Secure Resource Sharing for Embedded Protected Module Architectures

Jo Bulck, Job Noorman, Jan Mühlberg, Frank Piessens

Secure Resource Sharing for Embedded Protected Module Architectures

Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg and Frank Piessens

iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium

Abstract. Low-end embedded devices and the Internet of Things (IoT) are becoming increasingly important for our lives. They are being used in domains such as infrastructure management, and medical and healthcare systems, where business interests and our security and privacy are at stake. Yet, security mechanisms have been appallingly neglected on many IoT platforms. In this paper we present a secure access control mechanism for extremely lightweight embedded microcontrollers. Being based on Sancus, a hardware-only Trusted Computing Base and Protected Module Architecture for the embedded domain, our mechanism allows for multiple software modules on an IoT-node to securely share resources. We implement and evaluate our approach for two application scenarios, a shared memory system and a shared flash drive. Our implementation is based on a Sancus-enabled TI MSP430 microcontroller. We show that our mechanism can give high security guarantees at small runtime overheads and a moderately increased size of the Trusted Computing Base.

Keywords: Protected Module Architecture, Internet of Things, Embedded File System, Access Control, Resource Sharing, Trusted Computing

1 Introduction

Ongoing developments in our ever-changing computing environment have led to a situation where every physical object can have a virtual counterpart on the Internet. These virtual representations of things provide and consume services and can be assigned to collaborate towards achieving a common goal. This Internet of Things (IoT) brings us unpreceded convenience through novel possibilities to acquire and process data from our environment. With numerous applications in domains such as infrastructure management, transportation, and medical and healthcare systems, the increasing growth of the IoT raises questions regarding the safe and secure deployment and use of extremely interconnected devices. Computing nodes in the IoT are often equipped with inexpensive low-performance microcontrollers that provide just enough computing power to periodically perform their intended tasks, e.g., obtain sensor readings and pass them on to other nodes. As a result, well established concepts and mechanisms from desktop and server environments – hierarchical protection domains, virtualisation, virus scanners, firewalls, etc. – are either not available or cannot easily be employed on IoT-nodes [17].

The problem of trustworthiness and trust management of low-power low-performance computing nodes has previously been discussed in the context of sensor networks [8,14]. Most techniques proposed for this domain focus on observing the communication behaviour and on validating the plausibility of sensor readings to assess the trustworthiness of nodes, which is shown to reliably detect the systematic failure nodes. Yet mechanisms to protect software and data on a node are rare as most work in this domain focuses on efficiency and handle security and privacy requirements as second-class citizens at best.

Contributions. In this paper we describe and evaluate an approach to implement and securely enforce application-grained access control policies for IoT-nodes. Our access control mechanism can manage access to various system resources such as a file systems, Memory-Mapped I/O (MMIO) devices or specific devices attached to an external communication bus. While incurring low overheads, our mechanism guarantees at runtime that only authenticated software modules gain access to resources as specified in the policy; the internal state of the access control implementation is protected and cannot be tampered with.

Our approach is based on Sancus [16], a lightweight hardware-only Trusted Computing Base (TCB) and Protected Module Architecture (PMA) [18]. Sancus targets low-cost embedded systems which have no virtual memory. Recent research on Program Counter Based Access Control (PCBAC) [19] shows that, in this context, the value of the program counter can be used unambiguously to identify a specific software module. Whenever the program counter is within the address range associated with the module's code, the module is said to be executing. Memory isolation can then be implemented by configuring access rights to memory locations based on the current value of the program counter. Sancus also provides attestation by means of built-in cryptographic primitives to provide assurance of the integrity and isolation of a given software module to a third party, which we use to authenticate software modules.

We evaluate a prototypic implementation of our access control mechanism in two application scenarios that facilitate secure data sharing between software modules, (1) through a shared memory implementation and (2) through peripheral flash memory and the Coffee [20] file system. Our evaluation shows that module isolation and access control impose relatively low overheads that should be acceptable in deployment scenarios with stringent safety and security requirements. The application scenarios run on a Sancus-enabled TI MSP430 microcontroller, a single-address-space architecture with no memory management unit. The source code of the evaluation scenario is available at `https://distrinet.cs.kuleuven.be/software/sancus/wistp2015/`.

2 Protected Module Architectures and Sancus

As mentioned in the introduction, our work is built upon Sancus [16], a lightweight PMA [18] specifically designed for embedded systems. Sancus guarantees strong isolation of software modules, called Sancus Modules (SMs), through low-cost

hardware extensions. Moreover, Sancus provides the means for local and remote parties to attest the state of, or communicate with, the isolated software modules. This section gives a detailed introduction of the features of Sancus we use in the rest of this paper.

Isolation. Like many PMAs, Sancus uses PCBAC [19] to isolate SMs. Software modules are represented by a *public text section* containing the module's executable code and a *private data section* containing data that should be kept private to the module. The core of the PCBAC model is that the private data section of a module can only be accessed from code in its public text section. In other words, if and only if the program counter points to within a module's code section, memory access to this module's data section is allowed. Note that on systems that use MMIO, an SM can get exclusive access to a device by mapping its private data section around the MMIO region of the device.

To prevent instruction sequences in the code section from being misused by external code to extract private data, entry into a module's code section should be controlled. For this purpose, PMAs allow modules to designate certain addresses within their code section as *entry points*. Code that does not belong to a module's code section is only allowed to jump to one of its entry points. In Sancus, every module has a single entry point at the start of its code section. Tab. 1 gives an overview of the access control rules enforced by Sancus.

Table 1. Memory access control rules enforced by Sancus using the traditional Unix notation. Each entry indicates how code executing in the "from" section may access the "to" section. The "unprotected" section refers to code that does not belong to a SM.

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

SM Identification. Sancus allows SMs to reliably identify each other. To this end, Sancus assigns a unique ID to each SM when its isolation is enabled. The instruction `sancus_get_id` can be used to retrieve the ID of an SM at a specific address. This can be used to, for example, verify the expected SM is isolated at a specific location before calling its entry point.

To enable the implementation of access control policies, Sancus keeps track of the ID of the previously executing SM. This ID can be queried using the `sancus_get_caller_id` instruction. SMs typically use this feature to restrict access to their entry point to some specific SMs.

Besides enabling SMs to identify each other, Sancus also provides cryptographic primitives for modules to *attest* each other's state. That is, to verify that a module's code section has not been tampered with before the isolation was

enabled and that its code and data sections are loaded at the correct addresses. For this, Sancus employs an elaborate key management scheme that is beyond the scope of this paper. Suffices to say that SMs can be deployed with a Message Authentication Code (MAC) of the code section and load addresses of a module it needs to attest and Sancus provides instructions to verify that the actual isolated module corresponds to this MAC.

Sancus Module Compilation. To securely create SMs for Sancus, a number of specifics have to be considered. For example, every SM needs a separate stack in its private data section to ensure the integrity of control data and local variables. Also, whenever exiting an SM, registers need to be cleared to avoid data leakage. The Sancus distribution includes a C compiler to automate the process of creating SMs. The compiler generates the necessary entry and exit stubs to deal with intricacies mentioned above. Moreover, the compiler allows for the definition of multiple entry points that are dispatched from the single physical entry point supported by Sancus. A generic approach to securely compiling high-level code to low-level language with fine-grained memory access control is presented in [1].

3 Motivation & Related Work

In this section, we introduce the need for a secure embedded file system and discuss this in the light of recent related research. In a wider context, our prototype demonstrates the feasibility of encapsulating and controlling access to a shared system resource through a lightweight trusted software layer on top of hardware-enforced mechanisms.

3.1 Embedded File System Security

Existing embedded file systems [6,5] focus mainly on performance aspects: flash specific optimisations, RAM usage and energy consumption, whereas file protection is non-existing or remains very limited. This is in line with the original concept of a single static unprotected embedded application. Indeed, the design notes for Matchbox, a file system for TinyOS, state literally: "We do not need: Security in any form, [...]" [7]. As another example, Contiki features the Coffee file system [20], a dedicated lightweight flash file system without any form of access control. LiteOS [4] provides its own LiteFS UNIX-like file system in which files may represent data, binaries or devices. It also offers a coarse-grained user-oriented protection mechanism that classifies all users in one of three levels, each with its own `rwx` mode bits.

We argue that in an embedded context, featuring a dynamic multi-stakeholder deployment model, it is software modules rather than users that represent the unit of file protection. Indeed, recall from Sec. 2 that an SM represents the unit of memory protection and authentication. Extending these guarantees with SM-grained protection for shared system resources would thus be valuable.

File protection on a per-SM-basis would furthermore be interesting as it differs from conventional UNIX-like user-oriented file protection [2]. UNIX decides file access based on the identity of the owner of the currently executing program. This coarse grained scheme does however not shield a user from malicious programs that run with her permissions [3]. Moreover, fine-grained file protection is hindered by the default `owner/group/others` file attributes. Capability-based process-specific file protection for UNIX has been proposed [3] as a countermeasure and fine-grained access control can be accomplished with access control lists [9].

3.2 Secure Resource Sharing

PMAs reorganise an unprotected single-address-space into a set of hardware-delimited protected SM enclaves. Secluding SMs in their respective protection domains allows strong security guarantees on the one hand, but also limits the overall flexibility of the system. Indeed, Sancus [16] does not natively support complex policies, such as dynamically allocating and sharing of protected memory, or fine-grained peripheral access control. In this respect, our protected file system serves as a case study on how to encapsulate a typical shared system resource (i.e. secondary storage) in its own protection domain with flexible SM-grained access control policies.

Self Protecting OS Modules. An SM should either fulfil its own needs or rely on the services of an untrusted OS to interact with the outside world. This implies poor trade-offs between flexibility and protection. Consider for example an SM that wants to save confidential data in a file system or read secret values from a sensor. Without additional support this SM would have to either claim the file system / sensor for itself, effectively denying others access to the resource, or accept to use it in an unprotected way.

The key idea we explore in our secure file system prototype is to mitigate this flexibility vs. protection trade-off by adding a level of indirection. In our setup, we build upon the existing Sancus primitives to provide a dedicated module SM_{server} with exclusive access to a system resource and we implement a thin software layer on top to enforce flexible access control policies. Sancus' hardware logic ensures SM_{server} is solely responsible for the resource it encapsulates. This shows that even though this intermediate SM performs a typical OS task – shared resource management – it differs significantly from a conventional omnipotent trusted kernel software layer.

Secure resource sharing for PMAs thus requires a disjoint set of *self protecting OS modules*. Every such module encapsulates and controls access to a platform resource (e.g. a protected memory buffer, a file system, a keyboard, a network interface, etc.). This way, client SMs that use its services are offered availability and access control guarantees.

Zero-Software Microkernel. The idea of implementing the OS as a set of non-privileged modules echoes the widely known microkernel approach [12,13]. In

a microkernel architecture all non-essential OS services – such as device drives, file systems, process management, etc. – are implemented as regular user programs, known as *servers*. The main task of the privileged microkernel is to separate the applications from each others and provide inter-process communication between them. User programs and servers always communicate indirectly through the microkernel. From a security perspective, a true microkernel limits the TCB by reducing the kernel's size. The actual OS services are implemented in user space on top of these abstractions.

There is no consensus on which mechanisms should be implemented in the microkernel. In a way, the Sancus platform is a truly minimal zero-software microkernel that provides two basic mechanisms to SMs: memory isolation and authentication. The question then becomes whether such a zero-software microkernel is sufficient to securely implement OS-like services on top. In this respect, Liedtke [12,13] identifies only three basic abstractions for his minimalist second generation L4 software microkernel: address spaces, inter-process communication and threads. He argues a microkernel has to "hide the hardware concept of address spaces, since otherwise, implementing protection would be impossible." [12]. The Sancus platform provides fine-grained hardware-enforced protection domains in a single-address-space. Furthermore, Liedtke identifies the need for a microkernel to "establish a communication channel which can neither be corrupted nor eavesdropped" and states "uids are required for reliable and efficient local communication" [12]. This clearly resembles Sancus' hardware-supplied unique SM IDs and attestation features.

Our protected file system prototype, SM_{sfs}, demonstrates Sancus' hardware-enforced mechanisms are sufficient to realise SM-grained logical file access restrictions. SM_{sfs} offers security guarantees similar to those of user-space file system server which is effectively shielded from other protection domains. Moreover, a client is ensured confidentiality and integrity when communicating with SM_{sfs}. Importantly, Sancus realises these security guarantees without any trusted software layer. Its hardware-enforced protection scheme indeed makes an omnipotent kernel layer inherently impossible.

3.3 Application Scenarios

The problem domain of low-end embedded devices is characterised by conflicting interests between economic considerations on the one hand and security requirements on the other. Sancus presents the SM as the unit of lightweight memory isolation and authentication. Our protected file system SM_{sfs} module supplements these hardware-enforced security properties with logical file access control guarantees by means of an explicit software TCB. It thus shows the feasibility of securely sharing system resources on a per-SM-basis.

Protected Shared Memory. Being able to pass a moderate sized buffer securely between protection domains is useful in many contexts. A first scenario concerns parameter passing of large values. Indeed, one can only pass parameters securely through a limited number of CPU registers when calling an SM [16].

Protected shared memory is also useful in the context of secure I/O. Recall from Sec. 2 that an SM can be provided with exclusive access to a MMIO peripheral. As an example, a keyboard driver module $SM_{keyboard}$ could offer an entry function to get an input line confidentially from the user. The module may then use protected shared memory to pass the result to a client SM.

Secondary Storage. Several authors identify an emerging application area for embedded platforms using secondary storage file systems [6,5,20]. In a multi-stakeholder model with software extensibility by multiple untrustworthy vendors, fine-grained access control for secondary storage resources is essential. Consider for example a low-end extensible wearable device. One application could save sensitive medical logs in the file system; another one could simultaneously use the file system to save privacy-sensitive data such as environment sensor data, recordings, GPS locations, etc. Needless to say reliable and fine-grained memory protection and file access control is imperative in such a system.

4 Design and Implementation of a Protected File System

In this section, we present a protected file system for the Sancus platform [16]. The file system is encapsulated in its own SM_{sfs} protection domain with exclusive access to the storage device, ensuring file system integrity. Furthermore, our file system realises SM-specific access control, allowing fine-grained access control policies for logical file sharing between SMs.

4.1 Layered Design

The protected file system depicted in Fig. 1 features a layered design with a *front-end* access control layer deciding access to a private *back-end* software layer, encapsulating the actual resource. From the point of view of the front-end, the back-end is an abstract Contiki File System (CFS) interface implementation that can be plugged in when compiling the SM_{sfs} module. We provide two different back-end implementations. Sec. 4.3 discusses an implementation that operates on a Sancus-protected memory buffer, allowing a form of protected shared memory between SMs. Sec. 4.4 plugs in a real-world embedded flash file system, realising SM-grained protection on a shared system resource.

From a security perspective, the front- and back-end are merely a logical structure, as the entire file system runs in a single protection domain SM_{sfs}. The front-end offers the public interface (i.e. SM_{sfs}'s entry points) towards clients, whereas the back-end is called through private non-entry functions. As the PMA hardware guarantees a protection domain can only be entered from its predefined entry points, a client is effectively prohibited from bypassing the access-control front-end and calling the back-end directly.

The division of responsibilities between the front- and back-end is as follows. The front-end presents a transparent UNIX like file system interface towards client SMs to provide them with the concept of a contiguous logical file with

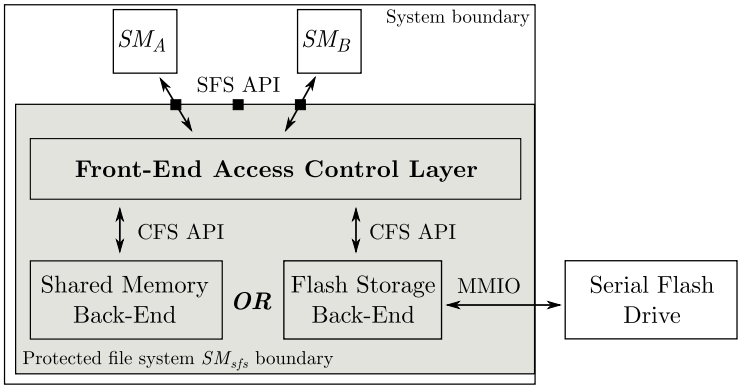

Fig. 1. Our protected file system SM_{sfs} module consists of a generic public front-end access control layer controlling access to a pluggable private back-end software layer, encapsulating the actual resource.

offset-addressable content. Internally however, the front-end is only concerned with SM-oriented access control policies and maintains the data structures to do so. It relies on the back-end CFS implementation for the concept of a logical file. The back-end in its turn encapsulates the actual file system implementation and is completely unaware of any access control going on. It is important to note here that the front-end has no notion of persistence and stores all its access control data structures in volatile protected memory. Our SM_{sfs} prototype does not support persistent SM-grained file protection (c.f. Sec. 6) since it uses Sancus' unique hardware IDs that do not last over multiple boot cycles [16].

4.2 Generic Front-End Access Control Layer

The front-end is conceived as a wrapper implementation that associates an access control list (ACL) of `(ID, permissions_flag)` pairs per logical file to validate the caller's permissions before passing the call to the back-end.

Software-Module-Grained Access Control. Recall from Sec. 2 that the IDs, uniquely identifying a Sancus module within one boot cycle, are inherently unforgeable as they are exclusively managed by hardware. They can therefore safely be used for subsequent client authentications in a software layer. Essentially, the front-end accomplishes its access control guarantees through the `sancus_get_caller_id` hardware instruction, which it uses to reliably retrieve the ID of the client – i.e. the SM that entered the currently executing module.

To realise our protected file system prototype SM_{sfs}, we build upon Sancus' hardware-enforced security guarantees in two ways. On the one hand, Sancus' memory isolation techniques grant SM_{sfs} exclusive access to its back-end resource. On the other hand, Sancus' SM identification scheme provides SM_{sfs} with a reliable client authentication mechanism that allows implementing a thin software layer to realise flexible access control policies for its private back-end resource.

Interface. We based our Sancus File System (SFS) interface on the UNIX-like Contiki File System (CFS) interface [20], modifying it where needed and extending it with SM-specific access control functions. Specifically, we had to replace the `cfs_read` and `cfs_write` functions, requiring a pointer to an unprotected memory buffer and a length argument, with `sfs_getc` and `sfs_putc` functions, which pass the arguments and return values securely through CPU registers. For the same reason we had to replace file name strings with single chars.

In addition, our interface supports SM-specific access control. Using the `sfs_chmod` function, the software module that first created a file can assign or revoke fine-grained permissions for a specific SM via a bit flag. Currently our prototype supports read-only, write-only and read-write permissions, but due to the generic access control scheme, more advanced policies such as append-only could be added relatively easy. Client SMs open files through a modified `sfs_open` function, requiring a permissions flag argument and an initial size hint which is passed to the back-end.

Data Structures. Our prototype stores all access control data structures in its protected private data section. It employs a linked list for logical files, each with a corresponding SM-grained permission ACL. This allows a two phase permission lookup procedure when specifying a file by name. The file list is first traversed to locate the file, using the name as a key. Thereafter, the corresponding ACL is searched using the calling SM's ID as a key. To speed up future accesses, using a file descriptor, we employ a fixed-sized file-descriptor-indexed array with pointers to the corresponding ACL entry.

On each function call, before translating the call to the CFS back-end, the front-end validates the caller's permissions. If the caller passes a file descriptor, the implementation first checks whether it is in the expected range and points to an ACL entry that belongs to the caller. Furthermore, to allow safe revocation of earlier assigned permissions, SM_{sfs} closes any remaining open file descriptors when revoking a permission – as opposed to the POSIX standard [10] which leaves such behaviour implementation-defined.

As Sancus [16] requires the protected memory section of an SM to be fixed-sized during the SM's lifetime, SM_{sfs} should fulfil its own dynamic protected memory requirements. To do so, our implementation enforces a maximum number of open file descriptors, pre-allocates a fixed number of file and permission structs at compile time and maintains them in a free list at run time. When running out of protected memory, the front-end rejects requests to create additional files.

4.3 Protected Shared Memory Back-End

In the protected shared memory implementation, the back-end operates on a fixed-sized Sancus-protected memory buffer. Internally, we use a dynamic memory allocation `malloc` implementation on this buffer, allowing clients to transparently claim a portion of the buffer through a UNIX-like API.

Logical files in the protected shared memory back-end have a fixed size during their lifetime. When creating a new file, the implementation uses the initial size

argument to allocate a buffer of the corresponding size. From then on, it does proper bounds checking, refusing to seek beyond the buffer's end.

Files are arranged in a linked list, each element containing a pointer to a location inside the private `malloc` buffer and the corresponding size. As in the front-end, we maintain a file-descriptor-indexed array to speed up common file operations and to store the current client-specific logical file offset. This bookkeeping information must also reside in protected memory. To support a dynamic number of logical files, the prototype implementation allocates the required structs using its own protected `malloc` buffer.

4.4 Protected Shared Flash Storage

The research presented here adopts Contiki's open source Coffee FS [20] as our case study flash file system back-end. Coffee FS is highly optimised for small flash memories, requires a small and constant RAM footprint per open file and does not provide any existing file protection mechanism.

The shared flash storage back-end introduces the important issue of *secure peripherals* [11]. Indeed, SM_{sfs} should be provided with exclusive access to the flash drive to ensure file system integrity and confidentiality. For peripherals that are being accessed through the memory address space, Sancus' program counter based memory access control scheme grants a dedicated driver SM exclusive access to a resource by including the relevant MMIO addresses in its private data section [16]. The driver module then mutually authenticates with SM_{sfs}, using attestation as discussed in Sec. 2, to realise end-to-end file system protection.

5 Experimental Evaluation

In this section we evaluate the protected file system SM_{sfs} prototype; our implementation and evaluation suite are available online. We discuss runtime overhead as well as the induced memory footprint and code size. We define total runtime overhead from a client SM's perspective as the additional number of CPU cycles needed to call an SM_{sfs} entry function, compared to calling the respective function of an unprotected file system. Furthermore, we split the overall overhead into a Sancus-dictated component, induced by switching Sancus protection domains, and an implementation-dependent component caused by the access control layer. Finally, we provide the relative overhead for the protected shared memory and Coffee flash file system back-ends.

All experiments were conducted on a Sancus-enabled MSP430 FPGA running at 20 MHz. The FPGA is connected to a Micron M25P16 serial flash drive, using the Coffee file system from Contiki release 2.7. For technical details on the MSP430 and Sancus extensions we refer the reader to [16].

Sancus Protection Domain Switching. As explained in Sec. 2, SMs need entry and exit code stubs that take care of private call-stack switching and clearing of CPU registers to avoid leaking of confidential data. These code stubs

thus incur overhead for function calls that switch protection domains. The exact number of cycles needed for such a function call varies with the number and size of the arguments and return value. Calling an unprotected function from within a module SM_A takes between 120 and 170 cycles, whereas calling an SM_B entry function from within SM_A requires between 210 and 280 cycles.

These results indicate an additional Sancus-dictated overhead of roughly 100 cycles for client SMs calling our protected SM_{sfs} module, as opposed to calling an unprotected file system. Note that this overhead is solely caused by encapsulating the file system in its own protection domain SM_{sfs}, independent from any additional access control logic.

Access Control Overhead. We first provide micro benchmarks of the access control front-end layer. The last column of Tab. 2 shows the total number of CPU cycles needed for a protected client SM_A to call our protected file system SM_{sfs} configured with a dummy back-end. The "Sancus Induced" column lists the number of cycles thereof caused by calling the respective Sancus entry function, depending on the number of arguments. These numbers are responsible for the vast majority of cycles, illustrating how SM_{sfs} realises SM-grained access control policies through a thin software layer on top of Sancus.

Table 2. The number of cycles needed for SM_{sfs} configured with a dummy back-end, assuming a single open file with one ACL entry. The "Sancus Induced" column lists the number of cycles needed to call the respective SM_{sfs} entry function. The next two columns show the overhead of the front-end and the last column lists the summation.

SFS API		**Sancus**	**Front-End Induced**		**Total**
function	case	**Induced**	ACL checks	back-end call	
`format`		211	181	17	409
`open`	creat	279	120	69	468
`open`	exist	259	95	69	423
`seek`		259	18	58	335
`getc`		229	46	59	334
`putc`		234	55	63	352
`close`		229	56	24	309
`remove`		226	138	27	391
`chmod`	add	247	120	0	367
`chmod`	revoke	247	158	0	405

We further detail the overhead induced by the front-end. The "back-end call" column of Tab. 2 lists the number of cycles needed by the front-end to call the back-end – the downside of a layered design. The "ACL checks" column shows the number of cycles needed to traverse the access control data structures, in the case of a single file and ACL entry. The impact of using the file-descriptor-indexed array is clearly visible, resulting in a constant and low access control overhead for the functions `seek`, `getc`, `putc` and `close`. As explained in Sec. 4.2, our prototype uses linked lists, resulting in a linear growing access control overhead

for functions without a file descriptor. We experimentally verified the worst-case overhead indeed grows linearly with a reasonable factor of about 12 extra cycles per additional logical file or ACL linked list entry.

The memory overhead of our SM_{sfs} prototype is bounded at compile time by pre-allocating the file descriptor array and a maximum number of structs for logical files and ACL entries, which is common practice in embedded file systems (as in the Coffee back-end). Both structs occupy 6 bytes. In our test set up, we configured the SM_{sfs} module with a maximum number of 10 ACL entries, 5 files and 8 file descriptor entries, resulting in a total memory usage of 106 bytes. In terms of code size, the access control layer of SM_{sfs} occupies 1.9 KB, whereas the Coffee back-end requires 5.3 KB. Our front-end access control layer thus increases the code size with a factor of 0.36.

Protected Shared Memory Back-End. To investigate the runtime overhead of the protected file system module SM_{sfs} configured with a shared memory back-end, we compare it to the case where two SMs communicate via an unprotected dynamically allocated shared memory buffer in the single-address-space. The "shm" column of Tab. 3 thus shows our baseline, i.e. the number of cycles needed to create a shared buffer of size 100 via an unprotected `malloc` call, read/write a character and `free` it.[1] The next two columns list the number of cycles needed for our protected shared memory SM_{sfs} module and the absolute overhead.

The key thing to note here is that, once the unprotected dynamic memory is allocated, read and write accesses are equivalent to normal memory accesses and thus require very few cycles. Our SM_{sfs} protected shared memory set up however adds a level of indirection, implying a huge relative overhead for memory accesses. Moreover, setting up the memory buffer takes longer as the meta data structures should be initialised and clients have to open the logical file before accessing it. Emulating flexible access control policies on top of Sancus' native protection model is however for the moment the only way of realising complex protected interactions between SMs.

Protected Shared Flash Storage Overhead. We investigate the runtime overhead of our protected SM_{sfs} file system prototype on top of Contiki's Coffee FS [20], a typical real-world embedded flash file system. The "coffee" column of Tab. 3 lists our baseline, i.e. the total number of CPU cycles needed for a protected client SM_A to call an unprotected Coffee flash file system. The "sfs-coffee" column shows the number of cycles needed by SM_A to call our SM_{sfs} protected file system module, configured with a Coffee back-end. Note that these numbers reflect the ideal case where the front-end as well as the back-end implementation and flash driver share the same protection domain SM_{sfs}. In our test set up the Coffee file system and the flash driver operate in unprotected mode, see also Sec. 6. We thus arrived at the presented data by carefully subtracting the fine-grained overhead of switching Sancus protection domains.

[1] Recall from Sec. 4.2 that we cannot support a multi-byte read/write API. Reading/writing a buffer will thus need multiple calls to `getc`/`putc`.

Table 3. The overhead for a client SM_A that uses SM_{sfs}'s services for each back-end, assuming a single open file with one ACL entry. The "Shared Memory" columns list from left to right: the number of cycles needed by SM_A to use unprotected dynamic memory, SM_{sfs} with a shared memory back-end and the absolute overhead. The "Flash Storage Back-End" columns list from left to right, the number of cycles needed for SM_A to call: an unprotected Coffee file system, SM_{sfs} with a Coffee back-end; the absolute and relative overhead and the overhead percentage induced by the ACL lookup.

API		**Shared Memory**			**Flash Storage Back-End**				
		baseline		*overhead*	*baseline*		*overhead*		
function	case	shm	sfs-shm	shm-abs	coffee	sfs-coffee	abs	rel	acl
`format`		-	584	584	360 E6	360 E6	286	0	63
`open`	creat	192	1,326	1,134	76,133	76,436	303	0	40
`open`	exist	-	706	706	2,604	2,862	258	10	37
`seek`		-	322	322	430	594	181	44	10
`getc`		2	342	340	902	1,081	179	20	26
`putc`		4	351	347	1,288	1,485	197	15	28
`close`		-	539	539	317	498	181	57	31
`remove`		192	670	478	8,033	8,293	260	3	53
`chmod`	add	-	367	367	-	367	367	-	33
`chmod`	revoke	-	405	405	-	405	405	-	39

The "abs" column of Tab. 3 lists the absolute number of overhead cycles caused by the protected file system implementation, as compared to the unprotected Coffee set up. To interpret these numbers, the next columns provide the relative overhead and the percentage of the total overhead that is caused by the access control front-end implementation. These results indicate the overhead of protected resource sharing on top of a real-world flash file system is reasonable. Due to the delay of the flash I/O and the file-descriptor-indexed array, the relative number of additional cycles remains limited for commonly used file operations: under 20 % for `getc` and `putc`; it even drops to zero for I/O-heavy operations such as `format`, `creat` and `remove`. Moreover, the additional SM-specific `chmod` access control function consumes a number of cycles of the same magnitude as the unprotected in-memory file operations, such as `seek`. Finally, the front-end access control software layer shows to be lightweight in the sense that over half of SM_{sfs}'s overhead – in the case of a single file and ACL entry – can be attributed to calling the respective Sancus entry function and the back-end function call.

Comparing the two back-ends reveals another characteristic of SM interactions: the relative overhead of switching protection domains decreases as the execution time of the callee module increases. Specifically, the relative overhead of SM_{sfs} with a flash back-end is reduced by the flash I/O delay, whereas fast unprotected memory access aggravates overheads in the protected shared memory case.

6 Discussion

In this section, we discuss the security guarantees and limitations of our protected file system SM_{sfs} prototype.

Trusted Computing Base. Our SM_{sfs} module builds upon Sancus' existing hardware primitives [16] to supplement the hardware-enforced security guarantees of its clients with logical file access restrictions. Clients using SM_{sfs} naturally incorporate it in their TCB. Our approach differs significantly from a traditional trusted OS computing base however for two major reasons.

Firstly, *only* client SMs using SM_{sfs} have to trust SM_{sfs} and Sancus offers strong authentication to verify SM_{sfs}. A client can attest an SM, guaranteeing that, i.e., SM_{sfs} has not been tampered with and was loaded correctly, with exclusive access to the MMIO flash drive addresses. This results in a small explicit TCB, as opposed to the implicit TCB induced by an omnipotent trusted kernel.

Second, the SM_{sfs} module is solely entrusted its dedicated file system task, echoing the well known principle of least privilege. Thus, a faulty SM_{sfs} module can only tamper with or leak the file system data it is entrusted. A client SM still preserves exclusive access to its private section. In this, SM_{sfs}'s security guarantees are similar to those of a microkernel file system running in user space as it is shielded from other protection domains. Notably, Sancus does not rely on any trusted kernel software layer to enforce this separation.

Limitations. We acknowledge several limitations in our SM_{sfs} prototype. Firstly, in our test set-up, the Coffee file system back-end runs in unprotected mode. We believe that protecting the Coffee implementation by an SM is relatively easy, albeit out of scope for the work presented in this paper.

A second limitation concerns the protected flash driver. Currently Sancus' program counter based memory access control hardware logic only allows a single contiguous private data section per Sancus module [16]. This implies that a module including a MMIO address range in its private data section, cannot at the same time have protected data. Moreover, as it cannot safely provide the stack needed by higher level programming languages, its corresponding code section should be entirely implemented in assembly. We therefore need a separate dedicated flash driver SM, exclusively communicating with SM_{sfs}. From a security perspective, there is no real issue here, but switching protection domains decreases the performance, as explained in Sec. 5. In a real-world set up however, Sancus' program counter based access control logic [16] could relatively easy be extended to allow a MMIO address range as well as another contiguous protected address range in a single protection domain SM_{sfs}.

Finally, SM_{sfs} ensures confidentiality and integrity of logical files as long as it is up and running (which can be verified by the client), but does not persist these guarantees across reboots. Indeed, since the IDs assigned to SMs by Sancus, do not persist after reboots (see Sec. 2), they may change when redeploying an SM. We argue that extending SM_{sfs}'s file protection guarantees across reboots is non-trivial, as anything could happen between crashing of the node and successful redeployment of SM_{sfs}. In this respect, our protected file system does also not protect against physically removing and reading out the flash drive. This matches Sancus' attacker model [16] which does not consider attackers with physical access to the hardware. The only way to protect against such attacks and to

support persistent file protection would be to encrypt all data on the flash disk with SM_{sfs}'s Sancus-provided private key. Such an approach would however dramatically reduce performance, especially since all data is transferred safely through CPU registers on a byte-per-byte basis. Moreover, there would be little advantage over the situation where clients encrypt the data themselves before passing it to SM_{sfs} or even an unprotected file system.

We therefore consider our protected file system SM_{sfs} module as a way for SMs to extend their fixed sized private data section considerably, while at the same time offering flexible access control guarantees. In this respect, it could be an interesting future work direction to ensure the hardware automatically clears the flash drive on system boot – even before SM_{sfs} is deployed – to enforce the non-persistence of file system data.

7 Conclusion

Low-end embedded devices are becoming increasingly present and interconnected in our everyday lives. Adequate software isolation for these platforms is crucial in a multi-stakeholder context. In this perspective, PMAs offer strong hardware-enforced memory isolation and authentication guarantees, but cannot realise flexible access control policies for shared system resources. SMs should either claim the resource for themselves or rely on the services of an untrusted OS when interacting with the outside world.

In this paper we presented a protected file system SM_{sfs} module that builds upon existing PMA hardware primitives to construct a software layer that realises access control, i.e. logical file protection guarantees for client SMs. In a broader perspective, this demonstrates the feasibility of supplementing the hardware-enforced security properties offered by PMAs with SM-grained access control guarantees enforced by a protected software TCB.

While our implementation is based on Sancus [16], a hardware-only TCB for lightweight embedded microcontrollers, our approach is fairly general and can be implemented with other PMAs that provide (1) memory isolation, (2) attestation guarantees and (3) exclusive use of MMIO ranges. Yet, to the best of our knowledge, Sancus is the only PMA satisfying all these requirements in the embedded world. In server and desktop computing, our approach can be implemented using a trusted hypervisor and a PMA such as Intel's SGX [15]. Since SGX enclaves cannot claim MMIO ranges directly, a rather large software TCB would be necessary.

In the future we will further investigate the effectiveness and efficiency of our access control mechanism based on extended evaluation scenarios that allow for meaningful macro-benchmarks. A particularly interesting scenario would be to provide access control for I/O devices connected to a peripheral bus.

Acknowledgements. This research is partially funded by the Research Fund KU Leuven, and by the FWO-Vlaanderen. Job Noorman holds a PhD grant from the Agency for Innovation by Science and Technology in Flanders (IWT).

References

1. Agten, P., Strackx, R., Jacobs, B., and Piessens, F. Secure compilation to modern processors. In *IEEE CSF 2012*, pp. 171–185. IEEE, 2012.
2. Bach, M. J. *The design of the UNIX operating system*, vol. 5. Prentice-Hall, 1986.
3. Berman, A., Bourassa, V., and Selberg, E. TRON: Process-specific file protection for the UNIX operating system. In *USENIX TCON'95*, pp. 165–175. USENIX Association, 1995.
4. Cao, Q., Abdelzaher, T., Stankovic, J., and He, T. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *IPSN'08*, pp. 233–244. IEEE, 2008.
5. Escolar, S., Carretero, J., Isaila, F., and Lama, S. A lightweight storage system for sensor nodes. In *PDPTA*, pp. 638–644, 2008.
6. Farooq, M. O. and Kunz, T. Operating systems for wireless sensor networks: A survey. *Sensors*, 11(6):5900–5930, 2011.
7. Gay, D. Matchbox: A simple filing system for motes. `http://www.docs.tinyos.net/tinyos-1.x/doc/matchbox.pdf`, August 21 2003.
8. Granjal, J., Monteiro, E., and Silva, J. S. Security in the integration of low-power wireless sensor networks with the internet: A survey. *Ad Hoc Networks*, 24, Part A(0):264–287, 2015.
9. Grünbacher, A. Posix access control lists on linux. In *USENIX TCON'03*, pp. 259–272. USENIX Association, 2003.
10. IEEE. Std 1003.1. `http://pubs.opengroup.org/onlinepubs/009695399/`, 2004.
11. Koeberl, P., Schulz, S., Sadeghi, A.-R., and Varadharajan, V. Trustlite: a security architecture for tiny embedded devices. In *EuroSys'14*, pp. 10:1–10:14. ACM, 2014.
12. Liedtke, J. On μ-kernel construction. In *SOSP'95*, pp. 237–250. ACM, 1995.
13. Liedtke, J. Toward real microkernels. *Comm. ACM*, 39(9):70–77, 1996.
14. Lopez, J., Roman, R., Agudo, I., and Fernandez-Gago, C. Trust management systems for wireless sensor networks: Best practices. *Comput. Commun.*, 33(9):1086–1093, 2010.
15. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 10:1–10:1. ACM, 2013.
16. Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX SEC'13*, pp. 479–494. USENIX Association, 2013.
17. Roman, R., Najera, P., and Lopez, J. Securing the internet of things. *Computer*, 44(9):51–58, 2011.
18. Strackx, R., Noorman, J., Verbauwhede, I., Preneel, B., and Piessens, F. Protected software module architectures. In *ISSE'13*, pp. 241–251. Springer, 2013.
19. Strackx, R., Piessens, F., and Preneel, B. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*, vol. 50 of *LNICST*, pp. 344–361. Springer, 2010.
20. Tsiftes, N., Dunkels, A., He, Z., and Voigt, T. Enabling large-scale storage in sensor networks with the coffee file system. In *IPSN'09*, pp. 349–360. ACM/IEEE, 2009.