Responsiveness Guarantee for the Sancus Protected Module Architecture

Michiel Van Beirendonck
KU Leuven, 3000 Leuven, Belgium
michiel.vanbeirendonck@student.kuleuven.be

ABSTRACT

Embedded systems are becoming more and more widespread, interconnected and software-extensible, thereby exposing themselves to security threats. Protected Module Architectures (PMAs) are a recent solution to these threats and have been used successfully in the low-end spectrum of embedded devices. However, while these architectures offer strong security guarantees for protected software modules, they do not guarantee their availability on a partially compromised system. Availability of safety-critical systems can refer to a broad range of subjects and is usually related to concepts such as memory availability, hardware redundancy or timely execution. This paper makes a first step towards availability for Sancus, an existing PMA, through a responsiveness guarantee for its protected modules. Protected software applications remain available to stimuli and respond in a timely manner, even on a partially compromised platform. Through a concrete application case the proposed design extensions are evaluated, both in the responsiveness properties they provide as in their performance.

KEYWORDS

Protected Module Architecture, Availability, Secure I/O, Sancus

1 INTRODUCTION

Small embedded devices become more and more widespread in modern day society. Through increased network connectivity and extensibility of the supported software these devices open themselves to security threats. Moreover, they commonly lack the hardware support to address these issues in ways that are well-developed for high-end devices. Recent research proposes to address these concerns through so-called Protected Module Architectures (PMAs) [1, 4, 6, 7]. Through software or hardware extensions, these architectures support the creation of protected software modules in the shared address space. By defining a contiguous code and data section, coupled with hardware extensions for program-counter based memory access control, these modules enable their secure and isolated execution even on a malware-infected device.

While existing PMAs can provide strong confidentiality and integrity guarantees for protected modules, they do not guarantee their availability on a partially compromised system. Numerous applications exist where the availability of modules is critical or even life-threatening, ranging from hospital equipment to the automotive sector. However, current PMAs either rely on an omnipotent software Trusted Computing Base (TCB) [1] — which in turn needs to be attested to a remote stakeholder — or put the availability of their protected modules explicitly out of scope. An example of the latter is Sancus [6], a PMA that provides a stakeholder with strong security guarantees with a minimal hardware-only TCB.

Availability is a commonly used term in system security and includes concepts such as memory availability, hardware redundancy, timely execution and more. In this paper, I make the contribution of extending Sancus with a responsiveness guarantee for protected software applications, as a first step towards their general availability. More specifically, I make the following contributions:

- I extend Sancus with a responsiveness guarantee for protected modules. Through hardware and software extensions, protected software applications remain available to external stimuli, even on a partially compromised platform.
- I evaluate the proposed extensions in efficiency and effectiveness. Through a real-life application case I show that the extensions are relevant, that the system is responsive and I compute the total application overhead incorporating Sancus' support for secure I/O. Experimental results show that the software overhead of the extended design is acceptable in those cases where the application benefits from the provided security and responsiveness guarantees.

The remainder of this paper is structured as follows. Section 2 discusses PMAs in more detail and specifically elaborates on Sancus to provide the reader with the background for this paper. In Section 3, I state this paper's objectives, identify possible attack vectors and in Section 4 propose the extensions needed to Sancus' design. Section 5 evaluates these extensions, both in terms of achieved properties and performance, based on a real-life application case. Finally, in Section 6, I talk about future and related work and Section 7 concludes this paper.

2 PROTECTED MODULE ARCHITECTURES: SANCUS

Sancus is a security architecture for low-end, software-extensible networked embedded devices [6]. Through inexpensive hardware extensions the Sancus architecture extends the low-end TI MSP430 microcontroller and provides strong isolation guarantees, remote attestation, secure communication and secure linking while keeping a small hardware-only Trusted Computing Base (TCB). The Sancus system model is that of a microprocessor-based system owned and administered by an *Infrastructure Provider (IP).* Individual, third-party *Software Providers (SP_j)* can then deploy protected *Software Modules ($SM_{j,k}$)* on the device. These software modules are simple binary files consisting of a text section, containing code and constants and a data section containing runtime (meta)data, such as the module's private call stack.

Memory isolation is built on a scheme that a module's data section is only accessible when the program counter is in the appropriate text section of the module. This concept is commonly referred to as Program Counter-Based Access Control (PCBAC) [8] and is enforced through memory access control logic in the processor.

Furthermore, to prevent against well-known return-oriented programming attacks, where an attacker selectively combines gadgets found in a module's text section, the same logic enforces that modules can only be executed by jumping to a single well-defined entry point. By means of a jump table, multiple logical entry points are easily dispatched through this single physical entry point. Sancus' memory access control rules as taken from Sancus 2.0 by Noorman et al. are listed in Table 1.

A software provider can verify with high assurance that one of its modules is loaded unmodified on the device, through a process called remote attestation. To this end, the Sancus design is extended with three cryptographic hardware primitives. By combining a hash function with a key derivation primitive, the Sancus hardware calculates a symmetric cryptographic key for every newly enabled module. This key is based on a hash of the module's text section and can thus be reproduced by its software provider through the same key derivation scheme. Remote attestation is then enforced by only allowing a currently executing and unmodified SM access to its cryptographic key. Confidentiality and integrity of the communication channel between a provider and its modules is further supplied through an authenticated encryption hardware primitive. Furthermore, a software provider can use his cryptographic key for the *confidential loading* of protected modules. He does so by deploying a decrypted module and then calling the `protect` instruction with a Message Authentication Code (MAC) as extra argument. When invoked this way, the protect instruction will decrypt the module in place, before enabling its protection. Sancus also provides the *secure linking* of software modules. If a module wants to securely link another module, it can invoke a dedicated `attest` instruction to check the existence and integrity of a module at a specified address. Using this instruction, a module can securely check for the unmodified presence of another expected module, and only then jump to that module's entry point.

The discussed Sancus extensions can make it extremely tedious to write modules for a Sancus-enabled device. Indeed, software developers need to account for numerous specificalities, such as ensuring a module's private call stack resides in its data section. Therefore, the Sancus distribution[1] comes with a dedicated C compiler, enforcing these restrictions automatically.

From/to	Entry	Text	Data	Unprotected
Entry	r-x	r-x	rw-	rwx
Text	r-x	r-x	rw-	rwx
Unprotected/Other SM	--x	---	---	rwx

Table 1: Memory access control rules enforced by Sancus using traditional Unix notation. Taken from Sancus 2.0 by Noorman et al. [6].

2.1 Attacker model

The Sancus architecture was designed with a strong attacker model in mind. Attackers have two main capabilities. Firstly, attackers can manipulate all the software on the device. They can act as a software provider and deploy malicious protected modules. Attackers can tamper with the operating system or even install a completely new one. Secondly, attackers can control all network communication. They can sniff the network and modify traffic, but cannot break cryptographic primitives. It is important to note that a Sancus protected module is still vulnerable to security threats in its own implementation, for instance if it contains buffer overflows or entry points for API-level attacks. This is especially relevant, since the scheme to handle memory violations as discussed in Section 4.3 relies on this bug-freeness of protected modules to retain their confidentiality. Furthermore, for the work discussed in this paper, attackers are assumed to only have access to the device after the initial loading phase, as is often the case in a multi-stakeholder context. Lastly, attacks against the device's hardware are considered out of scope.

2.2 Security properties

Corresponding to the attacker model described above, Noorman et al. list the following security properties for Sancus [6] that are relevant to this paper's design:

Software module isolation A software module runs isolated in the sense that no other module or unprotected code can read or modify its text or data sections. Modules can only interact by jumping to a dedicated, well-defined entry point.

Remote attestation A software provider can verify with high assurance that a specific protected module is loaded unmodified on the device.

Secure communication A software provider can communicate with protected modules on a device with high confidentiality, integrity and authenticity guarantees.

Secure linking A software module can specify to securely link another module. Before jumping to the other module's entry point, it can verify the presence and integrity of this module and thus have high assurance that it is calling the intended module. Furthermore, runtime interactions between module A and module B also have secure communication guarantees.

Confidential deployment A software provider can choose to deploy modules with the guarantee that no attacker will be able to access the module's code at any point in time.

2.3 Secure I/O

The Sancus architecture is well suited to handle the secure execution of distributed event-driven applications on shared infrastructures. Recent research by Noorman et al. [5] examined how to use Sancus to provide these applications with an *authentic execution* guarantee. In this work, the notion of authentic execution is described in the following way: "if the application produces a physical output event (e.g.,turns on an LED), then there must have happened a sequence of physical input events such that that sequence, when processed by the application (as specified in the high-level source code), produces that output event" (p. 1).

An important part of authentic execution, is secure peripheral access. The MSP430 architecture uses Memory-Mapped I/O (MMIO) to communicate with peripherals. MMIO uses the same address space for memory and I/O and peripheral devices can be accessed through the same CPU instructions used for regular memory. Therefore, providing exclusive access to peripherals on Sancus is supported

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

natively, by wrapping the device's memory region in the data section of a dedicated SM, a so-called *driver SM*. Because Sancus' SMs are contiguous, a SM can either be used for MMIO or data, but not both. This means driver SMs are severely limited in functionality because they cannot use any memory, including a private call stack. Therefore driver SMs are implemented as very primitive SMs, only providing entry points to read and/or write to its respective MMIO regions. A driver module can then further provide exclusive access to a device by only allowing a single application module access to its entry points. If so desired, the application module can implement a more complex API to provide other SMs access to the device. At load-time, the application module will verify if it was assigned exclusive access and abort otherwise. The application SM can also attest the driver SM to verify its integrity and ensure it is mapped over the relevant MMIO registers. If attestation fails, the application SM aborts as well. This procedure ensures that an application SM either has exclusive access to a device or is aborted. The issue of handling hardware interrupts in SMs is also supported out of the box by registering a SM's entry point in the Interrupt Vector Table (IVT). The respective interrupt then invokes the SM as an interrupt handler. Protected interrupt handlers verify their caller by use of unforgeable Interrupt Request (IRQ) identifiers to make sure attackers cannot spoof physical input events. Together, these concepts provide the property of *authentic execution*.

While authentic execution is a strong property on its own, it doesn't give any guarantees about when or even if in– or outputs will arrive. An attacker that indefinitely disables interrupts or reboots the system through a memory violation are only some of the attack vectors identified in the following Section. The Sancus design extensions discussed in this paper can provide such a responsiveness guarantee. Specifically, the real-life application case discussed in Section 5.1 must show that the extended Sancus design can react to hardware interrupts and respond in a timely manner, even if the platform is partially compromised.

3 OBJECTIVES

The Sancus architecture described above offers strong confidentiality and integrity guarantees for protected software modules. However, through some design choices, it places their responsiveness on a compromised system explicitly out of scope. Furthermore, the type of embedded systems that Sancus targets are increasingly used in applications where the responsiveness of modules is safety-critical. I therefore argue that responsiveness extensions for Sancus are relevant, even when making certain design simplifications to achieve them. The remainder of this section first discusses these simplifications to the Sancus model. Afterwards, the intended responsiveness properties are stated explicitly and the possible attack vectors are identified.

3.1 Design simplifications

The Sancus design divides the address space into an unprotected domain with arbitrary code and multiple contiguous domains containing protected modules. While these protected modules are well-isolated and mutually distrusting when considering confidentiality and integrity, I make the assumption of a *single trusted domain* when considering responsiveness. Protected modules are expected not to harm the system's responsiveness in any way: they do not monopolize shared system resources, put the platform in an unresponsive state or behave unexpectedly. In Section 4.3, I will reason that this further requires that protected modules be implemented bug-free, so as to not cause memory violations. Section 6.1 will discuss the extra challenges that come with the mutually distrusting case, as well as how this is addressed in related work. It is important to see that while this eases restrictions for a module's responsiveness, all the imposed Sancus properties of Section 2.2 remain valid for the security of protected modules. Protected modules are still mutually distrusting about the confidentiality and integrity of their code and data.

3.2 Explicit properties

The main goal of this paper is to provide a software provider with high responsiveness guarantees for software applications in the protected domain, while simultaneously guaranteeing all security properties as described in Section 2. Such a software application may consist of one or more protected modules, but as I will discuss in Section 4.4, may not contain unprotected code. Generally, the described software application consists of a single protected Interrupt Service Routine (ISR) invoked by a specific hardware event, which then calls more protected modules as needed to handle the event accordingly. In the remainder of this text, I will refer to such a protected software application as a *protected software thread*. Note that a protected software thread may traverse several protected modules, possibly deployed by different software providers. With *responsiveness* for these software threads, I signal a property specifying their timely execution. Timely execution is guaranteed in that there exists an unspecified interrupt latency, that assures hardware events are handled within a non-infinite deadline. More specifically, I build the property of *protected software thread responsiveness* on top of Sancus' remote attestation feature and formulate it explicitly as follows:

Protected software thread responsiveness A software provider can verify with high assurance that protected applications on a device — protected software threads — respond in a timely manner to external stimuli if he is able to *remotely attest* all protected modules on the device. Protected software threads are guaranteed to have a finite interrupt latency that is not otherwise specified.

If remote attestation succeeds, a software provider not only has high guarantees that modules are running unmodified, but also that protected applications remain responsive to external events. Note that a software provider may have to attest modules deployed by other providers. This is true because a module might have been maliciously modified before its protection was enabled, which means that module can no longer be trusted to preserve the responsiveness of other modules. Attesting all modules on a device is expensive. Luckily, such an action only needs to be performed once. Indeed, once a software provider has guarantees about the integrity of a module, only unloading the module could lift these guarantees. Because modules are trusted for responsiveness, a module may only be unloaded with full support of all engaging stakeholders.

3.3 Attack vectors

An attacker has numerous ways to harm the responsiveness of the system in the existing Sancus design. For each of the following attack vectors, Section 4 proposes design extension to handle them in a way preserving the responsiveness of protected software threads.

- **Halting protected software threads.** The current Sancus design does not yet enforce the uninterruptability of protected modules in hardware, which means an attacker could leverage the hardware to continuously interrupt a protected software thread. Because the current design excludes a trusted scheduling entity, this would effectively halt the thread's execution. Moreover, by pushing a self-chosen return address on the stack, the attacker could prevent the control flow from returning to the interrupted thread at all.
- **Monopolizing resources.** In the current Sancus design, an attacker is capable of monopolizing shared system resources such as the CPU.
- **Deploying modules.** As discussed in Section 3.1, protected modules are expected to trust each other concerning their responsiveness. Because the attacker has full software control and can freely launch its own protected modules, he could easily break this guarantee.
- **Overwriting crucial data structures.** By overwriting data structures crucial to the execution of protected software threads, an attacker can cause the thread to behave unexpectedly or even not execute at all. Moreover, by overwriting microcontroller state bits, an attacker is currently able to directly alter the system's responsiveness by putting it in low-power modes.
- **Redirecting unprotected outcalls.** By sniffing the network for unprotected outcalls, an attacker is able to identify where protected software threads are vulnerable. An attacker capable of full software control could overwrite this unprotected code with his own and cause the thread to behave unexpectedly.

4 PROPOSED SANCUS EXTENSIONS

The above shows that an adversary currently has numerous ways to harm the responsiveness of Sancus protected threads on a device. This section proposes design extensions so that protected software threads can be guarded against these attacks.

4.1 Uninterruptible protected domain

The existing Sancus design relies on the processor disabling interrupts while a protected module is executing to prevent its register contents being leaked outside the module should an IRQ be serviced. However, the current Sancus hardware does not enforce this yet. To prevent an attacker from continuously halting a protected thread's execution, he must be kept from leveraging the hardware to interrupt protected modules. Because there is mutual trust between protected modules for responsiveness, interrupts could be allowed if the program flow proceeds by jumping to a protected ISR. This ISR is then trusted to timely hand back control to the interrupted module. Although recent research has identified that such secure interrupts for Sancus are indeed possible, these works are either still work-in-progress [9] or do not address some of the new security threats that interruptible modules bring [2]. Therefore, in the current work I chose not to support interruptible modules and only briefly discuss them in Section 6.1 along with some of the new challenges they bring.

Uninterruptability of protected modules can be enforced in both software and hardware. The software approach involves disabling and again re-enabling interrupts on SM entry and exit, while the hardware approach consists of extending the existing IRQ logic to ignore interrupts while a SM is executing. Because the software approach incurs a performance overhead on every SM call and the hardware approach only needs minimal extra logic, I chose to implement the latter.

Along with an uninterruptible protected domain, the unprotected world must remain open to external stimuli at all times to prevent an attacker from holding on to the CPU and to provide a minimal interrupt latency for protected software threads. Two important observations can be made from this statement. Firstly, it should not be possible for an attacker to anyhow disable interrupts on the microcontroller. Secondly, because an attacker can freely launch its own ISRs and ISRs are generally uninterruptible, either nested interrupts must be enabled on the microcontroller or unprotected ISRs need to be forced to return after a well-defined number of CPU cycles. To keep the interrupt latency for protected threads as low as possible I opted to enable nested interrupts through further modifications of the existing IRQ logic.

Uninterruptible protected modules imply run-to-completion semantics for protected software threads. While the proposed design supports the presence of several such collaborative threads, this greatly increases the thread interrupt latency. Indeed, in the case of multiple threads, the worst-case execution time of protected software threads with a higher priority for interrupts is appended to the worst-case interrupt latency of threads with a lower interrupt priority. Note also that, if a software provider wants to keep the interrupt latency low, he must not provide any public logical entry points to its modules. This is true because an attacker could continuously call this entry point to append the module's execution time to the interrupt latency. In the current design, an attacker cannot be prevented from jumping to a module's physical entry point. However, if he cannot provide a valid logical entry index, this is resolved after a limited number of CPU cycles.

4.2 Extended protected domain

Certain memory regions of the the microprocessor need to be protected from a malicious attacker to provide responsiveness of protected software threads. Several such sections are identified here. Firstly, from the discussion above, it is clear that by distinguishing an uninterruptible protected domain and an interruptible public domain, an attacker must be prevented from deploying modules into the protected domain. Secondly, other important regions either handle the microcontroller's state or are crucial to the behavior of software threads such as the status register, peripheral configuration registers or the in-memory IVT.

4.2.1 Exclusive trusted domain. Protected modules in the trusted domain have a mutual trust relationship concerning their responsiveness. Because of this property, an attacker must be prevented

from deploying modules into the protected domain that could possibly harm the responsiveness of other modules. The Infrastructure Provider *IP* has a priori no reason to distrust any specific Software Provider SP_j, thus cannot distinguish between a trusted *SP* and an attacker *SP* beforehand. Therefore, the design considers SP_1 — the *SP* who deploys the first enabled module $SM_{1,1}$ — a trusted *SP* and relies on SP_1 to decide which other SP_j's to trust. The consequence is that, if SP_1 turns out to be malicious, *IP* will have to revoke the system.

The described strategy can be enforced by relying on Sancus' feature for confidential deployment, where the `protect` instruction must be invoked with a MAC. This MAC could be used to authenticate the *SP* that is trying to enable a module's protection. By exclusively allowing the decrypting form of the `protect` instruction after the initial loading phase, only software providers with a valid provider key could subsequently deploy modules on the device. In the current design, this would mean that only those SP_j that have access to the device during initial loading can deploy other modules. In future work, this scheme could further be expanded to where these SP_j can give *IP* permission to hand out other keys to trusted SP_k.

4.2.2 Status register. Section 4.1 identified that it must not be possible for an attacker to disable interrupts on the microcontroller. Enabling and disabling of interrupts is controlled trough one of the microcontroller's special registers called the Status Register (SR). In fact, an attacker should not be allowed to write to several bits in SR, since SR contains sensitive register bits that for example disable the CPU. To this end, I implemented hardware protection so that SR is only configurable from inside protected modules, which are trusted to preserve responsiveness. The downside to this approach is that hardware-level writes to SR are also affected. Following an unprotected ISR, the microcontroller will therefore be unable to restore itself to its original state. This is true because on ISR entry, the IRQ logic will push the current value of SR onto the stack and then clear any low-power mode bits. Upon ISR exit, the initial SR configuration is retrieved from the stack, but as the ISR is unprotected code, the subsequent write to SR is simply ignored. If allowed however, this is a mechanism that could easily be exploited by an attacker, because he can freely access and alter the SR configuration while it resides on the unprotected stack. Future work could investigate storing and retrieving the SR configuration at a write-protected location in memory, such as a Sancus-enabled SM.

4.2.3 Peripheral configuration. Section 2.3 discussed Sancus' support for secure I/O applications on shared infrastructures. By way of protected driver modules, a stakeholder is guaranteed exclusive access to a device. If a software provider wants his application to behave as expected, he must be careful to protect all I/O peripherals, as well as their configuration registers by mapping them in a dedicated SM.

4.2.4 Interrupt Vector Table (IVT). Sancus supports configuring a protected module to respond to an IRQ by registering its entry point in the in-memory IVT. When an interrupt is accepted, program execution begins at the address stored in the corresponding IVT entry. To ensure correct ISR execution, the entries in the IVT should be write-protected from a malicious attacker. This is again supported natively by wrapping the IVT in a dedicated SM. By wrapping the IVT in the *text* section of the SM, it is both write-protected and remote attestation can guarantee that the IVT is configured as expected. While achieving the desired property, this approach has the clear downside that it is unflexible and does not allow ISR (un)registering at runtime.

4.3 Memory access control

The current Sancus design relies on a combinational hardware circuit to enforce Memory Access Logic (MAL) and signal for violations. Any such violation is handled by the simple design choice of resetting the processor and clearing memory. While clearly being secure, such a policy has a very negative impact on the availability of a device. For debugging purposes, the Sancus design also comes with a synthesis option to generate a non-maskable interrupt instead of a full fledged reset. While this may superficially seem like a solution to the availability challenge, this again introduces the problem of supporting interruptible SMs, which I previously argued is not in the scope of this work. However, this problem can be circumvented by requiring protected modules to be *properly implemented*, which, as discussed in Section 2.1, must already be the case for their confidentiality. Indeed, if all protected modules are bug-free, memory violations will only occur in the untrusted domain, where code is fully interruptible and re-entrant.

With this extra requirement on protected modules, memory violations could be handled in software by invoking an ISR that deals with violations accordingly. On a memory violation the memory backbone will disable all memory accesses and the frontend will initiate the IRQ sequence. The Sancus MAL distinguishes between four types of memory violation which can all be handled by altering them to allowed actions: *(i)* Invalid write instructions or illegal module deployment calls are ignored. *(ii)* Invalid read instructions return some default value. *(iii)* Illegal jump instructions get redirected to an exception handler. Out of a performance perspective, only faulty jumps are handled in software by invoking an ISR and all other violations are resolved in hardware. Note that this ISR cannot be part of a protected module. Indeed, because protected ISRs are uninterruptible, an attacker could leverage this to hog the system by continuously jumping to a non-entry point in a module's text section.

4.4 Unprotected outcalls

With the attacker model of Section 2.1, it is easy to show that protected software threads must not make any outcalls to unprotected code. An attacker able to sniff the network for these outcalls, could simply overwrite unprotected subroutines with his own code and thereby cause the thread to behave unexpectedly. Therefore, I extended the Sancus linker with support to warn end-users of these unprotected outcalls. It remains up to the module developer how to handle these unprotected calls, e.g. by replacing them with protected code or removing them altogether.

5 EVALUATION

This section evaluates the proposed Sancus extensions. Starting from a real-life application case, an informal responsiveness argument is set out and the induced performance overhead is analyzed.

5.1 Application case

Protected software thread responsiveness coupled with Sancus' authentic execution guarantee gives software developers strong capabilities to handle physical events on Sancus-protected devices. Therefore it is possible to construct an application case that conveys these two previously discussed concepts. It is also this application case that is used as the basis for performance evaluations of the design. To ensure reproducibility, source code for this application case as well as the Sancus extensions discussed in this paper are publicly available at https://github.com/Michielvb/sancus-availability. The Sancus implementation is based on an open source implementation of the TI MSP430 architecture: the openMSP430 from the OpenCores project [3]. For the application case, the core was synthesized on a a Xilinx XC6SLX25 Spartan-6 FPGA using Xilinx ISE Design Suite and extended with a StickIt! board from XESS Corp. A StickIt!-LedDigits module also from XESS Corp. is used as output peripheral to produce palpable output events.

In short, the application case consists of a protected software thread cycling the LEDs on the LedDigits module, by use of timer interrupts on the microcontroller. When such an interrupt arrives, the IRQ logic retrieves the corresponding protected thread from the IVT, which then runs to completion and cycles the LEDs before passing back control. *Authentic execution* guarantees that only timer interrupts cause the LEDs to cycle, while *protected software thread responsiveness* guarantees that following a timer interrupt, LEDs are cycled within an unspecified but finite deadline.

5.2 Responsiveness argument

For the application case described above, I reason informally why my design achieves the property of *protected software thread responsiveness*. During the initial loading phase, infrastructural software is loaded onto the device and this also includes the module protecting the in-memory IVT. Now examine a time frame after the initial loading phase, with all protected modules of the LED software thread in place. At this time, the remote stakeholder attests that all protected modules on the devices are running unmodified and verifies that he trusts these modules to preserve system responsiveness. Because a possible attacker does not have access to a cryptographic provider key, he is excluded from deploying modules into the protected domain. Since the protected domain also includes a MMIO module protecting the timer configuration and the remote stakeholder attested all modules, he is certain that an interrupt will arrive at a time specified by the timer's configuration registers. That the platform will be responsive when the interrupt arrives, follows from the following observations: *(i)* The attacker is restricted to writing unprotected code, which can be freely interrupted. *(ii)* The attacker cannot alter the system's responsiveness to interrupts as unprotected code cannot write to the status register. He cannot disable interrupts on the microcontroller, nor place the system in any low-power mode. *(iii)* The attacker model excludes hardware-level attacks. Upon interrupt reception, the IRQ logic retrieves the protected ISR entry point from the in-memory IVT. Because the IVT's entries are write-protected by an infrastructural SM, the remote stakeholder is certain that the correct ISR gets loaded. Subsequently, only invalid jump instructions could possibly interrupt the currently executing module, which implies that the protected software thread runs to completion if only it is *properly implemented*.

5.3 Performance

The overhead of the Sancus-enabled application case is evaluated here. As most responsiveness extensions are either hardware only or get enabled during the initial loading phase, this overhead is almost entirely due to Sancus' authentic execution extensions. Still, because the extensions proposed in this paper greatly enhance Sancus' ability to handle peripheral I/O applications, I argue that an overhead evaluation is in place here.

The overhead is analyzed in the interval starting from the moment a timer interrupt arrives until the moment all LEDs are cycled and the ISR returns control through the `reti` instruction. Common practice in writing interrupt handlers is to keep the ISR itself as short as possible and only notify the main event-loop to call the application before passing back control. However, as the main event-loop is unprotected code, doing so would break the integrity of the run-to-completion semantics for protected software threads. Therefore, both the protected and unprotected variant are written without this common practice in mind, to provide better comparison between the two. Furthermore, the application assumes the desired LED outputs are not available in memory, but need to be read from the corresponding protected memory addresses on every new LED cycling.

With the above assumptions, I found that the unprotected application needs 116 CPU cycles to complete, while the Sancus variant needs 2958 CPU cycles. While these results seem unacceptable at first, the reader must keep in mind that the relative performance overhead — more than 2500% in this case — is greatly affected by the complexity of the application. Indeed, because the discussed aplication is case exceptionally simplistic (cycling 8 LEDs requires reading 8 values and writing 8 values), this requires virtually no processing for the unprotected ISR. On the other hand, the protected ISR needs to verify the integrity of the MMIO driver module a total of 16 times and vice versa, ammounting to a full 32 attestations. Also, virtually no work went into optimizing the application's code and grouping read and write calls could already greatly reduce the required number of integrity checks. Further analyzing the overhead found that protected ISR entry takes 92 cycles, exit takes 43 cycles, whereas each as a LED read and write (with caller/callee verification) take respectively 172 and 175 cycles, which proves above statements. Whether this overhead is acceptable will depend on the application, though I argue that for safety-critical tasks this will often be the case considering the benefits from security and responsiveness guarantees.

6 DISCUSSION

With the recently ongoing surge of low-end embedded devices in the IoT, their security and availability is an important issue and an active field of research. This section discusses both future challenges for the security of Sancus protected systems, as well as how the security and responsiveness problem are addressed in related work.

6.1 Distrusting multi-stakeholder context

As discussed in depth in previous sections, the current design relies on protected modules trusting each other to preserve responsiveness. While this greatly simplifies the problem to solve, it also significantly reduces the relevance of the design in a multi-stakeholder context. Because one of the stakeholders in such a context could very well be a malicious attacker, responsiveness will require protected modules to be fully interruptible and re-entrant. Recent research by de Clercq et al. explored such interruptible modules for Sancus [2]. They propose to handle secure interrupts by use of a *return trampoline function* that makes careful use to clear registers containing sensitive data when returning from a secure ISR. However, this work does not address some of the specific security threats that interruptible modules bring like stack overflow into another module's data section or time-of-check-to-time-of-use-attacks [7]. Furthermore, interruptible modules alone cannot provide responsiveness, let alone real-time guarantees in a multi-stakeholder context. Indeed, when a protected module or even arbitrary code can freely interrupt another module, the need arises for a trusted preemptive scheduler to provide applications with such a responsiveness guarantee. These issues are more thoroughly addressed in recent work by Van Bulck et al., describing their effort towards availability and real-time guarantees for Sancus [9]. While this research includes some interesting concepts like deterministic interrupt latency, trusted scheduling and atomic execution of critical code sections, it is still a work-in-progress implementation-wise and therefore remains mostly out of scope for the work discussed in this paper. Real-time guarantees for Sancus on a partially compromised platform are the long-term goal and I feel that its implementation could benefit from the results produced in this paper.

6.2 Availability

Responsiveness and the on-schedule execution of critical tasks is only one aspect of what is referred to as *availability* in information security. Availability can encompass among others memory availability, hardware redundancy, safe backup and timely execution. While not all of these concepts are applicable for low-end embedded devices, working towards other availability guarantees for Sancus could be an interesting avenue of research and would greatly enhance Sancus' capabilities for applications in the IoT.

6.3 Related work

Two protected module architectures that target the same low-end spectrum of embedded device as Sancus are Trustlite and TyTan. The Trustlite [4] trusted task model, which they refer to as *trustlets*, resembles that of Sancus protected modules. Compared to Sancus, Trustlite features an Execution-Aware Memory Protection Unit (EA-MPU), allowing for more flexible PCBAC policies in a configurable hardware table. This facilitates data sharing between modules and allows a module to define multiple private data sections. However, Trustlite requires a secure loader for loading trustlets into memory and programming the MPU and does further not allow trustlet unregistering at runtime. Trustlite also supports secure interruption of trusted tasks in much the same way as trampoline functions would for Sancus. Their trustlet invocation model however, differs significantly from Sancus in the sense that trustlets cannot directly call each other, but rely on message-passing for IPC. TyTan [1] extends Trustlite with real-time guarantees and includes dynamic loading as well as remote attestation through key derivation. Where as Sancus addresses these issues with a hardware-only TCB, TyTan relies on a trusted software layer for secure interrupts, protected IPC, dynamic loading and remote attestation. TyTan implements several extensions also included in this paper (e.g. protecting the integrity of the interrupt descriptor table through the EA-MPU), but does not prevent some of the Denial of Service (DoS) attacks addressed here such as disabling interrupts.

7 CONCLUSION

Protected Module Architectures are increasingly used in applications where system availability is safety-critical. As a first step towards full availability, these architectures could greatly benefit from a responsiveness guarantee for their protected modules. In this work, I provided Sancus with such a guarantee for protected software applications. Through software and hardware extensions I *(i)* addressed protected module uninterruptability. *(ii)* protected crucial memory regions from an attacker. *(iii)* implemented a scheme to handle memory-violations. These extensions give software developers powerful capabilities of handling event-driven distributed applications on shared infrastructures and were evaluated through a custom application case. Future work could further address the complex problem of fully distrusting parties in a multi-stakeholder context or work towards other availability guarantees for Sancus protected modules.

ACKNOWLEDGMENTS

I want to thank prof. Frank Piessens for offering me an interesting research project and introducing me to the fascinating world of system security. I also want to thank Jo Van Bulck, Jan Tobias Mühlberg and Job Noorman for their continuous assistance along the way. Lastly I want to thank the KU Leuven Honours committee for accepting my application to carry out this very project.

REFERENCES

[1] 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 52nd ACM/EDAC/IEEE*. IEEE, 1–6.

[2] Ruan De Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. 2014. Secure interrupts on low-end microcontrollers. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*. IEEE, 147–152.

[3] O Girard. 2013. Openmsp430 project. *available at opencores. org* (2013).

[4] 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, 10.

[5] Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2017. Authentic Execution of Distributed Event-Driven Applications with a Small TCB. In *STM '17 (LNCS)*. Springer, Heidelberg. Accepted for publication.

[6] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Transactions on Privacy and Security (TOPS)* 20, 3 (September 2017), 7:1–7:33.

[7] Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, and Frank Piessens. 2013. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*. Springer, 241–251.

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

[9] Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2016. Towards availability and real-time guarantees for protected module architectures. In *Companion Proceedings of the 15th International Conference on Modularity*. ACM, 146–151.