Towards Availability and Real-Time Guarantees for Protected Module Architectures

Jo Van Bulck Job Noorman Jan Tobias Mühlberg Frank Piessens
iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium
{jo.vanbulck, job.noorman, jantobias.muehlberg, frank.piessens}@cs.kuleuven.be

Abstract

Protected Module Architectures are a new brand of security architectures whose main objective is to support the secure isolated execution of software modules with a minimal Trusted Computing Base (TCB) – several prototypes for embedded systems (and also the Intel Software Guard eXtensions for higher-end systems) ensure isolation with a hardware-only TCB. However, while these architectures offer strong confidentiality and integrity guarantees for software modules, they offer no availability (let alone real-time) guarantees. This paper reports on our work-in-progress towards extending a protected module architecture for small microprocessors with availability and real-time guarantees. Our objective is to maintain the existing security guarantees with a hardware-only TCB, but to also guarantee availability (and even real-time properties) if one can also trust the scheduler. The scheduler, as any software on the platform, remains untrusted for confidentiality and integrity – but it is sufficient to trust the scheduler module to get availability guarantees even on a partially compromised platform.

Categories and Subject Descriptors D.4.7 [*Operating Systems Organization and Design*]: Real-time systems and embedded systems

General Terms Design, Security

Keywords Protected module architecture, secure scheduling, secure interrupt, mixed-criticality, real-time operating system, trusted computing

1. Introduction

Small embedded devices are becoming omnipresent and interconnected in our everyday lives. Through the rise of wireless sensor networks, ubiquitous computing and the Internet of Things, lightweight extensible platforms are increasingly entrusted with safety-critical and privacy-sensitive tasks. Yet, to minimize production costs and power consumption, these devices commonly lack hardware support for established security measures, such as virtual memory and processor privilege levels. They generally operate in a *single-address-space* where memory is treated as a global resource, addressable and accessible by everyone. Software running on these platforms is thus exposed to modification by malicious or buggy programs.

MASS'16, March 14, 2016, Málaga, Spain

DOI: http://dx.doi.org/10.1145/2892664.2892693

To address these concerns, recent research on Protected Module Architectures (PMAs) [10, 18, 25] proposes low-cost hardware extensions that enable the secure and isolated execution of software modules by means of fine-grained memory access control. These architectures support the creation of so-called Self-Protecting Modules (SPMs) that define a contiguous code and data section in the shared address space. Security guarantees are based on the property that an SPM's private data section is exclusively managed via its corresponding code section, which can only be entered via a few predefined entry points. We discuss PMAs in more detail in Section 2.

An interesting aspect of PMAs is that it has been shown [1, 20] that source code can be compiled to a PMA while maintaining source code level abstractions, even against attackers that can operate at machine code level. In other words, if a source code module encapsulates data using source code modularization mechanisms such as Java classes, then secure compilation of that source code to a PMA ensures that the encapsulation is maintained at runtime with respect to arbitrary (possibly even malicious) other machine code modules. For hardware-level PMAs, these guarantees even hold against attackers that compromise the Operating System (OS).

However, the security guarantees offered by current PMAs are limited to confidentiality and integrity guarantees – they do not extend to *availability*. A buggy or malicious application can still harm the availability of the platform by overwriting crucial OS data structures, or by monopolizing a shared system resource such as CPU time. This paper reports on our work-in-progress towards lifting this limitation. We argue that in addition to isolating software, hardware-level protection mechanisms can be extended to also preserve availability (possibly even real-time) guarantees on a partially compromised embedded system.

The objective of this paper is to identify some of the availability challenges induced by hardware-level PMAs, and to report on our work-in-progress to address them while keeping the Trusted Computing Base (TCB) small. More specifically, we make the following contributions:

- We outline a hardware mechanism that makes SPMs fully interruptible and reentrant. We show how our mechanism preserves deadlines for external events, and facilitates reasoning about real-time guarantees by ensuring a deterministic interrupt latency at all times.
- We sketch a multitasking model that introduces protection domains *within* a conventional control flow thread. We show how logical threads can be managed by an unprivileged scheduler, and we discuss new challenges regarding secure compilation and asynchronous communication.

The remainder of this paper is structured as follows. First, in Section 2, we briefly revisit existing embedded PMAs, and we

formulate our objectives in Section 3. Next, Section 4 elaborates on the hardware and software changes needed to support availability and real-time guarantees, where the real-time deadlines of one stakeholder are unaffected by the actions of another. Finally, we discuss related work in Section 5 and conclude in Section 6.

2. Background: Protected Module Architectures

As mentioned in the introduction, a recent line of research [10, 18, 23, 25] investigates an alternative memory access model for lightweight embedded microcontrollers. Since we build our work upon the Sancus [18] platform, this section provides a more detailed overview of general PMA concepts and Sancus' internals.

While these PMAs have a striking resemblance to similar architectures for higher-end systems, most notably the Intel Software Guard eXtensions (SGX) [16], there are also important differences. In this paper, we focus strictly on lightweight embedded PMAs.

2.1 Self-Protecting Modules

An SPM corresponds to a fine-grained protection domain in a shared address space, and is defined by two contiguous memory regions: a public *text* or *code section* containing a fixed number of *entry points* and a private *data section*. PMAs ensure that memory access to the private data section of a module is only allowed when the program counter is pointing within the module's corresponding code section. Attackers from the outside can never directly access the protected data; an SPM is solely responsible for its own private data section, hence the name *self-protecting* modules.

To ensure that control flow within an SPM protection domain cannot be influenced from the outside, an SPM can only be entered via a few predefined entry points and should maintain its own private call stack. Table 1 summarizes the memory access rights for an SPM. All memory addresses are categorized as either "protected", that is belonging to the SPM, or "unprotected". Protected memory is further subdivided as belonging to an entry point, a text section or a data section. Each entry indicates how code executing in the "from" section may access the "to" section. Note that when deciding access to an SPM the program counter is only considered "protected" when corresponding to an address within the code section of the *same* module (i.e., any other module is treated as if it were unprotected).

Table 1. Program counter based access control rules for an SPM in the traditional UNIX notation.

From \ to	Protected			Unprotected
	Entry	Text	Data	
Protected	`r-x`	`r-x`	`rw-`	`rwx`
Unprotected/ Other SPM	`r-x`	`r--`	`---`	`rwx`

2.2 Sancus and Embedded PMAs

The Sancus [18] architecture extends the memory access logic and instruction set of a low-end TI MSP430 microcontroller to support software module protection at hardware-level. Sancus explicitly targets low-end platforms featuring runtime extensibility by several mutually distrusting parties. To this end, Sancus allows an external software provider to remotely deploy an SPM, without having to trust any of the host software on the embedded device. Through a process known as *remote attestation*, the software provider is given the assurance that their software is loaded on a specific device, without being tampered with. When enabling protection for a newly loaded module, the Sancus hardware calculates a symmetric cryptographic key based on the contents of that module's text section. The remote software provider obtains an identical key through a key-derivation scheme, so that he or she is provided with an authenticated confidentiality- and integrity-preserving communication channel to the deployed module.

Sancus also supports the *secure linking* of SPMs residing on the same device. That is, a dedicated hardware instruction enables an SPM to cryptographically verify the integrity of another module, before jumping to its entry point. Likewise, a newly entered SPM can verify the identity of its caller. To speed up subsequent authentications and enable the implementation of access control policies, the Sancus hardware assigns a unique unforgeable spmID to each module, and offers specialized instructions to query the spmID of the caller or another module.

The Sancus distribution[1] comes with a dedicated C compiler to automate the process of creating SPMs. For this, the compiler inserts short `spm_entry` and `spm_exit` assembly code stubs in the text section of every SPM. These code stubs are executed whenever a module is entered or exited, and take care of secure linking, private call stack switching, clearing of CPU registers, and multiplexing multiple logical entry points through a single physical entry point.

While Sancus hard codes the memory access rules in a dedicated combinational Memory Access Logic circuit per SPM, the TrustLite [10] architecture features an Execution-Aware Memory Protection Unit (EA-MPU) that records program counter based memory access rules in a configurable hardware table. The TyTAN [2] security architecture employs a trusted software layer that reconfigures TrustLite's EA-MPU table at runtime to provide dynamic loading, and local and remote attestation guarantees. We compare our approach in detail to these proposals in Section 5.

3. Objectives

The above way of isolating software in hardware-enforced protection domains allows for strong security guarantees in a dynamic multi-stakeholder context, but does not address the issue of availability on a partially compromised system. Existing embedded security architectures either put availability explicitly out of scope [6, 18, 25], or rely on an omnipotent software TCB [2, 13]. The integrity of such a trusted software layer should in turn be attested to the remote stakeholder, and protected against an attacker capable of arbitrary code execution in the shared address space. Moreover, a trusted software component may be vulnerable to well-known [7] low-level software security attacks. We therefore argue that mixed-criticality embedded systems will benefit from real-time availability guarantees, imposed without invalidating the property that confidentiality and integrity are protected with a hardware-only TCB.

More specifically, via careful hardware/software co-design we aim to develop a security architecture with the following central features:

Zero-Software TCB. An SPM should solely rely on its own implementation and trusted hardware for the integrity and confidentiality of its internal state.

Real-Time Compliance. Strict real-time deadlines for external events should be met, even while an adversary is executing arbitrary code on the platform. For this, our system should ensure a deterministic worst-case *interrupt latency* (i.e., the amount of time before an interrupt is serviced) at all times.

Secure Multitasking. We will extend the notion of secure linking to introduce protection within a conventional control flow thread. An *unprivileged* preemptive priority-based scheduler will be able to provide such threads with strong availability guarantees, independent from attackers executing at a lower priority level.

[1] `https://distrinet.cs.kuleuven.be/software/sancus/`

4. Working Towards Availability and Real-Time Guarantees

In the following, we report on our ongoing work in leveraging PMA concepts to not only guarantee the untampered execution of hardware-enforced software modules, but also assure their timely execution. We first outline a secure hardware interrupt mechanism. Next, we introduce a multitasking model where isolated threads can be managed by an untrusted scheduler.

4.1 Interruptible Isolated Execution

Existing hardware-level PMAs such as SMART [6] and Sancus [18] protect the runtime state of a software module, under the explicit assumption that interrupts are disabled during SPM execution. This constraint seems unreasonable for mission-critical embedded systems that should remain responsive to external stimuli at all times. For such systems, interruptible protection domains have been proposed [2, 13] via a trusted software layer that saves the internal execution context of a task before passing control to an untrusted Interrupt Service Routine (ISR). Alternatively, researchers [5, 10, 16] have implemented secure hardware interrupt engines that preserve a zero-software TCB in the presence of interrupts.

To the best of our knowledge however, no hardware interrupt mechanism so far considers the combination of (*i*) multiple SPMs, (*ii*) multithreading within an SPM, and (*iii*) atomicity constraints. In the following, we present our progress towards addressing these challenges. First, we explain how we realize the autonomous operation of SPMs by automatically invoking them in response to an interrupt. Next, we discuss recent work on a secure hardware interrupt mechanism that makes SPMs fully interruptible and re-entrant. Finally, we outline our planned work on developing compiler measures and a hardware atomicity monitor to address the security implications of preemption on secure SPM execution.

4.1.1 Interrupt-Serving SPMs

We enable secure interrupt handling by registering an SPM's entry address in the system-wide Interrupt Vector Table (IVT). Several specifics should be considered when doing so. First, we want to protect the read-only nature of the in-memory IVT. In the existing Sancus prototype, this can be easily achieved by wrapping the IVT memory region in the text section of a dedicated SPM. Second, to ensure an interrupt-serving SPM is only invoked in response to a specific Interrupt Request (IRQ), we extended Sancus' hardware primitive for caller authentication with unforgeable IRQ identifiers. Finally, we will enable the use of non-maskable interrupts on the microcontroller (cf. Section 4.1.3). Together these measures ensure the untampered and guaranteed execution of an ISR, without having to trust any infrastructural software on the embedded device. Such guarantees may for example reinforce recent research on *trust assessment modules* [17] where a protected module periodically measures the trustworthiness of an untrusted execution environment.

The above approach leaves several research questions open, especially when considering multiple protected ISRs that might interrupt each other. To avoid complications resulting from such interrupt nesting, we will investigate hardware changes that assure run-to-completion semantics for ISRs, while still limiting their effective length. We will also consider alternative, more fine-grained hardware-assisted mechanisms to protect the integrity of the in-memory IVT, facilitating flexible ISR (un)registering at runtime.

4.1.2 Interruptible SPMs

Being able to interrupt and resume SPMs without requiring their cooperation ensures responsiveness and enables the implementation of preemptive real-time scheduling. We have modified the interrupt logic of a Sancus-enabled MSP430 microcontroller to ensure that, even in the presence of interrupts, an SPM solely relies on trusted hardware for the integrity and confidentiality of its internal state. More specifically, our IRQ logic (*i*) pushes the program counter, status register, and general-purpose CPU registers on the interrupted SPM's private call stack, (*ii*) stores the stack pointer on a fixed location in the private data section, (*iii*) clears the CPU registers, and (*iv*) vectors to the appropriate untrusted ISR. Importantly, the hardware IRQ logic is subject to the same memory access checks as normal user code. When encountering a memory violation, the logic will immediately shortcut to step (*iii*), assuring ISR execution – at the cost of a non-resumable (misbehaving) interrupted SPM. This is especially relevant in a multi-module context where a malicious or buggy SPM might change the stack pointer register, or overflow its internal stack into the private data section of another SPM. To make the interruption of SPMs fully transparent to the programmer, we rely on Sancus' compiler-generated assembly code stubs. The `spm_entry` stubs of our current prototype automatically restore internal execution context on the next invocation of a previously interrupted SPM.

Some existing embedded security architectures feature similar interruptible protection domains. The TrustLite [10] security architecture includes a comparable hardware exception engine for hardware-enforced software modules. Recent work by De Clercq et al. [5] implements secure hardware-level interrupts for a Sancus-like architecture with two protection domains. We anticipate more challenges when dealing with multithreaded SPMs (cf. Section 4.2), as multiple internal stack pointers should then be considered. Moreover, none of the existing approaches consider denial-of-service attacks where an untrusted ISR breaks the integrity of the `reti` (return from an ISR) control flow, or refuses to return at all. We therefore plan to extend our hardware interrupt mechanism to keep track of the entry point of the previously interrupted module. Genuine ISR exit can then be asserted through a modified `reti` instruction that jumps to this address. Furthermore, we will limit the length of an untrusted ISR by automatically invoking a `reti` instruction when a certain number of execution cycles was exceeded.

4.1.3 Atomicity Constraints

Our architecture will support the atomic execution of small critical code sections. Apart from application-specific atomicity constraints, we will employ atomic sections to preserve security guarantees for interruptible isolated execution. A newly-entered SPM can for example not be interrupted before it has (*i*) stored the address and hardware spmID of its caller (to enable subsequent caller authentication), and (*ii*) restored its internal stack pointer. Moreover, to avoid well-known time-of-check-to-time-of-use attacks [23, 25] when calling into an SPM, authentication of the SPM and jumping to its entry point should happen atomically. If not, the SPM might be unloaded in the meantime. Existing security architectures work around these issues by either not allowing SPMs to be (un)loaded at runtime [10], requiring their uninterrupted execution [18, 24], or relying on an omnipotent trusted software entity to mediate all inter-module communication [2].

Our architecture on the other hand will allow SPMs to call each other directly, without trusting external software, and without invalidating their security properties. From an availability perspective, we will prevent an attacker from abusing our atomicity mechanism to hold on to the CPU. Inspired by related work [13], we plan to develop a hardware atomicity monitor that limits the maximum duration of an atomic section and prevents their nesting. We expect such a bounded execution time for atomic sections to facilitate reasoning about real-time guarantees and the schedulability of the system, as it enables a deterministic interrupt latency at all times.

Using our atomicity primitive, we plan to develop automatic compiler measures that allow a programmer to write SPMs that can

be safely executed in an existing preemptive scheduling context. We expect our atomic solution to be sufficiently powerful to realize secure compilation, but further research is needed to evaluate the (application-specific) expressiveness limitations of strictly bounded and unnested critical sections. For atomic sections that run across multiple interacting SPMs, we will look into the influence of an (untrusted) caller on the atomic behavior of a callee.

4.2 Real-Time Secure Multitasking

At some point in the complexity curve, embedded applications will benefit from a (real-time) OS. This in mind, we are developing an *unprivileged* scheduler that manages protected SPM threads. By building upon the isolation and secure interrupt hardware primitives, we ensure the scheduler's guaranteed and untampered execution. As such, the scheduler software component will enable multiple distrusting parties to share an embedded computing platform, without loosing their (real-time) availability guarantees.

We now describe a multithreading model where isolated, protected threads may run through multiple fine-grained SPM protection domains. Next, we show how such isolated threads may be interleaved by an unprivileged scheduler. Finally, we outline future work on asynchronous Inter Process Communication (IPC) to allow secure threads to communicate with untrusted threads.

4.2.1 Multithreading Model

Developing a suitable concurrency model for PMAs can be regarded as an open research question of its own. Classical C-style threading is the predominant paradigm in embedded systems, as well as in existing PMA implementations. We will therefore primarily focus on secure multithreading, but ideas from other concurrency models such as reactive or continuation passing style programming are to be considered as well.

A thread traditionally represents an activity within the address space (protection domain) of a process. When implementing threading in a single-address-space, it is important to understand that the unit of execution does not necessarily correspond to the unit of fine-grained memory protection. Instead, logical control flow "threads" can be constructed by allowing SPMs to authenticate and call each other directly. Securely linking multiple such interacting modules in a control flow thread is however non-trivial, as the call stack is now divided over different SPM-private call stacks and the integrity of the inter-module control flow should be preserved. The TrustLite [10] and TyTAN [2] architectures tend to avoid these issues by considering SPMs as stand-alone "secure tasks" that are only allowed to communicate indirectly via message-passing IPC. The complexity of such secure tasks is increased, as they should internally queue IPC requests, and may rely on an intermediate trusted software entity [2] to supervise IPC, inducing considerable runtime overhead and an enlarged TCB. Moreover, without further consideration, synchronous IPC between separately schedulable SPMs can introduce unbounded priority inversion where a high-priority task waits for a reply from a low-priority service that is interrupted by a medium priority task.

We therefore argue that logical "threads", jumping from module to module, are the better option to represent (synchronous) control flow in a single-address-space. We regard SPMs as an *execution context* for threads, defining their current memory access rights and private call stack. The same schedulable entity may traverse multiple SPM protection domains during its lifetime. Our prototype transparently preserves the integrity of inter-module control flow through the aforementioned compiler-generated `spm_entry` assembly code stubs. Participating SPMs guard the entry of their protection domain to ensure the integrity of inter-SPM calls/returns. For this, we rely on Sancus' unforgeable spmIDs and the hardware primitives for caller/callee authentication, introduced in Section 2.2.

When interleaving the execution of multiple logical threads, participating SPMs have to be made *threading-aware*. That is, they should not accumulate the execution contexts of different logical threads on the same private call stack. To this end, our protected scheduler module (discussed below) assigns a unique thrID to each logical thread, and provides an entry point to retrieve the thrID of the currently executing thread. This scheme enables the compiler-generated `spm_entry` procedure to properly separate internal execution contexts on different call stacks. In case the SPM runs out of internal call stacks, a failure indicator is returned via an agreed CPU register. Our current prototype only features a single private call stack for each module, but our thrID mechanism should scale to multithreaded SPMs.

We plan to further develop our multithreading model and reason about its security and availability guarantees. Logical threads will be provided with strong, real-time availability guarantees, as long as they do not include unprotected code, and authenticate and trust all participating SPMs before calling them. We will also consider server-like modules that may be part of multiple threads, and that may be internally multithreaded, as in Intel's high-end SGX [16] architecture. Furthermore, we will analyze the exact security and (real-time) availability guarantees of our multithreading model. To do so, we will look into recent (theoretical) developments on multi-module *fully abstract* compilation [1, 21].

4.2.2 Protected Scheduler

Having developed a suitable concurrency model, we multiplex the CPU time resource via a preemptive real-time scheduler. Related work [13] employs a pre-configured hardware scheduler to enable the on-schedule execution of critical tasks in a static, single-purpose embedded system. Given that we target highly dynamic deployment scenarios, we implement the scheduling policy completely in software, allowing for maximal flexibility without introducing an omnipotent TCB. More specifically, by building upon the secure interrupt and isolation hardware primitives, we (*i*) preserve the integrity of the scheduler's internal state, (*ii*) guarantee its regular invocation, and (*iii*) avoid a trusted software layer [2, 13] that saves/restores task state on context switch. When interrupting an executing thread, our scheduler solely keeps track of the address of the currently executing SPM. Continuing a thread at a later time then simply comes down to "returning" into that SPM. As explained above, individual software modules remain responsible to guard the entry of their protection domain, and restore internal execution context on successful entry. Any invalid entry attempts should be reported to the scheduler, who may subsequently decide to prevent further execution of the offending task.

To ensure the practical relevance of our results, we base our prototype on an existing real-world embedded OS. We selected FreeRTOS[2] as our case-study real-time kernel, motivated by its (*i*) widespread use, (*ii*) relatively small code base, and (*iii*) multithreading task model. We expect the task of encapsulating the FreeRTOS scheduler in an SPM to be feasible, and already made progress in doing so. Our current prototype supports the interleaved execution of multiple FreeRTOS tasks via explicit `yield` calls. From our experiences, we can tell that FreeRTOS contains lots of optional features and was not designed with security or guaranteed availability in mind. Applications are allowed to stop the scheduler, or disable interrupts for an unlimited amount of time. Moreover, the kernel contains lots of critical sections to prevent interrupts from bringing the system in an inconsistent state.

One of the challenges will be to prevent an attacker from breaking the scheduling policy by abusing public API functions or registering custom interrupt handlers. We aim to prevent this by (*i*) re-

[2] `http://www.freertos.org`

moving optional features from the FreeRTOS API, (*ii*) employing our atomicity primitive, and (*iii*) "locking" the scheduler on entry, so that scheduler functions have run-to-completion semantics, while interrupts are still being served.

4.2.3 Asynchronous Communication

Being able to *synchronously* call unprotected code or an SPM within a single thread does not always suffice in practical real-time application scenarios. Consider for example a small dedicated task that wants to communicate (confidential) sensor readings *asynchronously*, without risking to miss a sensor value while waiting for the receiver to return. To support such scenarios, we will build an IPC mechanism that allows tasks to communicate indirectly, via an untrusted intermediate software module. Our mechanism will allow a secure thread to communicate with another untrusted thread, without loosing its availability guarantees and while preserving the confidentiality and integrity of the passed data.

Consistently, we will base our prototype on FreeRTOS's IPC component. FreeRTOS realizes inter-task communication via fixed-length queues, which are also used to implement semaphores and mutexes. FreeRTOS allows receivers to *block* on a queue until data becomes available by removing the corresponding task control block from the scheduler's ready list. Following the principle of least privilege, our queue implementation will be encapsulated in its own protection domain, and communicate via a restricted privileged interface with the scheduler. We will make a clear distinction between queue-private and scheduler-private memory, and only allow the IPC module to request the currently executing task to be blocked (with a timeout). These measures will ensure that a FreeRTOS application not using queues, cannot be affected by a vulnerability in the queue implementation.

We are confident on the feasibility of such an IPC module. In previous work [26] we presented a similar intermediate SPM that realizes a form of protected shared memory. This shared memory service can be regarded as a primitive form of message passing IPC, where data is copied in a private memory buffer, and subsequent access is restricted according to some access control policy.

5. Related Work

Protecting the internal state of a running software entity against other potentially malicious entities is a well-known requirement, and an active research field. There has been much work on isolating software components in space and time, both in conventional as well as real-time environments. In the following we compare our approach to these proposals – ranging from lightweight embedded techniques to higher-end solutions.[3]

Embedded security architectures. Considerable research effort has been put in providing software isolation for embedded microcontrollers that lack related hardware support. Safe TinyOS [3] enforces type and memory safety through compile time modifications. Harbor [11] partitions a single-address-space into protection domains by inserting runtime checks into compiled binaries. t-Kernel [8] equally modifies untrusted application code at load time to provide virtual memory and OS protection. While these approaches remain compatible with existing microcontrollers, they inevitably decrease performance and rely on a trusted software layer.

In addition to isolating software, various researchers have addressed the issue of availability on a partially compromised embedded platform. t-Kernel [8] includes a rather crude mechanism that rewrites untrusted application code to transfer control to the OS every few instructions. Masti et al. [13] propose a combination of hardware and software components to prevent misbehaving applications or peripheral devices from holding on to the CPU or peripheral bus. In contrast to our approach however, they target static single-purpose embedded systems, and employ a "trusted domain" software layer that is responsible for initializing the system and saving/restoring task state on context switch. The work that comes closest to ours is the TyTAN [2] security architecture that enables an untrusted OS to schedule dynamically loadable isolated tasks in between normal tasks. As indicated above, our multitasking model differs significantly in that TyTAN does not allow modules to call each other directly. Instead, TyTAN relies on an omnipotent kernel-like software layer responsible for (*i*) dynamic loading, (*ii*) saving/restoring state on context switch, (*iii*) inter-module authenticated communication, and (*iv*) remote attestation. Furthermore, to the best of our knowledge, TyTAN does not protect against denial-of-service attacks where an adversary for example disables interrupts for an unlimited period of time.

While we focus explicitly on the low-end side of the embedded spectrum, more resource-intensive solutions exist to host mixed-criticality software on the same processor. The ARINC 653 avionics standard defines a partitioned embedded architecture that provides strict time and space partitioning for safety-critical real-time systems. Spatial as well as temporal isolation can be achieved via a trusted hypervisor [4] that assigns a fixed periodic time slice for a number of partitions, each containing their own OS and applications. The local OS within a partition executes its applications via a preemptive priority-based scheduling policy, whereas the hypervisor is responsible to (*i*) predictably schedule partitions, (*ii*) virtualize interrupts, and (*iii*) direct (limited) inter-partition communication. ARINC 653 thus allows mutually distrusting applications to be isolated in terms of memory and processor resources. In this, the approach shares our goals, but there are two major differences. First, we do not rely on a trusted hypervisor software layer that enlarges the TCB, and imposes a significant runtime overhead. Second, we do not regard SPMs as "partitions" with a dedicated internal OS, and expensive inter-protection domain communication. Instead, our approach relies solely on hardware for the spatial isolation of fine-grained protection domains, and enables a (single) unprivileged real-time scheduler to implement the desired temporal isolation.

High-end security architectures. Ongoing research seeks to realize PMAs for conventional high-end computer systems. Fine-grained protection domains enforced by a small TCB could indeed improve the security guarantees offered by a large omnipotent OS kernel that sandboxes each application in its own virtual address space. Such PMAs have successfully been implemented as an additional layer of protection enforced by a small hypervisor [14, 15, 24] or incorporated in a commodity OS kernel [22]. Moreover, recent Intel x86 processors are equipped with Software Guard eXtensions (SGX) [16] that allow the isolated execution of security-critical code via hardware-enforced *enclaves* in the virtual address space of a process, managed by an untrusted OS. SGX enclaves may be multithreaded, where internal threads are bound to a larger unprotected execution thread.

Our concurrency model where execution threads may run through multiple SPMs during their lifetimes resembles the *migrating thread* [19] approach, which facilitates predictable inter-protection domain communication in component-based OSs. Our approach of implementing OS services in unprivileged modules also relates to microkernels [9, 12]. Importantly however, we do not rely on a privileged kernel, as microkernel abstractions – including protection and authentication – are enforced at hardware level. In this regard, we previously [26] suggested that a Sancus-like PMA can be regarded as a zero-software microkernel. On the other hand, the complete seL4 microkernel [9] has been formally verified, in-

[3] Note that the division between "embedded" and "high-end" architectures is somewhat arbitrary, due to the wide range of hardware/software solutions that are considered as embedded systems.

cluding worst-case execution times, which makes it a suitable and trustworthy alternative for virtual memory architectures. Notably, our approach works in a single-address-space, and allows a remote stakeholder to explicitly verify the OS-like server modules.

6. Conclusion

Minimization of the Trusted Computing Base (TCB) has been one of the key principles of computer security since the start of the field. There has been considerable progress over the last years in building systems that can securely execute software modules with a small hardware-only TCB. But focus has mainly been on confidentiality and integrity properties rather than on availability properties. As part of our ongoing work in enforcing availability and real-time guarantees with a small TCB, we have presented (*i*) a prototypic hardware mechanism that enables interruptible isolated execution and a deterministic worst-case interrupt latency, and (*ii*) an unprivileged cooperative scheduler that interleaves the execution of cross-protection domain threads.

Our long-term goal is to assure the on-schedule execution of critical tasks on a partially compromised platform. For future work, we plan to further develop our hardware interrupt request logic to enforce the guaranteed and unmodified return control flow from an untrusted interrupt service routine. We will furthermore employ our interrupt mechanism to enable real-time preemptive scheduling, while preserving secure compilation guarantees.

Acknowledgments

The authors thank the anonymous reviewers. This research is partially funded by project grants from the Research Fund KU Leuven, and from the Research Foundation Flanders (FWO).

References

[1] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 171–185. IEEE, 2012.

[2] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: Tiny trust anchor for tiny devices. In *Design Automation Conference (DAC 2015)*, pages 1–6. IEEE, 2015.

[3] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 205–218. ACM, 2007.

[4] A. Crespo, I. Ripoll, and M. Masmano. Partitioned embedded architecture based on hypervisor: The XtratuM approach. In *Dependable Computing Conference (EDCC 2010)*, pages 67–72. IEEE, 2010.

[5] R. De Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede. Secure interrupts on low-end microcontrollers. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 147–152. IEEE, 2014.

[6] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *NDSS*, volume 12, pages 1–15. Internet Society, 2012.

[7] Ú. Erlingsson, Y. Younan, and F. Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*, pages 633–658. Springer, 2010.

[8] L. Gu and J. A. Stankovic. t-kernel: Providing reliable OS support to wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 1–14. ACM, 2006.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[10] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 10:1–10:14. ACM, 2014.

[11] R. Kumar, E. Kohler, and M. Srivastava. Harbor: Software-based memory protection for sensor nodes. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 340–349. ACM, 2007.

[12] J. Liedtke. On μ-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 237–250. ACM, 1995.

[13] R. J. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun. Enabling trusted scheduling in embedded systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 61–70. ACM, 2012.

[14] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 143–158. IEEE, 2010.

[15] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 315–328. ACM, 2008.

[16] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10:1–10:1. ACM, 2013.

[17] J. T. Mühlberg, J. Noorman, and F. Piessens. Lightweight and flexible trust assessment modules for the Internet of Things. In *European Symposium on Research in Computer Security (ESORICS 2015)*, pages 503–520. Springer, 2015.

[18] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX SEC'13*, pages 479–494. USENIX Association, 2013.

[19] G. Parmer. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2010)*, page 91, 2010.

[20] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems*, 37(2):6:1–6:50, April 2015.

[21] M. Patrignani, D. Devriese, and F. Piessens. Multi-module fully abstract compilation (extended abstract). In *Workshop on Foundations of Computer Security*, July 2015.

[22] R. Strackx, P. Agten, N. Avonds, and F. Piessens. Salus: Kernel support for secure process compartments. *EAI Endorsed Transactions on Security and Safety*, 15(3), 2015.

[23] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.

[24] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 2–13. ACM, 2012.

[25] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*, pages 344–361. Springer, 2010.

[26] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. Secure resource sharing for embedded protected module architectures. In *International Conference on Information Security Theory and Practice (WISTP 2015)*, pages 71–87. Springer, 2015.