, and *(ii)* the code section can only be entered through one or more designated entry points. These isolation guarantees are simple, but they have been shown to be strong enough and useful to enforce confidentiality and integrity properties of enclaved applications or modules. For instance, Patrignani et al. [32] show how encapsulation mechanisms from Java-like object-oriented languages can be securely compiled to a platform that supports enclaves. This implies that confidentiality and integrity properties of the enclave can be guaranteed in an *open* system: an enclave developer only needs to trust (or verify) the code of their own enclave (and possibly other enclaves that the enclave depends on, such as device driver enclaves). As a consequence, mutually distrusting enclaves can co-exist on the platform, and neither one needs to trust the other to maintain its own security, which is limited to confidentiality and integrity. The construction and the benefits of such a base platform is well understood, and Maene et al. [24] provide a survey of existing platforms.

However, these platforms lack any kind of *availability* guarantee. On some platforms [13, 30, 31] enclaves can protect themselves from being interrupted (and, hence, get atomicity guarantees) for security purposes, but as a consequence a misbehaving enclave can abuse such atomicity guarantees to disrupt the system and make it unavailable to other enclaves. On systems [7, 20, 21, 26] where enclaves are interruptible, on the other hand, enclaves do not get any guarantees on progress. For instance, enclaves might never get scheduled, or when they are scheduled they can get interrupted again without having made any progress. Also, enclaves may need to acquire resources other than memory or CPU, e.g., access to I/O devices like sensors or communication channels, and no guarantees can be provided that the enclave can acquire these within a bounded time span. Note that some Memory-Mapped I/O (MMIO) devices may only use a specific memory region to interact with the applications. This means that this memory region needs to be temporally shared between applications as a spatial sharing may not be possible for certain control or status bits. Finally, some platforms handle security violations in such a way that a security violation from one enclave can impede the progress of another one. For instance, a security violation might lead to a platform reset [13, 30, 31].

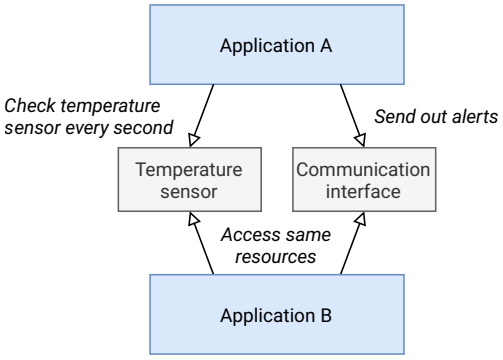

Figure 1: Simple example of two applications periodically accessing the same shared resources.

This set of shortcomings leads us to the problem we set out to solve in this paper: how can an enclave platform provide availability guarantees, while maintaining the desired strong confidentiality and integrity guarantees, *i.e.*, in particular that only the hardware plus the enclave itself and any dependent enclaves need to be trusted or verified. By doing so, the platform we design is the first enclave platform to provide a strong notion of availability for mutually distrusting enclaves, where neither one needs to trust the other to maintain its own security, which includes confidentiality, integrity, *and availability* properties.

2.2 A Running Example

Figure 1 depicts a scenario with two applications $\mathcal{A}$ and $\mathcal{B}$ that execute periodically, monitoring the same temperature sensor. Each application will trigger an alert if the temperature exceeds a programmed threshold. These alerts are communicated over the same, shared communication interface. We assume an open system where all system resources, including the CPU, the sensor and the communication interface, may be used by multiple applications. The deploying stakeholders of $\mathcal{A}$ and $\mathcal{B}$ are neither aware of each other's applications, nor would they trust each other's applications to behave collaboratively. However, both stakeholders consider their applications to be critical as harm may be caused if the alarms are not triggered within strict time bounds. The stakeholders do trust the execution platform to uphold a notion of spatial and temporal isolation for their respective applications, and they may rely on primitives such as remote attestation to be ensured of their application execution on the intended platform. In regards to input and output from the temperature sensor and to the communication interface, the applications trust the utilized peripherals and an attacker controlling one of the peripherals themselves or a failed sensor are out of scope of their attacker model. This means that the platform aims to provide guarantees only up to the device boundaries and tamper-resistant sensors or resilience against network denial-of-service attackers are left to orthogonal research. At the same time, peripheral drivers on the system and their communication with attached devices are in scope of the guarantees as long as the attached peripheral is responsive.

While the spatial isolation properties required by our running example are generally well understood in existing TEE platforms, these platforms do not provide the required *availability* guarantees. This includes the temporal sharing of MMIO devices. Specifically, the requirements of $\mathcal{A}$ and $\mathcal{B}$ to run periodically, make progress, and get a guaranteed opportunity to send out the alert cannot be realized with existing TEE platforms. Especially not considering that in our example, no application trusts any other application on the device, for example considering them as compromised by a strong software adversary.

2.3 Security & Availability Guarantees

We follow the established attacker model in TEE research (cf. Maene et al. [24]), where all software that is not explicitly part of an application's TCB is considered to be under the control of the attacker. We consider hardware-level attacks to be out of scope for our prototype. In particular, an attacker cannot physically disconnect components or control peripherals.

Under this model, the platform should provide the same guarantees as the described generalized base platform above, *i.e.*, confidentiality and integrity of mutually distrusting applications combined with the possibility to attest applications to remote parties. As a generalization of the availability requirements of the running example, the platform has to provide the following *additional* availability guarantees for a bounded number of protected applications:

- *Bounded activation latency:* the platform guarantees a specific finite bound on the maximal time that can elapse between an event (in the example case, a timer interrupt) and the execution of the first instruction of an enclave that wants to act on the event.
- *Guaranteed progress:* the platform guarantees that within a specific time interval T (e.g. a second), at least x percent of the CPU cycles goes to the monitoring application (where T and x can be configurable, but an application can securely attest these values to a remote stakeholder).
- *Guaranteed device access:* device drivers can be programmed to provide assurance to an application that it can acquire access to all devices it needs within a specific finite time T. Obviously, the temperature monitoring application needs to trust (or verify) the code of the sensor driver and communication channel driver, and use it appropriately to get these guarantees. But an important point is that *no other applications competing for the same resources need to be trusted.*
- *Safety independence:* faults in the executions of other applications do not impact the availability of the temperature monitoring application. Only the application itself (including dependent code) must be trusted (or verified) not to have faults (including security faults) to preserve availability.
- *No trust hierarchy:* the same guarantees can be given to multiple mutually distrusting applications. Two independent applications can perform monitoring tasks and compete for the communication channel to send out alerts, and both of them will get the availability guarantees we discussed, without either having to trust the other. It is in this sense that our platform is truly an *open* system: progress and real-time guarantees can be offered to a number of protected applications that run at the *same* priority.

Considering the last guarantee, we note that equivalent guarantees can only be given to competing applications up to an upper limit depending on the nature of the resource. Intuitively, no realistic guarantee can be given if the requirements exceed the available schedulability of the resource. This restriction spans across all shared resources such as time (managed and guaranteed by the scheduler), and attached peripherals (such as the temperature sensor and communication interface drivers). We see it as a software responsibility of each (trusted) resource driver to only provide a guarantee if this guarantee can realistically be given.

In summary, these guarantees make it possible to ensure for our example applications $\mathcal{A}$ and $\mathcal{B}$ that temperature alerts will be sent out within a hard real-time bound in the presence of buggy or malicious code on the platform. More specifically, the protected $\mathcal{A}$ is capable of achieving its goals even if $\mathcal{B}$ is malicious and attempts to monopolize resources, and vice versa. In Section 5 we will show how this simple application can be realized on our platform with the above availability guarantees. To the best of our knowledge, no other TEE is capable of providing these combined security and availability guarantees.

2.4 Related Work

Most closely related to our approach are existing hardware/software co-designs for light-weight embedded systems with a strong emphasis on security. The key publications here are Masti et al. [25], TrustLite [20], TyTAN [7], and Sancus [30]. We explicitly focus on light-weight embedded systems and on related work that can be used as a base platform for our design. Thus, we focus on related work that at least provides spatial isolation to its software components and that enables the deployment of *mutually distrusting enclaves*. This leaves literature such as SMART [13] or VRASED [31] out of scope. We also explicitly omit research from this list that either focuses on higher-level embedded systems such as Cure [5] or CHASE [10], or that targets the problem domain of real-time and mixed-criticality systems without discussing their security. While Masti et al. also lacks certain spatial isolation properties that would be necessary to use it as a base platform, their solution does already provide some availability features.

There is a wide body of work on mixed-criticality systems in the real-time community, but for most of this literature, important differences with our approach are *(i)* priorities and *(ii)* not aiming for the same strong confidentiality and integrity guarantees that enclaves offer. In mixed-criticality systems, a clear priority order is applied to all running applications. As such, the operating system can prioritize a closed, known set of applications and ensure the progress of important code [8]. On an *open* platform, however, such clear priority order does not exist and all dynamically deployed applications that share a resource need to be assumed to have the same priority. A number of designs have been proposed that do focus on security, but for more heavy-weight processors than the ones we consider in this paper. We will discuss all such additional related work further in Section 2.4.2.

2.4.1 Light-weight embedded systems. We summarize the temporal isolation guarantees given by closely related work and Aion in Table 1. Masti et al. [25] investigate the topic of trusted scheduling on embedded systems and present a hardware/software co-design that, based on crafted hardware components plus an omnipotent trusted domain software layer, can securely schedule applications even under attack from a software adversary. In their approach, the authors assume a conventional priority-based system and can provide availability guarantees to the highest-priority thread. Their approach can, furthermore, allow for guaranteed peripheral access through a hardware-level peripheral manager that is responsible for all peripherals on the device.

TrustLite [20] and TyTAN [7] are two security architectures based on the Intel Siskiyou Peak platform. TrustLite utilizes an execution-aware memory protection unit that links the program and data sections of applications together and shields them as a unified trustlet against interference from untrusted parties. While TrustLite does provide secure peripherals and the ability to preempt enclaves, it neither implements sharing of resources, a bounded limit on atomic periods, nor guaranteed progress for any other application than the one with the highest priority. TyTAN is an improvement over the TrustLite platform in terms of a dynamic deployment of applications, but it does not extend the guarantees that are the concern of this work. Although TyTAN does provide a version of trusted scheduling, this does not entail a strict bounded activation latency since attackers can still trigger infinite atomic sections. Furthermore, peripherals can be secured, but not securely and safely shared with other applications without losing availability guarantees. Both the design of Masti et al. and TrustLite require a static deployment of software and a platform reset is needed to load additional applications. Sancus [30] is a program counter-based TEE for the 16-bit MSP430 processor that can utilizes its memory isolation capabilities for enclaves to also support secure memory mapped peripherals. We introduce Sancus more thoroughly in Section 4.1 but note that the original Sancus does not allow the preemption of enclaves and faces the same limitations as TrustLite and TyTAN in regards to our availability guarantees.

In contrast with earlier approaches, Aion provides applications with a complete set of the discussed temporal isolation guarantees. While Masti et al. can give the largest subset of the desired guarantees, their solution enforces strict priorities among applications using a static, hardware-level scheduler. This means that progress can only be actually guaranteed for a single application with the highest priority instead of multiple applications at once. Their solution also requires applications to trust each peripheral since a single peripheral manager manages all platform peripherals. Aion, however, only requires applications to trust the peripherals they utilize, does not require applications to depend on any higher-priority application outside of their TCB, and can provide progress guarantees to multiple applications simultaneously through a flexible, software-defined scheduling policy.

2.4.2 Beyond light-weight embedded platforms. Outside of the scope of what we refer to as light-weight embedded TEE processors, related approaches have been presented. In [8], Burns and Davis provide a comprehensive review of approaches to implement mixed-criticality systems, albeit with little consideration for security.

System designs with a focus on security that provide at least a subset of the guarantees provided by Aion are CURE and seL4. CURE [5] is a multicore RISC-based TEE that provides exclusive assignment of system resources, e.g. peripherals, to single enclaves.

Table 1: Comparison of Aion to earlier work on light-weight embedded hardware/software co-designs in regards to temporal isolation guarantees. ◐ denotes that this guarantee can only be fulfilled for a single enclave.

	Masti	TrustLite	TyTAN	Sancus	Aion
Bounded activation latency	✔	–	–	–	✔
Guaranteed progress	◐	◐	◐	◐	✔
Guaranteed device access	✔	◐	◐	◐	✔
Safety independence	–	–	–	–	✔
No trust hierarchy	–	–	–	–	✔
Architecture	AVR	Siskiyou Peak		MSP430	

This exclusive access allows for secure I/O operations similar to Sancus [30] (which we extend in this paper). In addition, CURE features novel enclave types which can, e.g., span multiple privilege levels and might be interesting for mixed-criticality use cases. However, CURE is not designed around real-time guarantees and does not provide a notion of availability for enclaves.

Notably, the seL4 microkernel [19] enforces strong security properties with formally proven access control mechanisms. Kernel operations also have verified safe upper bounds on their worst-case execution times [6, 36] and interrupt latencies. Based on these features, mixed-criticality support has been implemented in seL4 [23], with similar guarantees for isolation and availability as in Aion. However, our approach is unique in that we do not rely on a trusted kernel for security but instead build upon a security-centric approach to hardware/software co-design. Therefore, seL4 comes with a larger software TCB than Aion and does not provide TEE features such as sealing and attestation.

A number of approaches aim to build real-time systems on top of ARM's TrustZone TEE[2, 12, 22, 27, 33, 34]. In difference to our work, TrustZone [1] TEEs do not implement a hardware-only TCB as they rely on a trusted operating system to isolate processes in the secure world; they further do not natively provide enclave attestation and sealing. Most importantly, these works all investigate the impact of TEEs on real-time behavior and demonstrate the feasibility of using TEEs in these systems. However, none provide availability guarantees in the presence of a strong software attacker.

Azab et al. [2] proposed a TrustZone-based implementation of a protected security monitor that is capable of securing the operating system that runs in the insecure world. While this system monitor cannot be bypassed and operates deterministically, it does not provide dependable scheduling of guest tasks.

Pinto et al. [33, 34] presented a virtualization solution that demonstrated how multiple guests efficiently co-exist in isolation, and with deterministic execution. The approach does consider an attacker with the ability to trigger interrupts to harm system availability, and proposes a solution based on privileged and unprivileged interrupts. In difference to Aion, Pinto's work provides integrity only at boot time, by means of TrustZone's secure boot process, and does not consider dynamic updates to code or scheduling policy.

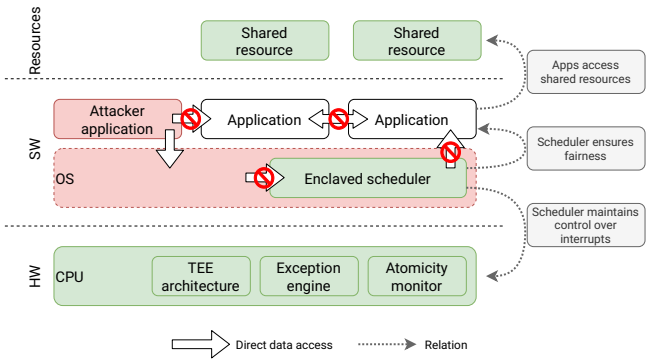

Figure 2: System overview with trusted components highlighted in green. The scheduler has exclusive control over interrupts and can enforce a periodic scheduling, but cannot access protected application enclaves directly.

With a specific focus on mixed-criticality, Dong et al. [12] proposed and evaluated a dual-criticality approach to kernel virtualization, which enables a real-time kernel to share the same platform with a general-purpose operating system. No provisions are made to address the security impact of software-level attackers.

Finally, Mukherjee et al. [27] presented a technique to enforce the correct timing requirement of a task, along with a sufficient test for schedulability. The paper focuses on reducing the overall number of transitions between the insecure and secure worlds in applications by fusing together secure sections of that application. This allows to minimize the associated I/O traffic and improves the temporal predictability of the system, but dissolves spatial isolation between the secure sections.

3 DESIGN

In the following we present the design of Aion that, based on conventional light-weight embedded TEE architectures, can bring strong temporal isolation guarantees to multiple, mutually distrusting applications. We base our prototype implementation on Sancus, but stress that the general design of Aion is independent of the underlying platform. Figure 2 shows an overview of the Aion system and its core components.

The first core component is the underlying hardware-based **TEE architecture** that provides the core guarantees of confidentiality and integrity. In the following, we focus only on TEE characteristics that are necessary in addition to the established protection mechanisms, e.g., how interrupts or violations of the TEE's security policy are handled. We are confident that these additions could be implemented on top of all discussed light-weight embedded TEE architectures. The second component of our design is a hardware-based **exception engine** that is triggered whenever an interrupt or violation occurs. This exception engine cannot only interrupt unprotected but also protected, *i.e.*, enclaved, applications. Furthermore, the exception engine is triggered on any violation of a platform policy such as reading from protected memory or jumping into the middle of a protected code region. The third and fourth elements of Aion are an **atomicity monitor** and an **enclaved scheduler**. The hardware-based atomicity monitor ensures that the enclaved

scheduler is the only entity that has full control over handling any system events, e.g. interrupts or violations. For this, the atomicity monitor implements a notion of *bounded atomicity* and carefully controls interrupt behavior during context switches, e.g., when entering an enclave. The sofware-level enclaved scheduler is the handler of all events on the system, and orchestrates the execution flow of the system when events occur. All four components play together to enable the scheduler to issue fair scheduling decisions. We will now detail these four core pillars of our design.

3.1 TEE Architecture

We build Aion around TEEs that provide memory isolation for dynamic enclaves. From the investigated TEEs, TyTAN [7] and Sancus [30], support these requirements natively.

In addition, Aion requires two additional features that need to exist to design our security architecture. First, violations of the TEE security policy should not result in a reset or in blocking the system. A system reset is a common solution to violations since illegal writes or reads from protected memory regions may only be detected after the offending instructions has completed. If an architecture detects a security violation after it occurred, a system reset prevents any malicious code to use the result or side effect of this access. In Aion, however, the platform must not be impacted by any offending instruction but instead proceed with an exception and hand control over to a handler of this violation. It is crucial that offending instructions do not complete but are instead either stopped or their effects rolled back before control is handed over to a violation handler in constant time. As such, the handler of the violation must not necessarily be privileged or trusted by any party.

Second, TEE-internal hardware operations must be interruptible. While we discuss preempting enclaves in Section 3.2, some operations of the TEE architecture may need a large amount of cycles to complete. Common examples of such operations are cryptographic operations or the enabling or disabling of enclaves. Adversaries in Aion are capable of arbitrary code execution and may attempt to stall the system by issuing long-running cryptographic operations. To prevent this, the TEE architecture must support the preemption of these operations. A successful or unsuccessful completion must be notified by the hardware to the issuer of the operation when control is resumed so that benign applications can restart the operation in case of an interrupt; the policy for this must be part of the hardware-software contract to enable developers to design enclaves that can make progress. Additionally, the hardware must ensure that any cryptographic state is cleared and removed from memory before interrupts are handled to prevent information leakage. We implement our prototype of Aion on Sancus which builds on MSP430 and has no cache or advanced microarchitecture. Therefore, execution time is fully deterministic and only depends on the instruction type and memory accesses. This simplifies our approach but does not limit generality: Aion can be implemented on any TEE-platform for which a WCET-analysis is possible. Determining upper bounds for the execution of scheduler operations is the only strict requirement for Aion.

Also note, that while remote attestation may on first glance not seem essential to Aion, attestation in Aion provides remote stakeholders with the guarantee that *(i)* the right code is loaded untampered in a protected application enclave; *(ii)* the scheduler enclave was loaded correctly, ensuring a fair availability policy; and *(iii)* expected implementations of shared drivers are used. We thus see attestation as an integral part of how Aion would be used in practice.

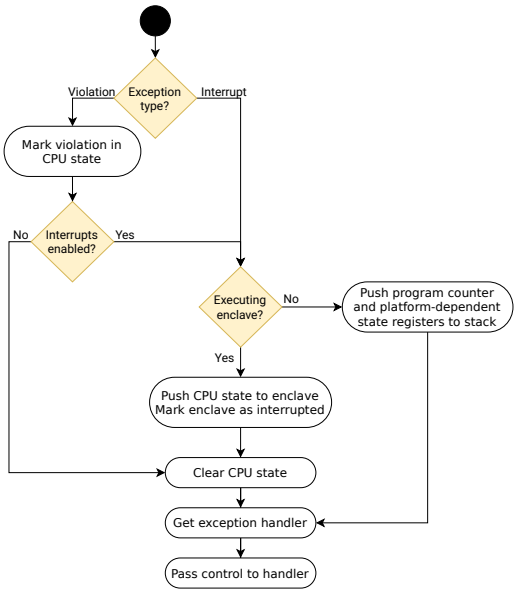

Figure 3: High-level flow of the exception engine. Two main paths are distinguished: interrupts and violations. On interrupts, context state is saved in the enclave. On violations, a marker is first set in the CPU state.

3.2 Exception Engine

Whenever an interrupt or a policy violation occurs, the exception engine in Aion is responsible for switching from the current job to the enclaved scheduler. This ensures that the scheduler can always fairly schedule the next application and ensure that all applications maintain a fair share of the resource CPU time. In its operation, the exception engine distinguishes between two types of exceptions: *interrupts* due to periodic or aperiodic events and *violations* of platform policies. Figure 3 shows a high-level flow of the exception engine. Note, that violations are always handled immediately after the offending instruction completes but the handling of interrupts is delayed by the platform-specific global interrupt-enable flag. An immediate handling of violations ensures that even in atomic sections, dangerous violations are immediately handled and the offending job can be punished.

Handling interrupts. On interrupts, the exception engine has to store the current state of the running job in a way that execution can be resumed at a later point. For this, the exception engine needs to distinguish whether the current execution is of an unprotected application or whether an enclave is being executed. For unprotected applications, the behavior of the exception engine is the same as for regular platforms where usually only the current program counter and potential state registers need to be saved on the program's stack. Since the running program is unprotected, the process of

storing the program state in the application's memory region can be a responsibility of the scheduler and be done in software.

For protected applications, however, *i.e.*, enclaves, the exception engine needs to store all context information of the running job in the enclave's protected memory. Depending on the implementation platform, the context information usually entails all CPU registers. This process is done in hardware because the enclaved scheduler should not have access to the protected memory of the interrupted enclave and can thus not perform this process in software. After storing the context information in the enclave, all context information is cleared before handing execution over to the enclaved scheduler. Since enclaves can only ever be entered through predefined call gates, the enclave's entry routines must on their next execution, furthermore, also be able to detect whether the enclave was interrupted previously. Thus, the exception engine also leaves a marker for the enclave that it should restore its execution context instead of accepting potential execution parameters that could overwrite a currently running execution flow. The specifics of this marker can be left to the implementation of Aion, e.g., storing a single bit at a known location is sufficient.

Handling violations. In contrast to interrupts, violations do not occur during the normal behavior of a platform but are usually the result of an unauthorized attempt by an adversarial job. We consider two types of violations that are both handled by the exception engine: security and availability violations. Security violations are defined by the TEE architecture and revolve around the hardware protections of the TEE such as protecting memory regions or preventing illegal jumps into the middle of protected regions. Availability violations on the other hand are introduced by the atomicity monitor and occur whenever a program attempts to enter too long atomic periods or attempts to illegally prolong the current atomic period. We explain the atomicity monitor and how it enforces an upper bound on all atomic periods below.

For both types of violation, we can assume that they are not usually triggered by a benign job and it can be assumed that if a job experiences one, it is either controlled by the adversary, a victim of the adversary, or being tricked by the adversary, e.g., to access another protected memory region through an unchecked pointer [39]. Since the last example can be ruled out by proper input vetting of enclave code, we design Aion around the assumption that any violation is the result of an adversary. To alleviate the impact this may have on applications that do suffer a policy violation during benign behavior, we additionally introduce a violation marker that is set on enclave violations in the CPU context to inform the enclave that it recently suffered a violation. The exact implementation is left to architecture specifics, but any available bit in a status register suffices as long as it cannot be set by software.

Figure 3 shows the behavior of storing violations on the left side. After setting the violation marker, the whole CPU state is stored as it would be for an interrupt. On its next entry, the enclave can check that its last operation was aborted due to a violation. However, if interrupts were not enabled at the time of the violation, the exception engine does not perform this context save to ensure that it not accidentally overwrites an old interrupt context. This is needed since attackers could otherwise call into enclaves and create an availability violation at the cost of the called application.

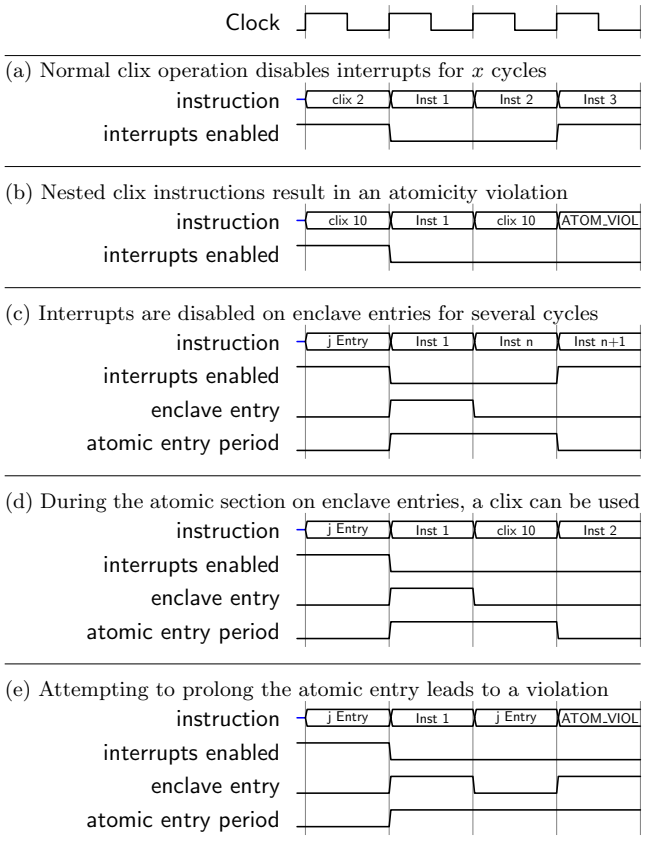

Figure 4: Representation of the desired behavior of bounded atomic sections. `clix` instructions temporarily disable interrupts but cannot be nested. On enclave entry, a short atomic period is started which can be prolonged with a `clix`.

If a violation occurs during the process of storing the CPU context, this process is aborted and the exception engine jumps ahead to clearing the CPU state. This ensures that the hardware cannot be tricked into performing memory writes to areas that the current enclave is not privileged to access.

3.3 Atomicity Monitor

To prevent attackers from impacting the availability of the system, it is necessary to block all attempts that completely disable interrupts. At the same time, the enclaved scheduler in Aion is the main driver of the resource CPU time and requires special privileges in regards to this resource. As such, the scheduler in Aion is the only entity that has the capabilities to disable interrupts on the platform. Since the scheduler is crafted carefully, this privilege does not change the availability guarantees of the system.

While denying any program aside from the scheduler the ability to disable interrupts is beneficial to the availability guarantees of the system, it is certainly not desirable to also prevent all benign usages of atomic sections. In addition to functionality issues that may arise for shared resources if they are interrupted in a critical state, there are also additional concerns in the context of enclaved programs. During entry of an enclave, atomic sections are crucial

to allow the enclave to restore its interrupt context from memory before another interrupt context can be written over the current one. To overcome this limitation, we introduce a special `clix` instruction similar to the design of Masti et al. [25] which starts a bounded atomic period. Figure 4 shows the use and several edge cases of this `clix` instruction. When issuing a `clix`, the hardware disables interrupts for exactly x cycles after which it automatically enables interrupts again (Figure 4.a). Programs can choose x individually up to an upper bound that is set by the platform designer depending on the deployed shared resources. Any `clix` instruction that requests a number of cycles larger than the upper bound and any attempt to nest `clix` periods trigger an atomicity violation (Figure 4.b). We design atomicity violations to be handled by the exception engine as described previously and assume that the atomicity monitor clears all current state when experiencing a violation such as the current count of remaining cycles left in the `clix` period. Issuing atomicity violations ensures that attackers cannot perform `clix` instructions that are out of the bounds of the system's designers chosen acceptable worst-case latency between two interrupts. It, furthermore, ensures that attackers can never prolong their granted atomic period without at least experiencing one cycle of enabled interrupts in which an incoming event can be processed.

While the `clix` instruction technically allows to perform the critical part of an enclave entry in an atomic section, adversaries could still issue an interrupt right at the moment when an enclave is entered. This may lead to issues as an existing interrupt context in the enclave could be overwritten by the adversary's interrupt with a new context that points to the start of the enclave entry. Such a data loss and integrity violation is not acceptable. To prevent this, the atomicity monitor additionally ensures that on each entry of an enclave, *i.e.*, on each context switch into a new protected region, interrupts are disabled for a very limited amount of cycles as shown in Figure 4.c. This gives the enclave entry code enough time to issue a `clix` instruction of the length it needs to restore its interrupt context. Since the exact cycle duration that each application needs to be interruptible again may vary, we allow applications to define this cycle length via the `clix` instruction rather than automatically issuing a long atomic period at each enclave entry. Furthermore, this dynamic `clix` length at enclave entry allows each enclave to decide whether it wants to utilize several cycles of hardware-guaranteed progress before the scheduler could preempt this application again. For some applications, such a guaranteed immediate progress may be more valuable than other progress longer after the deadline. As can be seen in Figure 4.d, issuing a `clix` during the few cycles of an atomic entry period terminates the atomic entry and seamlessly proceeds into a `clix` period. However, any attempt to prolong this atomic entry is prevented with atomicity violations (Figure 4.e).

Our atomicity design serves two main purposes: First, Aion allows the use of atomic sections while at the same time maintaining hard limits on the activation latency of an arriving interrupt. Second, the length of issued atomic sections are purely in the responsibility of software under the restriction enforced by the hardware. This helps in the potential attestation of code that uses atomic sections and increases the performance of benign applications that do not always have to enter a long atomic period if this is not necessary.

A complete overview of the atomicity state machine can be seen in Figure 6 in Appendix A.

3.4 Enclaved Scheduler

The previous core elements of Aion have laid the foundation for a trusted scheduler that is in full control of the shared resource CPU time. The exception engine ensures that all state is cleared and control is handed over to the scheduler on all interrupts and violations. The atomicity monitor limits the atomic periods of any job besides the scheduler itself. To enable a scheduler to utilize this foundation and provide trusted scheduling, however, the scheduler must itself also be protected by the TEE architecture and, hence, run inside an enclave. This is crucial as the scheduler can only provide consistent and fair scheduling decisions if it is unaffected by any attempts of the adversary and if control is always deterministically returned to the same scheduler entry code. With the combination of these properties, the enclaved scheduler can provide a fair real-time scheduling of dynamic applications on an open system.

Practical implementations of Aion benefit of a timer peripheral that is solely controlled by the scheduler. This allows the scheduler to ensure fair scheduling for configurable time periods and can also be used as a basis for a trusted time service for applications.

4 PROTOTYPE IMPLEMENTATION

We implemented Aion on top of the Sancus TEE and the RIOT operating system, specifically Sancus 2.0 as presented by Noorman et al. [30] and RIOT in major version 2019.10 which bases on the original work of Baccelli et al. [3, 4]. We chose this combination as Sancus is an open-source architecture based on the 16-bit TI MSP430, running at 8 MHz, and RIOT is equally available as open-source and has support for MSP430 processors. Sancus already provides the desired confidentiality and integrity guarantees. However, certain modifications were still necessary, especially surrounding the additional requirements Aion makes on the TEE architecture (cf. Section 3.1). Furthermore, because RIOT is designed to be a highly modular priority-based operating system, certain adjustments were required to the scheduler and the way threads are handled to implement an open system with this OS.

In the following we briefly describe Sancus and RIOT, and then discuss how we adapt these systems to implement our solution. The full source code of Aion and the modified toolchains of Sancus and RIOT are available as open-source[1].

4.1 Background: Sancus and RIOT

The Sancus TEE. Sancus [28, 30] is an open-source embedded TEE [24] with a hardware-only TCB that extends the memory access logic and instruction set of a low-cost, low-power openMSP430 [15] microcontroller. Sancus supports multiple mutually distrusting software components that each consist of two contiguous memory sections in a shared single-address-space. A hardware-level program counter-based access control mechanism [38] enforces that an enclave's private *data section* can only be accessed by its corresponding *code section*, which can only be entered through a single *entry point*. Sancus's generic memory isolation primitive can, furthermore, be used to provide secure driver enclaves with exclusive ownership over MMIO peripheral devices that are accessed through the address space. Since Sancus modules only feature a

[1] https://github.com/sancus-tee/sancus-riot

single contiguous private data section, however, secure I/O on Sancus platforms requires these small driver modules to be entirely written in assembly code, using only registers for data storage [29].

Sancus also provides hardware-level authenticated encryption, key derivation, and key storage functionality by extending the CPU with a cryptographic core. This cryptographic core can be used to implement secure communication as well as both local and remote attestation by employing a key hierarchy between the infrastructure provider, the application developer, and individual enclaves. Finally, Sancus comes with a dedicated C compiler that automates the process of enclave creation and hides low-level concerns such as secure linking, private call stack switching, and multiplexing user-defined entry functions through the single physical entry point.

RIOT OS. RIOT is an open-source operating system for the IoT, which puts special emphasis on supporting real-time applications on resource-constrained devices [3, 4]. In contrast to other embedded OS kernels, RIOT provides the full set of features expected from an OS, ranging from hardware abstraction, kernel capabilities, system libraries, to tooling.

RIOT is designed to be tickless, which means that the scheduler is not executed at specific intervals but instead only when necessary. The standard RIOT model is a cooperative scheduling model where it is assumed that applications actively yield whenever they wish to pass control over to the next application. However, to also support periodic events, RIOT allows jobs to set timers to sleep for a period of time. For this, RIOT accesses the timer peripheral, tracks the passed time of the system, and maintains a list of active timers and the thread they are connected to. This setup is ideal for mixed-criticality systems as the highest priority job will always be scheduled next and can run as long as necessary until it either cooperatively yields to pass control over to the next job or until an interrupt arrives and stops the job. For applications of the same priority, however, RIOT assumes a fair and cooperative scheduling through yields which places all other applications of the same priority within an application's TCB.

The RIOT scheduler can provide scheduling decisions in constant time, *i.e.*, in O(1) due to its reliance on a bitmask that depends on the amount of configured priority levels. Sleeping is implemented in O(n) due to an unlimited amount of possible timers.

4.2 Modifications to Sancus

We made multiple changes to the Sancus hardware to implement Aion. All of these changes are made under the assumption that a scheduler has a fixed enclave hardware ID of 1, *i.e.*, the scheduler is the first enclave that is loaded. Specifically, we *(i)* modified the exception engine to handle interrupts and violations according to Section 3.2, *(ii)* implemented an atomicity monitor component according to Section 3.3, *(iii)* placed restrictions on parts of the status register to only be modified by the scheduler, and *(iv)* made cryptographic operations interruptible (in an abandon-restart fashion).

All changes to the Sancus architecture are backwards-compatible with Sancus 2.0 [30] and the MSP430 specification. This was validated with the default tests provided by the OpenMSP430 project that Sancus is based on and with new tests for cases where we added functionality. To provide full backwards-compatibility with

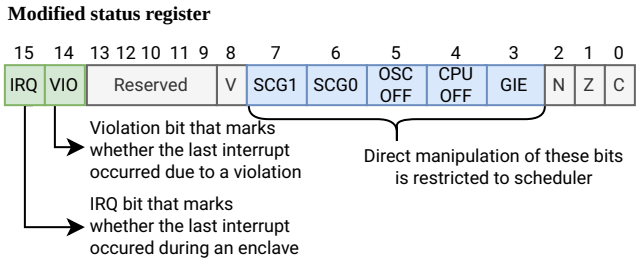

Figure 5: Overview of the status register and our changes. Bits highlighted in blue (bits 3-7) are restricted to the scheduler. Bit 15 marks whether the last interrupt occurred during an enclave. Bit 14 marks whether a violation occurred.

the specification, our availability restrictions do not come into effect before the first enclave, *i.e.*, the scheduler, is initialized.

In the following, we focus only on the essential aspects of our implementation that are not immediately derived from the design of Aion as presented in Section 3. Specifically, this applies to the exception engine, the status register, and the cryptographic core.

Sancus exception engine. Sancus 2.0 originally only supports the preemption of non-enclave code. Thus, we extend the exception engine to perform the tasks as outlined in Section 3.2. In our implementation, we utilize a configurable location in the enclave data region to store the CPU context plus a violation marker when receiving an interrupt or a violation respectively. The context-saving hardware logic is subject to the same access-control checks as the interrupted enclave, and any violations during the processing of an interrupt or another violation lead to the abort of the context saving as shown in Figure 3.

In MSP430, an Interrupt Vector Table (IVT) at a fixed location in the memory layout is used to determine the handler of an interrupt. In our implementation, we assume that the scheduler registers itself for all interrupts and violations, and then locks the IVT by wrapping it in the protected section of a small driver enclave, thus preventing any further access to the IVT.

Status register modifications. The MSP430 status register contains multiple flags e.g., for arithmetic operations and is stored on interrupts and restored together with the program counter on a `reti` instruction. However, in addition to these flags, the MSP430 status register also contains flags that are considered sensitive in Aion. Figure 5 shows an overview of the status register and our modifications. Most obvious, we restrict the disabling of the Global Interrupt Enable (GIE) bit to the scheduler. However, we allow the setting of this bit at all times which allows applications to terminate their own `clix` or atomic entry period ahead of schedule. Additionally, we also restrict bits 4 to 7 to the scheduler which could be used to completely switch off the platform, such as the CPUOFF flag, or which switch off the internal oscillator that is used as a timer. Furthermore, we add two flags to the reserved portion of the status register that are set by hardware and cannot be written from software: the IRQ flag (in bit 15) and the violation flag (in bit 14). The violation flag in bit 14 is set by the exception engine when it processes a violation and is the implementation of the violation marker as described in Section 3.2.

The IRQ flag in contrast exists for purely functional reasons and helps the scheduler to restore jobs as either unprotected code or as enclaves. Since by default the scheduler has no reliable method of deducing whether the hardware interrupted an enclave or unprotected code, the exception engine sets the IRQ flag after clearing the CPU state and before handing control to the scheduler.

Cryptographic core. Finally, we changed the behavior of Sancus's cryptographic instructions to update the Zero flag (bit 1) in the status register to indicate whether the operation completed or was aborted due to an interrupt arrival. The resulting abandon-restart semantics is similar to how Intel SGX handles long-latency cryptographic operations, such as `einit` [18, §40.3]. Specifically, whenever an interrupt request arrives during a cryptographic operation, the CPU resets the cryptographic core (without committing or leaking any internal state), sets the zero flag, and updates the program counter before state saving proceeds as usual via the exception engine. This behavior ensures that interrupt response times cannot be delayed by long-standing cryptographic operations (cf. requirements in Section 3.1). Interrupted cryptographic instructions can be simply restarted later when they are followed by a conditional jump that tests the zero flag.

4.3 Modifications to RIOT

In Aion, we need to protect the scheduler and its associated data structures from outside interferences. At the same time, it is desirable to provide a similar functionality as the unmodified RIOT. Thus, we map the scheduler enclave over the RIOT scheduler and incorporate core features of the RIOT timer. Since the scheduler is executed on every interrupt already, we also grant it exclusive access to the timer peripheral which we map into the protected memory region of the scheduler. This allows scheduling decisions not only based on expiring timers such as sleeping jobs, but it also allows other applications to use the scheduler as a source for trusted system timings. It, furthermore, enables the scheduler to be the only instance that monopolizes the shared resource of CPU time. In our prototype, the scheduler disables interrupts during its execution and will never interrupt itself. This increases the interrupt latency of our prototype and is not strictly necessary to uphold the defined guarantees. With more engineering effort, the scheduler could also be implemented to allow interrupts at carefully selected parts of its execution paths.

As discussed above, a fair scheduling can only exist if the default state of the system is schedulable. Any platform owner that accepts new application to be deployed to the open system must check that the requirements of the new application do not exceed the capabilities of the available shared resources. If the shared resources are schedulable, however, the Aion scheduler can enforce a fair share for each deployed application. For the prototype, we limit the number of maximum running or sleeping applications but allow the attacker to register additional applications up to this limit.

5 EXPERIMENTAL EVALUATION

We evaluate Aion in two steps. First, we present a case study implementing the running example from Section 2.2. Then, we provide a cycle-accurate performance evaluation for all operations impacting the real-time performance of the hardware and the scheduler.

```
def sync_input:
  CLIX <cycles to complete>
  return read_sensor()

def async_output(payload):
  CLIX <cycles to complete>
  try:
    i = buf_free(get_caller_id())
  if i != 0:
    output_buf[i] = payload
  except: fail

def async_io_task:
  while True:
    for i in output_buf
      output_buffer(i)
```

Listing 1: Pseudo-code of the I/O enclave $\mathcal{I}$ of our case study.

5.1 Case Study

We demonstrate the security and availability features of Aion by implementing the running example from Fig. 1. Our case study features three enclaved RIOT jobs that all run with the highest priority. These jobs implement the application enclaves $\mathcal{A}$ and $\mathcal{B}$, and an I/O enclave $\mathcal{I}$. The latter provides an interface to synchronously read the sensor and to asynchronously dispatch messages to a serial line. The enclaves make use of Sancus's TEE features [30], including isolation guarantees and secure linking between $\mathcal{A}$ and $\mathcal{I}$, and $\mathcal{B}$ and $\mathcal{I}$; they can further be remotely attested. All three enclaves schedule timer interrupts to be woken up at regular intervals.

In Listings 1 and 2 we illustrate interesting aspects of our implementation. Aion's development toolchain is based on that of Sancus and currently supports programming in C and assembly. We decided to present only the enclave entry functions (as opposed to internal functions that can only be called from within the same enclave) in Python-like pseudo code to reduce the complexity and focus on important security and software engineering aspects that are enabled by Aion. The C implementation of our case study is given in Appendix B and as part of the open-source artifact.

I/O job and API. Following Listing 1, $\mathcal{I}$ provides three entry points: `sync_input` returns a sensor reading; the code to operate the MMIO resource – a few assembly instructions – reside in the internal function `read_sensor`. The function first executes `clix` to ensure atomic execution of this operation. Following our semantics of `clix`, it is up to the developer to guarantee that `sync_input` completes with the end of the requested `clix` period. The execution of the `clix` itself is protected by the atomic entry period. Similar to `sync_input`, `async_output` is also an atomic function. But instead of performing the I/O operation immediately, the `payload` is buffered. The function may throw an exception if no free buffer is available for the specific calling context and we anticipate that $\mathcal{I}$ would provide guaranteed buffers for a number of protected jobs such as $\mathcal{A}$ and $\mathcal{B}$, while other jobs would have to share buffers. In our example, this decision is based on the Sancus `get_caller_id` primitive, which allows $\mathcal{I}$ to identify the calling enclave. We have hard-coded this for reasons of simplicity and discuss a more general implementation in Section 6. Finally, `async_io_task` is an interruptible function to output buffered payloads from `async_output`. The implementation of `output_buffer` would again be atomic to ensure non-interference during the I/O operation. Indeed, $\mathcal{I}$ is free

```
def worker:
  while True:
    t = sync_input()
    if (t > threshold):
      async_output("WARNING")
    sleep(1s)
```

Listing 2: Pseudo-code of the application enclaves $\mathcal{A}$ and $\mathcal{B}$

to implement a wide range of policies for accepting and executing I/O operations. $\mathcal{A}$ and $\mathcal{B}$ can attest $\mathcal{I}$ to be ensured that they use an I/O implementation suitable to implement their requirements.

Application jobs. The application enclaves $\mathcal{A}$ and $\mathcal{B}$ can be implemented as illustrated in Listing 2. A single function `worker` will use the functionality provided by $\mathcal{I}$ to acquire sensor readings, evaluate these readings, and, if necessary, queue a warning message with $\mathcal{I}$. We assume that $\mathcal{I}$ is programmed such that our $\mathcal{A}$ and $\mathcal{B}$ are guaranteed a free I/O buffer once per second, thus we do not handle the exception. Other applications, in particular code that is not enclaved, may not enjoy these guarantees and therefore need to handle the exception. The application then schedules a sleep of 1 s and is guaranteed to be woken up by the scheduler when this time period is elapsed, plus the scheduling margins summarized in Table 3. Note that our application does not make use of `clix` and is therefore interruptible. Making the execution of $\mathcal{A}$ and $\mathcal{B}$ entirely atomic is neither feasible (nested `clix` with $\mathcal{I}$ are not allowed) nor intended, as this would reduce the responsiveness of the overall system. However, even if $\mathcal{B}$ would deviate from the behavior in listing Listing 2 by performing a `clix` or causing a violation, this would not impact the security or availability of $\mathcal{A}$, which we discuss more comprehensively in Section 6.

Our case study shows that applications and drivers can be implemented such that, even in the presence of an uncooperative or malicious application that monopolizes system resources and maximizes delays, well-behaving protected applications make progress with deterministic latencies.

5.2 Performance Evaluation

One core performance metric of Aion implementations is the activation latency of applications. This activation latency is the time from when an application should be scheduled up to the time when control is actually passed over to it and it can start executing. In the following we consider the best and worst-case activation latencies for our prototype. An important characteristic of our prototype implementation is that any operation that the scheduler performs itself is atomic, *i.e.*, interrupts are disabled during scheduler execution so that the scheduler will not interrupt itself. In addition to regular scheduling, the scheduler also offers multiple operations to applications that return back to the caller or switch to another application. This means that activation latencies of application may be delayed by currently running scheduler operations. We first evaluate the performance of each scheduler operation in the best and worst-cases and use the results from this evaluation to perform an in-depth analysis of the activation latency of pending applications.

All timing overheads below are measured in CPU cycles and were retrieved through repeated measurements with the prototype implementation in a cycle-accurate simulation of Aion with Verilator [37]. Note that all performance numbers depend on the implementation of the trusted scheduler and show observed cycles only. Our prototype can only be seen as a baseline for real-world performance, that could be improved substantially with additional development effort.

Table 2: Detailed overhead in observed cycles for the operations provided by the scheduler.

Scheduler operation	Best case (cycles)	worst-case (cycles)
Create job	688	860
Exit job	512	736
Sleep	1124	1320
Yield	424	628
Get time	212	

Table 2 shows an overview over the timing overhead of all operations that applications can request from the scheduler. All scheduler operations are carefully designed to have a constant worst-case execution time. The remaining differences between best and worst-case execution time mostly depend on the amount of already scheduled or pending applications in the system. Since the prototype implementation places a sensible upper bound on the number of maximum running or sleeping applications, the worst-case execution times are strictly bounded and cannot be extended by adversaries. The longest operation that an adversary can attempt is to sleep while the maximum amount of other applications are already sleeping, which means that the scheduler needs to insert a new timer into a list of the maximum length. We observe a deterministic overhead of 1320 cycles for this operation.

Building on these first evaluation numbers, we craft an attacker that *(i)* enters an adversary-controlled enclave right before the victim deadline, *(ii)* executes a `clix` of the maximum length, and finally *(iii)* enters the scheduler with the worst-case sleep operation before the `clix` expires. At the end of the triggered scheduler operation, the scheduler will then detect the pending interrupt and process that interrupt instead of returning back to the adversary or another application. This represents the longest chain of events that an attacker can craft before a periodic enclave is executed. Table 3 shows the best and worst-case latencies that are possible for such an application deadline. In the absence of an attacker (*i.e.*, in the best case), interrupts are already enabled (*i.e.*, GIE=1) when the application is to be woken up, and the exception engine can process the interrupt immediately. In the presence of an attacker, however, the attacker would perform the sequence of steps as described above in order to delay the handling of the deadline. Since in our implementation, interrupts are disabled during scheduler operations, this prolongs the time until an interrupt is triggered by the time of the running operation. This bounds the worst-case latency between an issued interrupt and its actual processing in the scheduler by a maximum of 2330 cycles (10 cycles of atomic entry, 1000 of `clix` operation, followed by the 1320 cycles of the worst-case sleep operation). Note, that the adversary does not benefit from creating a violation during the last cycles of the `clix` instruction as a violation is also handled by the scheduler which can check whether other interrupts are currently pending before resuming execution of a job.

Processing the interrupt in hardware takes 7 cycles if an unprotected job is being interrupted, while interrupting enclaved jobs

Table 3: Detailed overhead for an event that preempts a running job. Shown are measurements with default Aion parameters and the overheads in the best and worst-case. Values in parentheses show the worst-case in the absence of an attacker and are zero for the crafted attacker scenario.

Task/Stage	Best case (cycles)	worst-case (cycles)
1. Interrupt arrival	0	10 + `clix` + 1320
2. Interrupt processing	7	(35)
3. Scheduler entry	157	(115)
4.1 Timer	1356	4075
4.2 Scheduler run	443	443
5 Scheduler resume	72	72
Activation latency	2035	5920 + `clix`

takes 35 cycles. The overhead stems from the additional work to store the CPU context in the enclave versus only storing the program counter and status register on the unprotected job's stack. This overhead is reversed on entering the scheduler for unprotected code (157 cycles) versus entering the scheduler after interrupting an enclave (115 cycles). In the crafted attacker scenario, the scheduler can detect the pending interrupt at the end of the running operation and before it would resume execution to the next application. Thus, in the worst-case, steps 2 and 3 are skipped by the scheduler as it can start processing the interrupt without needing to reenter itself.

In our prototype, processing a timer tick requires the processing of all software timers to evaluate whether a software timer is ready to be fired. This means that in the best case, no timer has to be processed, leading to a latency of 1356 cycles while in the worst-case, all 15 jobs currently have set a timer which leads to a latency of 4075 cycles. Identifying the next job to schedule takes a static duration of 443 cycles as periodic enclaves are always scheduled with the highest priority on the system. Resuming from the scheduler then takes 72 cycles.

Overall, our prototype can *guarantee* an activation latency of 2035 cycles in the best and 6920 cycles in the worst-case. This means that in the presence of an active adversary that controls all 14 other threads besides the victim thread and performs the sequence of steps as described above, our best-effort Aion prototype can guarantee that the first guaranteed application to be scheduled is served at the latest 6920 cycles after its trigger occurred. We discuss below what activation latency can be given to any application other than the first to be scheduled if multiple applications received guarantees simultaneously.

6 DISCUSSION AND SECURITY ANALYSIS

Confidentiality and integrity. Firstly, our reliance on TEEs and enclaved execution protects $\mathcal{A}$ and dependent code from a range of attacks on confidentiality and integrity. TEEs and their limitations are well understood in general [24] and for Sancus in particular [30]. For example, it is clear that enclaved applications must be developed such that they are free of vulnerabilities that allow an attacker to hijack the enclave's control flow or to extract secrets. The TCB reduction provided by TEEs helps to implement secure enclaves, relying on extensive code reviews, testing, and formal verification, which are orthogonal lines of research.

An important consideration to nuance the architectural confidentiality guarantees offered by TEEs is information leakage through software-exploitable side channels [14]. Fortunately, the class of light-weight embedded systems considered by Aion have a significantly reduced microarchitectural attack surface in comparison to notoriously complex x86 processors. In particular, known side-channel attacks on MSP430-Sancus platforms are mostly reduced to classic start-to-end timing [16], as well as more fine-grained interrupt latency timing attacks [40]. Side channels can generally be ruled out entirely by manually rewriting the application code to adhere to established constant-time programming best practices [14]. Alternatively, in the case of deterministic execution platforms such as MSP430, static code balancing solutions can provide an automated solution, either by transparently generating compensation code in the compiler backend [41] or statically analyzing execution path timings at the level of the generated assembly code [11, 35]. Finally, for the particularly relevant problem of interrupt latency timing side channels [40], recent work has proposed a provably secure, hardware-level padding defense for a simplified version of Sancus [9]. We leave integration of such architectural changes to further strengthen Aion against side channels as future work.

Guaranteed availability. Importantly, the activation latencies from Section 5.2 apply to $\mathcal{A}$ and $\mathcal{B}$ in our case study, even in the presence of strong software-level attackers who are capable of manipulating all software that is outside of the TCB. Specifically, we consider attackers that might attempt to *(i)* block the CPU by performing extensive uninterruptible computations, *(ii)* influence the scheduler to disrupt the execution of (other) jobs, *(iii)* block I/O resources through continuous use, or *(iv)* cause illegal memory access or atomicity violations.

First, considering attack *(i)*, a misbehaving or malicious enclave can try and prevent progress by using the `clix` instruction and potentially invoke a scheduler operation. This is limited to a fixed number of cycles after which the scheduler will serve pending interrupts and schedule other jobs. In Aion, `clix` and scheduler operations are the only means by which an application can prevent interruption. Importantly, `clix` periods cannot overlap to form continuous uninterruptible sections.

Alternatively, in attack *(ii)*, the attacker could try and schedule many short sleeps to maximize scheduling effort. We consider this attack in our evaluation and show that it has a substantial but still deterministic impact on the available CPU cycles for applications (cf. Section 5.2), and that the attack does not impact the baseline guarantees. The attack can be prevented by a scheduling policy where sleep requests below a certain threshold are not accepted, or where a misbehaving job is terminated.

In attack scenario *(iii)*, attackers try to continuously use an I/O resource. This can be ruled out by implementing `clix`-based atomic interactions with I/O drivers, which are followed by a scheduler interaction. As we have illustrated in our case study, it is feasible to program enclaved device drivers that either synchronously or asynchronously serve application enclaves, where the entry functions for applications have a bounded execution time and return within a single `clix`. This prevents the attacker from continuously blocking a resource and guarantees deterministic worst-case latencies for the next scheduling decision. Sancus's secure I/O functionality [30]

can be used to guarantee that no code other than the driver enclave has access to the memory addresses used to control the peripheral, excluding non-driver code to interfere with the peripheral.

Finally, considering attack *(iv)*, Aion's exception engine guarantees that all interrupts, including violations of platform policies, are handled by the scheduler and do not trigger a platform reset. The specifics under which jobs are scheduled and how violations are handled are subject to the protected scheduler implementation.

Aion provides real-time guarantees based on a deterministic worst-case latency that is followed by M cycles of progress. By means of specific scheduler implementations, more elaborate policies can be provided, including the "at least x % of the CPU cycles per interval T"-guarantee from Section 2.3. For this, the scheduler must *(a)* allow at most N jobs with availability guarantees, *(b)* implement round-robin scheduling among these N jobs, and *(c)* run on a platform where `clix` provides atomicity for M cycles. Then each of these N tasks is guaranteed to execute at least M CPU cycles (within a `clix`) of every $T = (5920 + M) + (1845 + M) * (N - 1)$ cycles. This is under the assumptions that $N - 1$ jobs are under attacker control, all attacker jobs are placed before the victim job in the round robin scheduling, and the attacker jobs all schedule a timer to be woken up together with the victim. Furthermore, each attacker job executes a maximum `clix` for M cycles, which ends in a scheduler invocation where the job schedules a timer for the next period. Thus, the first scheduled job experiences the above calculated worst case delay while the scheduler will only need to perform steps 1, 4.2, and 5 from Table 3 for the remaining jobs. For our prototype implementation with 15 allowed jobs and a `clix` length of 1000, the absolute worst-case activation latency for the last-scheduled victim job is $6920 + 2845 * 14 = 46750$ cycles. This represents the absolute worst-case where the platform developer decided to provide the same guarantees to 14 attacker jobs other than the victim job and it shows that our system can give deterministic guarantees based on highly flexible platform configurations.

Using attestation. Applications include dependent code in their TCB, e.g. device drivers or the scheduler, and trust these for availability. The trustworthiness of this code is to be established by validation techniques beyond the scope of this paper. Remote attestation of the application enclave, together with Sancus's secure linking mechanism [30], give the deployer the guarantee that the application is indeed executing on a platform with the intended properties. For this, the scheduler and I/O drivers must be provided as enclaves and implement scheduling and access policies in code, the identity of which is then part of the attestation procedure. Enclaves can make use of mutual attestation and rely on enclave IDs to identify each other and provide specific guarantees, such as the availability of output buffers for $\mathcal{A}$ and $\mathcal{B}$ in the case study.

Our case study illustrates these features in a rather static scenario and based on fixed enclave IDs. To enable the open system that we describe in this paper, where protected applications can be deployed at any time and without a platform reset, applications and driver enclaves need to provide APIs that allow an application to, e.g., request a guaranteed I/O buffer, and to communicate success or failure to the deploying stakeholder after the initial attestation. This allows the deployer to ascertain schedulability of a deployment.

The latter approach also enables the use of I/O devices that require more complex access policies and that cannot complete an I/O operation atomically. For instance, a sensor might need to be calibrated for a specific use and multiple applications may require different calibrations. We believe that the Aion design is flexible enough to integrate adequate access logic for such scenarios into driver enclaves, yet our MSP430-Sancus platform, being a very light-weight 16-bit processor, has limitations regarding the implementation size of application and driver logic.

Summary, limitations, and future work. As a result of our joint spatial and temporal isolation, an application's security is no longer impacted by faults in other applications. Specifically, vulnerabilities in $\mathcal{B}$ may lead to $\mathcal{B}$ being compromised, and scheduling faults caused by $\mathcal{B}$ may lead to the termination of $\mathcal{B}$. But, these events do not affect the security and availability of $\mathcal{A}$, and vice versa. Importantly, Aion does not impose a hierarchy of trust or criticality on applications. We enable multiple mutually distrusting and non-collaborative applications that operate at the same "priority" to execute under equally strong security and availability guarantees.

We consider a hardware attacker with the ability to arbitrarily trigger external interrupts to be out of scope for Aion. However, a platform where the scheduler is capable of temporarily masking these interrupts or disabling interrupts completely, would be able to resist these attacks. We note, however, that this could compromise the trusted time guarantees of the scheduler if a timer tick is missed.

A specific challenge of Aion comes with the use of cryptographic operations for attestation or secure communication, which may take many CPU cycles to complete. Sancus [30] implements these operations in hardware for reasons of security and performance. While Aion makes cryptographic operations interruptible, the state of the cryptographic engine is lost upon interruption and the operation needs to be restarted entirely. Therefore, these operation complicate timing analysis and may prevent applications from making progress if they cannot complete a cryptographic operation within a `clix`. There are several ways to address these issues, e.g. by making the crypto engine resumable, tuning the semantics of `clix` to specific progress requirements, or relying on cryptographic operations in software, which we will investigate in future work.

7 CONCLUSION

We presented Aion, a configurable security architecture that can preserve real-time availability guarantees for embedded systems even in the presence of a strong software attacker. This set of guarantees is especially of interest for open systems that execute arbitrary dynamically deployed code from multiple, mutually distrusting stakeholders which all request their same fair access to resources. Aion is the first design for modern TEE architectures that provides a strong notion of trusted scheduling, derived from preemption, bounded atomicity, and an enclaved scheduler. We implemented and evaluated a prototype of Aion on a light-weight TEE and conclude that our system can deterministically guarantee a worst-case latency of 6920 cycles until a protected job is scheduled. This allows platform developers to derive more complex scheduling policies that can enable a future class of truly open IoT systems.

ACKNOWLEDGMENTS

This research is partially funded by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and by a gift from Intel Corporation. Fritz Alder and Jo Van Bulck are supported by a grant of the Research Foundation – Flanders (FWO). Specific funding was provided under the SAFETEE project by the Research Fund KU Leuven.

REFERENCES

[1] Tiago Alves and Don Felton. 2004. TrustZone: Integrated hardware and software security. *ARM white paper* 3, 4 (2004), 18–24.

[2] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 90–102.

[3] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. 2018. RIOT: An open source operating system for low-end embedded devices in the IoT. *IEEE Internet of Things Journal* 5, 6 (2018), 4428–4440.

[4] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*. IEEE, 79–80.

[5] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *Proceedings of the 30th USENIX Security Symposium*.

[6] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. 2011. Timing analysis of a protected operating system kernel. In *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 339–348.

[7] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: Tiny trust anchor for tiny devices. In *Design Automation Conference (DAC 2015)*. IEEE, 1–6.

[8] Alan Burns and Robert Davis. 2019. Mixed criticality systems – a review. *Department of Computer Science, University of York, Tech. Rep* (2019), 1–81.

[9] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. 2020. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *33rd IEEE Computer Security Foundations Symposium (CSF'20)*.

[10] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, and Ahmad-Reza Sadeghi. 2019. CHASE: A Configurable Hardware-Assisted Security Extension for Real-Time Systems. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[11] Florian Dewald, Heiko Mantel, and Alexandra Weber. 2017. AVR Processors as a Platform for Language-Based Security. In *European Symposium on Research in Computer Security (ESORICS)*. 427–445.

[12] Pan Dong, Alan Burns, Zhe Jiang, and Xiangke Liao. 2018. Tzdks: A new trustzone-based dual-criticality system with balanced performance. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 59–64.

[13] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust.. In *NDSS*, Vol. 12. Internet Society, 1–15.

[14] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27.

[15] Olivier Girard. 2009. openMSP430 – a synthesizable 16-bit microcontroller core written in Verilog. https://opencores.org/project,openmsp430.

[16] Travis Goodspeed. 2008. Practical attacks against the MSP430 BSL. In *Twenty-Fifth Chaos Communications Congress*.

[17] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 653–669.

[18] Intel Corporation. 2020. *Intel 64 and IA-32 architectures software developer's manual – Volume 3D: System programming guide, part 4.* Reference no. 332831-072US.

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

[20] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, Article 10, 14 pages.

[21] Dayeol Lee, David Kohlbrenner, Kevin Cheang, Cameron Rasmussen, Kevin Laeufer, Ian Fang, Akash Khosla, Chia-Che Tsai, Sanjit Seshia, Dawn Song, and Krste Asanovic. 2018. Keystone: Open-source Secure Hardware Enclave. https://keystone-enclave.org/.

[22] Songran Liu, Nan Guan, Zhishan Guo, and Wang Yi. 2020. MiniTEE: A Lightweight TrustZone-Assisted TEE for Real-Time Systems. *Electronics* 9, 7 (2020).

[23] Anna Lyons and Gernot Heiser. 2014. Mixed-criticality support in a high-assurance, general-purpose microkernel. In *Workshop on Mixed Criticality Systems*. 9–14.

[24] Pieter Maene, Johannes Götzfried, Ruan De Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. 2017. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Trans. Comput.* 67, 3 (2017), 361–374.

[25] Ramya Jayaram Masti, Claudio Marforio, Aanjhan Ranganathan, Aurélien Francillon, and Srdjan Capkun. 2012. Enabling trusted scheduling in embedded systems. In *Annual Computer Security Applications Conference (ACSAC)*.

[26] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 10:1–10:1.

[27] Anway Mukherjee, Tanmaya Mishra, Thidapat Chantem, Nathan Fisher, and Ryan Gerdes. 2019. Optimized trusted execution for hard real-time applications on COTS processors. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*. 50–60.

[28] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium*. USENIX Association, 479–494.

[29] Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2017. Authentic execution of distributed event-driven applications with a small TCB. In *STM '17 (LNCS)*, Vol. 10547. Springer, Heidelberg, 55–71.

[30] J. Noorman, J. Van Bulck, J. Tobias Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. 2017. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)* 20, 3 (2017), 7:1–7:33.

[31] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1429–1446.

[32] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 2 (2015).

[33] Sandro Pinto, Tiago Gomes, Jorge Pereira, Jorge Cabral, and Adriano Tavares. 2017. IIoTEED: an enhanced, trusted execution environment for industrial IoT edge devices. *IEEE Internet Computing* 21, 1 (2017), 40–47.

[34] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. 2016. Towards a TrustZone-assisted hypervisor for real-time embedded systems. *IEEE computer architecture letters* 16, 2 (2016), 158–161.

[35] Sepideh Pouyanrad, Jan Tobias Mühlberg, and Wouter Joosen. 2020. SCF MSP: static detection of side channels in MSP430 programs. In *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES)*. 1–10.

[36] Thomas Sewell, Felix Kam, and Gernot Heiser. 2017. High-assurance timing analysis for a high-assurance real-time operating system. *Real-Time Systems* 53, 5 (2017), 812–853.

[37] Wilson Snyder. 2020. Verilator, the fastest Verilog/SystemVerilog simulator. https://www.veripool.org/wiki/verilator.

[38] Raoul Strackx, Frank Piessens, and Bart Preneel. 2010. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*. Springer, 344–361.

[39] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. 2019. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS'19)*. ACM.

[40] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*. ACM.

[41] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. 2021. Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks. In *EuroS&P '21*. IEEE, Washington, DC, USA.

A ATOMICITY STATE MACHINE

Figure 6 shows the complete atomicity state machine with all edge cases.

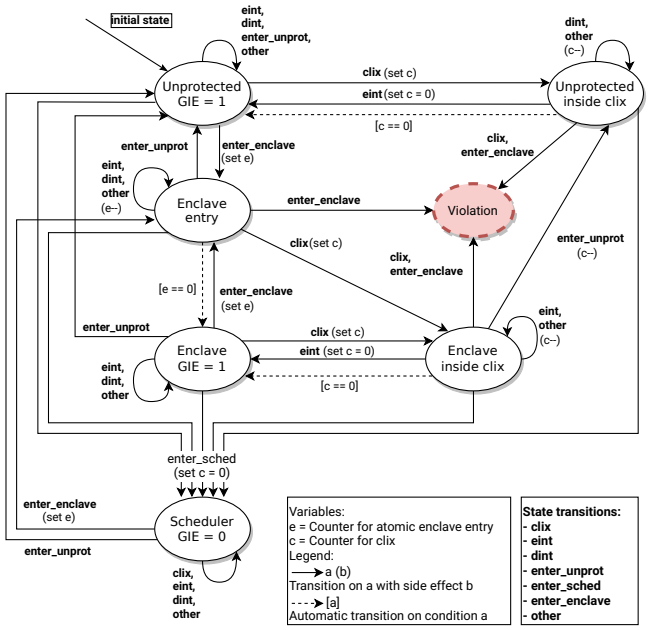

Figure 6: Atomicity state machine showing explicit and implicit transitions from unprotected, enclaved, and scheduler states. Note, that interrupt transitions are not explicitly shown in this figure but can be interpreted as `enter_sched` transitions.

B CASE STUDY SOURCE CODE IN C

```c
#include <msp430.h>
#include "uart.h"
#include "uart_hardware.h"
#include <stdio.h>
#include "kernel_defines.h"
#include "secure_mintimer.h"
#include "log.h"
#include "sancus_helpers.h"

#define ___MACRO_CLIX(clix_length)  \
    __asm__("push r15");    \
    __asm__("mov.w %0, r15" : : "i"(clix_length));    \
    __asm__(".word 0x1389");    \
    __asm__("pop r15");

#define _HAVE_APPA            1
#define _HAVE_APPB            1
#define _HAVE_APP_SLEEP       1
#define _HAVE_IO_THREAD       1

/* --- IO Enclave ---------------------------------------------- */
DECLARE_SM(ioenclave,  0x1234);

#ifdef _HAVE_IO_THREAD
#define IO_BUFS       4
SM_DATA(ioenclave) unsigned char io_bufs[IO_BUFS] = {0, 0, 0, 0};
SM_DATA(ioenclave) bool io_ready[IO_BUFS] = {false, false, false,
     false};
#endif

// Output
bool SM_ENTRY(ioenclave) io_uart_write_byte(unsigned char b)
{
#ifdef _HAVE_IO_THREAD
    // Async I/O
    ___MACRO_CLIX(50);
    int caller = (int) sancus_get_caller_id();
    if (!caller || caller >= IO_BUFS) { caller = 0; }
    if (io_ready[caller]) {
      return (false);
    } else {
      io_bufs[caller]  = b;
      io_ready[caller] = true;
      return (true);
    }
#else
    // Sync I/O
    ___MACRO_CLIX(30);
    while (UART_STAT & UART_TX_FULL) {}       // !!
    UART_TXD = b;
    return (true);
#endif
}

// Read sensor
uint64_t SM_ENTRY(ioenclave) io_get_reading(void)
{
    ___MACRO_CLIX(30);
    return (secure_mintimer_now_usec64());
}

#ifdef _HAVE_IO_THREAD
static char sm3_unprotected_stack[THREAD_EXTRA_STACKSIZE_PRINTF];
// Async I/O thread
void SM_ENTRY(ioenclave) io_thread(void)
{
    while (true) {
      // this could implement *any* policy.
      for (int i = 0; i < IO_BUFS; i++) {
        if (io_ready[i]) {
          ___MACRO_CLIX(30);
          while (UART_STAT & UART_TX_FULL) {} // !!
          UART_TXD = io_bufs[i];
          io_ready[i] = false;
        }
      }
#ifdef _HAVE_APP_SLEEP
      ___MACRO_CALL_SLEEP_FROM_SM(0x0100, 0x0001, ioenclave)
#endif
    }
    return;
}
#endif

/* --- APP A ---------------------------------------------- */
#ifdef _HAVE_APPA
static char sm1_unprotected_stack[THREAD_EXTRA_STACKSIZE_PRINTF];
DECLARE_SM(appa,       0x1234);

SM_DATA(appa) uint64_t reading_a = 0;

void SM_ENTRY(appa) a_entry(void)
{
    printf2("A: ID %d, called by %d\n",
        sancus_get_self_id(), sancus_get_caller_id());

    while (true) {
      reading_a = io_get_reading();
      printf1("A: t is %lu\n", reading_a);
      if (reading_a >= 50000) { io_uart_write_byte('A'); }
#ifdef _HAVE_APP_SLEEP
      ___MACRO_CALL_SLEEP_FROM_SM(0x0100, 0x0001, appa)
#endif
    }
}
#endif
```

```
107
108 /* --- APP B ---------------------------------------------------- */
109 #ifdef _HAVE_APPB
110 static char sm2_unprotected_stack[THREAD_EXTRA_STACKSIZE_PRINTF];
111 DECLARE_SM(appb,       0x1234);
112
113 SM_DATA(appb) uint64_t reading_b = 0;
114
115 void SM_ENTRY(appb) b_entry(void)
116 {
117     printf2("B: ID %d, called by %d\n",
118         sancus_get_self_id(), sancus_get_caller_id());
119
120     while (true) {
121       reading_b = io_get_reading();
122       printf1("B: t is %lu\n", reading_b);
123       if (reading_b >= 50000) { io_uart_write_byte('B'); }
124 #ifdef _HAVE_APP_SLEEP
125       ___MACRO_CALL_SLEEP_FROM_SM(0x0100, 0x0001, appb)
126 #endif
127     }
128 }
129 #endif
130
131 /* --- Unprotected Job Creation ------------------------------- */
132 int main(void)
133 {
134     LOG_INFO("######## Riot on Sancus\n");
135     LOG_INFO("Case study with same prio levels\n");
136
137     while(sancus_enable(&ioenclave) == 0);
138 #ifdef _HAVE_APPA
139     while(sancus_enable(&appa) == 0);
140 #endif
141 #ifdef _HAVE_APPB
142     while(sancus_enable(&appb) == 0);
143 #endif
144
145 #ifdef _HAVE_APPA
146     thread_create_protected(
147         sm1_unprotected_stack,           // Unprotected stack for
      OCALLS
148         THREAD_EXTRA_STACKSIZE_PRINTF,   // size of the
      unprotected stack
149         1,                               // Priority to give
150         THREAD_CREATE_WOUT_YIELD,        // Thread create flag
151         SM_GET_ENTRY(appa),              // SM Entry address
152         SM_GET_ENTRY_IDX(appa, a_entry), // SM IDX address
153         "A");                            // Name for console
      logging
154 #endif
155 #ifdef _HAVE_APPB
156     thread_create_protected(
157         sm2_unprotected_stack,
158         THREAD_EXTRA_STACKSIZE_PRINTF,
159         1,
160         THREAD_CREATE_WOUT_YIELD,
161         SM_GET_ENTRY(appb),
162         SM_GET_ENTRY_IDX(appb, b_entry),
163         "B");
164 #endif
165 #ifdef _HAVE_IO_THREAD
166     thread_create_protected(
167         sm3_unprotected_stack,
168         THREAD_EXTRA_STACKSIZE_PRINTF,
169         1,
170         THREAD_CREATE_WOUT_YIELD,
171         SM_GET_ENTRY(ioenclave),
172         SM_GET_ENTRY_IDX(ioenclave, io_thread),
173         "IO");
174 #endif
175
176     LOG_INFO("Thread initialization done\n");
177     while(true){
178         secure_mintimer_usleep(300000);
179     }
180     LOG_INFO("Exiting main thread by shutting down CPU\n");
181     sched_shut_down();
182
183     UNREACHABLE();
184     return 0;
185 }
```

Listing 3: Source code of our case study implementation in C. Note that two while-loops in l.48 and l.71 do have deterministic execution time unless there is a hardware fault.