Hardware/Software Co-designed Security Extensions for Embedded Devices

Maja Malenko(✉) and Marcel Baunach

Institute of Technical Informatics, Graz University of Technology, Graz, Austria
{malenko,baunach}@tugraz.at

Abstract. The rise of the Internet of Things (IoT) has dramatically increased the number of low-cost embedded devices. Being introduced into today's connected cyber-physical world, these devices now become vulnerable, especially if they offer no protection mechanisms. In this work we present a hardware/software co-designed memory protection approach that provides efficient, cheap, and effective isolation of tasks. The security extensions are implemented into a RISC-V-based MCU and a microkernel-based operating system. Our FPGA prototype shows that the hardware extensions use less than 5.5% of its area in terms of LUTs, and 24.7% in terms of FFs. They impose an extra 28% of context switch time, while providing protection of shared on-chip peripherals and authenticated communication via shared memory.

Keywords: Memory protection · Resource protection · Inter-task communication · RISC-V · MPU

1 Introduction

The number and heterogeneity of embedded devices which are emerging with the rise of the IoT is increasing massively. Their span ranges from very small and lightweight devices up to very complex computer systems, many of which implement security and safety critical operations [7,11]. Therefore, they must offer some form of protection mechanism which will ensure isolated execution of applications. There is an extensive research in this area at the moment, focused on finding lightweight solutions and protection mechanisms. A lot of concepts have been developed, each with a different purpose, but so far, none of them has solved all the problems.

In this paper our focus are low-cost microcontrollers (MCUs) which operate in a single physical address space. Devices that are based on such MCUs are especially susceptible to attacks, intentional or not, and an efficient isolation mechanism is necessary to protect them. In order to reduce cost and energy usage, due to their lightweight nature, these devices often lack any form of protection. Thus, even though tasks might be designed to cooperate, not trying to intentionally harm each other, a small bug in one of them can potentially corrupt the whole system. The security solution must be implemented in a very

M. Schoeberl et al. (Eds.): ARCS 2019, LNCS 11479, pp. 3–14, 2019.
https://doi.org/10.1007/978-3-030-18656-2_1

efficient manner, regarding both memory and hardware consumption. If real-time constraints are present, the security implementation should not impose significant runtime overhead, but must still provide integrity and confidentiality guarantees. Many hardware and software-based security architectures have recently emerged, isolating the execution of sensitive operations on a wide range of devices. They all differ in the type of devices they are tackling and the amount of hardware and software in their Trusted Computing Base (TCB). In higher-end systems a very common approach is to have a trusted operating system which uses a Memory Management Unit (MMU) to isolate processes in their private virtual address space. This approach requires a lot of hardware and has big TCB. Researches recently have been working on developing Protected Module Architectures (PMAs) [15] as a more efficient and lightweight approach for memory isolation in a shared physical address space, using small-sized TCBs [5,6,8,12,14]. Some of them completely exclude the software from the TCB, while others implement just the most necessary software operations.

We propose a hardware/software co-designed embedded platform which provides dependability at low-cost. The architecture is based on a RISC-V *vscale*[1] implementation on which a minimal microkernel (*Smart*OS) [4] is running. All memory accesses are mediated by a tailored Memory Protection Unit (MPU) which provides three essential isolation concepts: isolation of private code and data regions of individual tasks, protecting the usage of shared on-chip memory-mapped peripheral devices from unauthorized access, and providing protected communication between tasks.

The paper is structured as follows. First we describe the protection concept and introduce the platform used for implementation and evaluation (Sect. 2). Then, a detailed description of each protection mechanism is given along with the hardware and software implementation (Sect. 3). Next, several test cases and measurement results are presented (Sect. 4). We relate our work to similar approaches (Sect. 5) and discuss the differences and benefits from our approach. Finally, we draw a conclusion (Sect. 6).

2 Concept and Platform Overview

The goal of most embedded computer systems is to run applications securely and efficiently. To achieve this goal both the hardware and the software should cooperate as effectively as possible. However, especially in today's security-related research the co-design aspect of hardware and software seems to be missing. In this work we explicitly target low-cost embedded devices and we try to close this gap by developing a hardware/software co-designed memory protection architecture.

In a multitasking environment where concurrent tasks reside in a single address space and extensively use shared resources, attacks from malicious or malfunctioning tasks are expected to happen. We propose an inexpensive and

[1] https://github.com/ucb-bar/vscale.

effective approach for isolated execution of preemptable tasks in an environment with frequent interactions. In order to have an isolated task execution, we must ensure several properties: protect the internal state of the running task (private code and data), ensure correct resource usage by enforcing access policies of the resources the task owns, and finally, authenticate communication between cooperating tasks using protected shared memory. The ultimate goal of our hardware/software co-designed architecture is to achieve efficient (in terms of hardware and software) task-based protection for low-cost MCUs, at the same time trying not to violate the real-time characteristics of the underlying OS by keeping the context switch time constant, and trying to avoid expensive system calls as much as possible. Thus, we implement kernel-based security mechanisms, which are then enforced by lightweight hardware extensions.

2.1 RISC-V

The MCU we are using is based on a *vscale* processor, which is a single-issue, three stage pipeline implementation of the RISC-V ISA [16,17]. We decided to use a RISC-V-based MCU mainly because of its simplicity, minimalism, openness, and room for extensions. The *vscale* implementation already comes with two privilege modes, which are the main prerequisite for implementing protection. Tasks can request services from the kernel only by system calls, which trap into machine mode. The MCU includes several on-chip peripherals, which, as in most embedded devices, are mapped into a single address space with the memories, and if no memory protection is available they are fully accessible to everyone. The architectural overview of the system is given in Fig. 1.

2.2 *Smart*OS

*Smart*OS is a small, modular, real-time operating system suitable for low-cost embedded devices [4]. It is ported to *vscale* and uses two operational modes: the kernel runs in privileged machine mode, while tasks as well as the libraries run in user mode. Tasks are preemptive, use individual stacks and execute all API functions in their context. The kernel uses its own stack and is responsible for priority-aware scheduling, system call execution, interrupt handling, dynamic resource management, and inter-task synchronization using events. The kernel code and data, including the control blocks for tasks, events, and resources, can only be accessed using system calls, which are atomic and executed in machine mode.

In order to properly design a secure system where tasks and OS coexist in a single address space, we must ensure that an incorrect (or malicious) task cannot interfere with the proper operation of the system and other tasks. That is why the linker is instructed to efficiently organize the memory map into regions, as shown in Fig. 1. Each task is associated with its individual code and data (stack) regions, as well as a shared region for API functions. On demand, the OS also grants access to additional regions for accessing peripherals and shared memory for inter-task communication.

Resource Management. In order to protect shared resources (i.e. peripherals) from unsynchronized access by several tasks and enable collaborative resource sharing, *Smart*OS uses a resource management concept, based on the Highest Locker Protocol (HLP) [18], which due to its simple implementation, is frequently used in RTOSs. It allows dynamic access coordination to temporarily shared, but exclusive resources and prevents a resource from being used by a task as long as it is allocated to another task. Each resource at compile time or system startup receives a ceiling priority, which is the highest priority of all registered tasks that announced the usage of that resource. As soon as a task successfully allocates a resource, its priority is raised to the resource's ceiling priority.

The resource concept in *Smart*OS enables, but does not enforce synchronization on physical resources without hardware support. The kernel can not prevent (neither detect, nor block) illegal access attempts to a peripheral by a task which does not hold it as a resource. To avoid this, instead of a very slow approach of allocating/deallocating the resource inside each driver function which has direct access to it, we are proposing a hardware-based enforcement of the HLP, which locks peripherals in order to protect them from unauthorized access. Without an additional hardware support there is a risk that even though one task has claimed the resource for itself, another task uses it in an unprotected way.

Inter-task Communication. Tasks in *Smart*OS are not self-contained, they frequently interact with each other and with the environment. For that reason, explicit synchronization between tasks is achieved through events, which can be invoked by tasks or interrupt service routines (ISRs). In this work we are extending the event synchronization concept of *Smart*OS and provide an effective solution for authenticated data exchange between tasks.

3 Implementation Details

In order to achieve flexible memory separation for variable-sized regions, each defined with its start and end address, we are using a segmentation-based memory protection approach. Our MPU is configured with four Control and Status Registers (CSRs), as specified in the RISC-V privilege architecture [17], which hold the address ranges of task's private code and data regions. Two more registers store the address range of the API functions, called libraries in Fig. 1, and are not reprogrammed on every context switch. By configuring few additional registers we also enable efficient and lightweight access protection to shared resources, including memory for communication (see Fig. 1).

3.1 Basic Memory Protection

When a programmer creates a task, she specifies the priority, the amount of stack to be used as well as the entry function of the task. When a task is loaded, the kernel assigns unique code and data (including stack) memory regions, which are stored in the Task Control Block. Since drivers in our system are executed in the

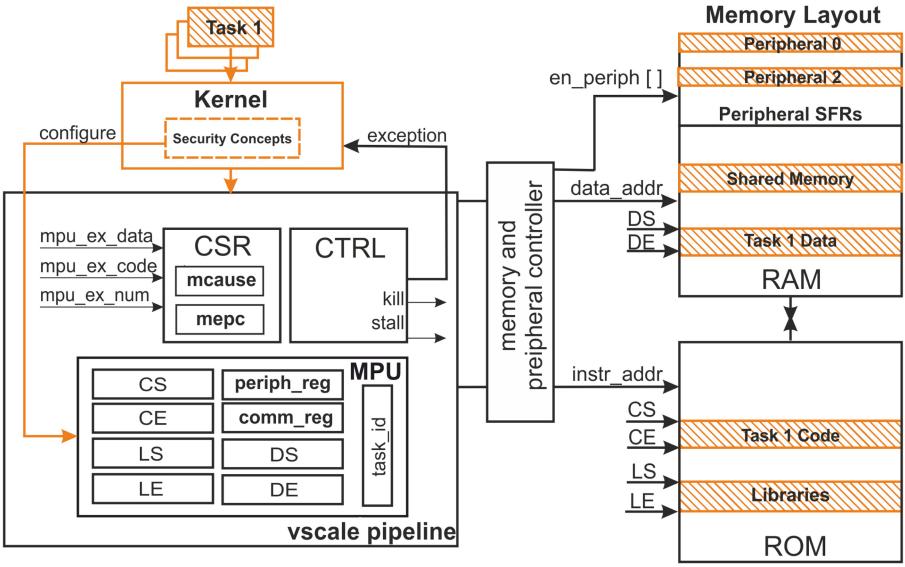

Fig. 1. Architectural overview of the system

context of the currently running task, by placing all API functions sequentially in memory, we reduce the number of MPU registers needed for protecting the shared code area accessible to all tasks.

In *vscale*, in order to achieve basic memory protection, we take advantage of its two privilege levels. The MPU configuration is allowed only in privileged machine mode and is done by the kernel on every dispatch of a newly scheduled task. While in user mode, the MPU monitors all bus activities and raises an exception in case of an access violation. We made several modifications to the *vscale* pipeline by introducing three new exception vectors: instruction access fault, load access fault, and store access fault vector. The MPU's data-access policy implies that memory locations accessed by store and load instructions can be performed only within the task's data regions (including the authorized peripheral and shared memory addresses) which are implicitly non-executable, while the control-flow policy implies that the control-transfer instructions must stay within the memory reserved for the task's code and library regions which are implicitly non-writable.

The MPU is attached to the *vscale* pipeline and the exception signal it produces is then handled by the controller module in the decode/execute stage. The processor invalidates the executing instruction by flushing the pipeline, saves the exception vector number in the *mcause* CSR, the faulty instruction in the *mepc* CSR, and immediately jumps to a predefined exception handler. From there, the kernel handles the exception appropriately.

3.2 Peripheral Protection

In *Smart*OS the kernel's resource management protocol implies that all shared peripherals must be declared as resources. Tasks can request an access and if granted, they can use the peripheral as specified by the HLP protocol. By introducing hardware checks, we force tasks to explicitly request a resource before using it. If a task hasn't previously announced usage of the resource, or if the resource is currently being used by another task, access is not granted (the *en_periph[i]* signal for the particular peripheral i in Fig. 2 is disabled) and a data access exception is raised.

Resources in *Smart*OS are declared in so-called driver constructors, and their declaration is mandatory. Every task announces its resource usage at creation time (*OS_REGISTER_RESOURCE* macro in Fig. 2), in order for the HLP to calculate the resource's ceiling priority, and an MPU bitfield peripheral register is appropriately programmed (*periph_reg* in Fig. 2). Each peripheral that is used by the task is encoded with a '1' in the bitfield register on its specific index. The index of the peripheral is associated with the ordinal number the peripheral has in memory (in our MCU implementation, each peripheral has a fixed memory address range). The order in which resources are used is announced during runtime by two system calls (*getResource()* and *releaseResource()*). Task-awareness is integrated into the hardware by storing the *id* of the currently executing task (*task_id* register in Fig. 2) as well as the *id* of the task which was granted access to a peripheral (*periph_owner* register in each peripheral in Fig. 2).

The MPU needs only one register for protecting up to 32-peripherals, which is sufficient for most small embedded systems. The peripheral access is checked by hardware on each peripheral register access (load or store), which is more efficient (both in terms of execution time and required software) than performing expensive software checks in device drivers.

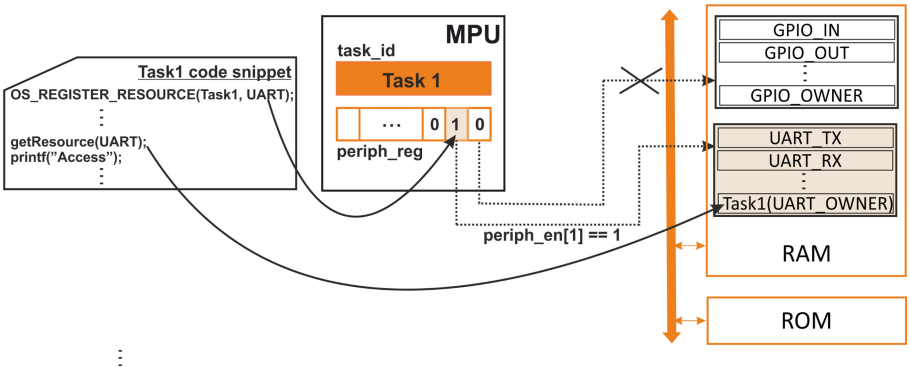

Fig. 2. Protected usage of shared peripherals

3.3 Protected Inter-Task Communication

We implement a shared memory approach for communication between tasks, since it has been proven to be a very versatile architectural choice, mainly because of the constant access time to variable-sized values. The shared memory region is divided into pages of configurable, but mutually equal sizes. Every page is easily indexed with the lower bits of the address (*page_index* in Fig. 3), which are used as the page's position in the MPU bitfield register (*comm_reg* in Fig. 3). For each task, this register has a '1' only if the communication is requested and acknowledged by both sending and receiving tasks. When the sender wants to communicate, it explicitly grants privileges for communication to the receiver task, by specifying the receiver task's id and the size of the message. But, only when the receiver task acknowledges the request, the shared memory region is open for both of them. The system calls (*registerSharedMem()* and *ackSharedMem()*) used for establishing the mutually authenticated communication are shown in Fig. 3, and are used to configure the values which will be placed inside the *comm_reg* register. After successful authentication, every communication between the two tasks is done in user mode, by calling the functions *sendData()* and *receiveData()*, thus preventing expensive system calls. When a task requests memory area for communication, the kernel inspects which pages are still free, and designates them to both communicating tasks. Tasks authenticate each other with their *id-s*, which are unique for each task and are maintained by the kernel. When running out of memory, the kernel rejects any request for new communication, until memory is freed. The nature of communication can be asynchronous in order to prevent blockage due to unresponsive tasks, or can be made synchronous by using the *Smart*OS's event concept.

In hardware, every time a data memory operation is performed in the designated shared memory area for communication, the MPU calculates the page index, checks the *comm_reg* register, and raises a data memory access exception if the executing task has no privileges to use the indexed page. This approach is substantially faster then completely software-based implementation, because no system calls are involved for the communication itself. The more expensive authentication is only done once, before establishing the communication.

4 Test Cases and Performance Evaluation

We implemented the presented protection concept into our research platform consisting of a *vscale*-based MCU with several on-chip memory-mapped peripherals (e.g., UART, GPIO), on which *Smart*OS is running. For the MCU implementation we use the Basys3 Artix-7 FPGA board from Digilent[2].

First, we are going to present two test cases, along with the simulation results[3], which show an invalid way of accessing memory. On the left side of Fig. 4 the disassembly of *task1*'s entry function is shown. In this example *task1*

[2] https://reference.digilentinc.com/reference/programmable-logic/basys-3/start.
[3] Performed by Vivado Simulator 2017.3.

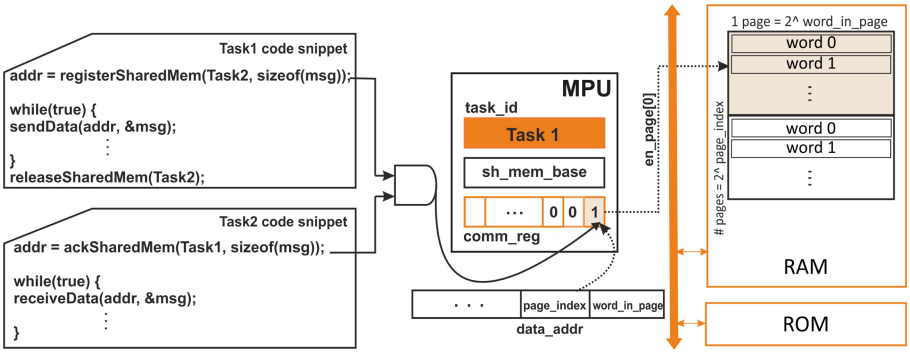

Fig. 3. Protected and authenticated communication between tasks

tries to change the control-flow by jumping into the address space of *task2*. As can be seen from the simulation output, at the time the jump instruction is executed with the address which does not belong to *task1*'s code region, an instruction access fault exception is raised. After the exception, the code continues from a predefined exception handling routine (0x100). The scenario in Fig. 5 shows a task trying to access the *GPIO_OUT* register, when it hasn't requested it before. A load access fault exception is raised, and as in the previous example, the processor jumps to the exception handling routine.

Hardware Footprint. To evaluate the used FPGA resources of our MPU implementation we measured the hardware footprint of the baseline system as well as the one just for the protection hardware components. The overall utilization of lookup tables for logic (LUTs) and flip-flops (FFs) as reported after synthesis by Xilinx Vivado 2017.3 is shown in Table 1.

Execution Time Overhead. Context switches are one of the most costly operating system operations in terms of CPU time. The context frame which is saved on the stack on each context switch when dispatching a newly scheduled task consists of a total of 36 load operations. Out of those, configuring the code and data regions takes only 4, while both peripheral and communication protection information counts for another 4 load operations, giving a total increase of 28%. However, except the initial peripheral allocation and communication authentication which happens at task's load time, during task execution the kernel does not perform any additional checks.

In order to compare the number of clock cycles needed for protected communication, besides the communication using shared memory, we implemented a simple message queue-based communication, as specified in [1]. As shown in Table 2 the number of clock cycles for creating a message queue and registering a shared memory address for same-sized messages are similar, since both are implemented as system calls. However, the number of clock cycles for actual communication is significantly lower when using shared memory, because *sendData()* unlike *xQueuePut()* is not a implemented as a syscall.

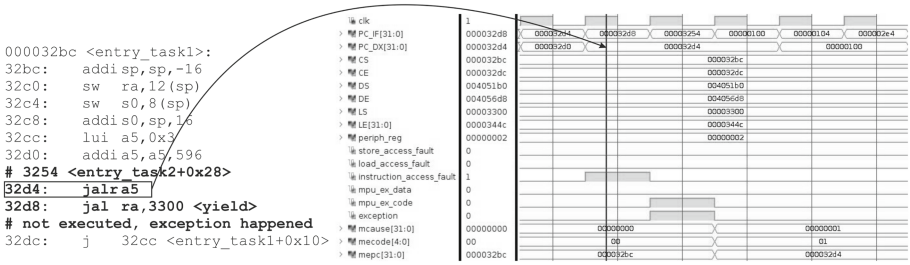

Fig. 4. Invalid control-flow memory access

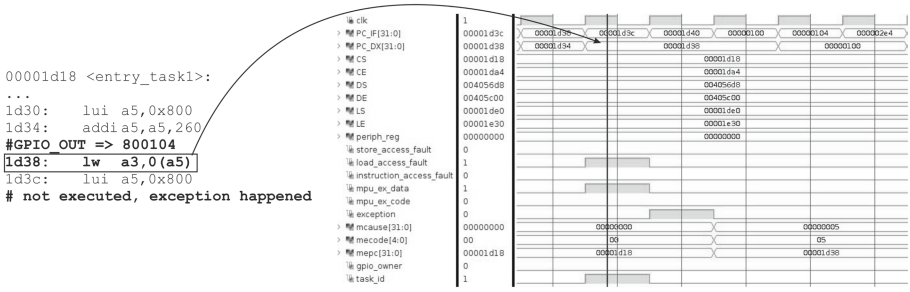

Fig. 5. Invalid GPIO peripheral access

Table 1. Synthesis results of our hardware extensions.

Category	Components	LUTs (Util%)	FFs (Util%)
Baseline system	vscale	2547 (4.02%)	1293 (1.02%)
Our hardware resources	Basic MPU	75 (0.36%)	192 (0.46%)
	Added resource protection	106 (0.51%)	256 (0.62%)
	Added communication protection	140 (0.67%)	320 (0.77%)
	Total % over Baseline system	5.5%	24.7%

Table 2. Clock cycles for protected communication.

Category	System call/Function	Clock cycles
Message queue	xQueueCreate (len, msgSize)	135
	XQueuePut (handle, msg)	115
Protected shared memory	registerSharedMem (taskId, msgSize)	150
	sendData (addr, msg)	30

5 Related Work

We analyzed several security architectures, both from the mainstream computing domain and the embedded domain. On conventional high-end systems, software isolation is usually realized by a trusted operating system, which relies on advanced hardware support in a form of an MMU [13]. Since this mechanism is too costly to be used in embedded systems, a lot of work is recently concentrating on creating Protected Module Architectures (PMAs) which provide cheap, small, and effective isolation in a shared address space, usually with a very small Trusted Computing Base (TCB). Architectures like SMART [9] and Sancus [14], with a zero-software TCB, have a static way of loading self-contained secure modules (SMs), which have very limited access to shared resources. This implies that SMs must claim a resource for themselves in order to have security guarantees on it and are not able to share it with other modules. Even more, Sancus requires that the peripheral is wired in the same contiguous data region of the SM that owns it. Other architectures like TrustLite [12] and TyTAN [5], in order to provide greater flexibility, have created many kernel-like services and put them in the TCB. There is a difference between Trustlite and Sancus in the way they enforce the PC-based memory access control rules. While in Sancus there is a dedicated hardware-based Memory Access Logic (MAL) circuit per SM, which provides the code section with an exclusive access to a single contiguous data section, in TrustLite the access rules are programmed in a fixed-size Execution-aware MPU (EA-MPU) hardware table. Multiple non-contiguous private data sections per trusted module, or protected shared memory between trusted modules can be configured, but such flexibility is limited by the number of entries in the hardware table. Regarding peripheral protection, TrustLite allows limited number of EA-MPU's data regions to be used for securing peripherals, which are allowed for access only to trusted modules. Our peripheral protection approach allows for configuration of up to 32 peripherals which do not have to be mapped in a contiguous memory addresses. ARM TrustZone [2] is another security architecture which separates a computer's resources into a secure and a non-secure world. While the secure world can access everything, the non-secure world is limited only to non-secure memory regions. Peripherals are also split between the two worlds. TrustZone requires a security kernel for managing the secure world, which increases the size of the TCB.

Since *Smart*OS has a very small kernel code base, which performs almost all the functions which in [5] are put in a TCB, we are considering it to be trusted and to be part of our system's TCB. In contrast to the above mentioned architectures, *Smart*OS does not distinguish between secure and non-secure worlds. It equally protects all tasks, not just their own code and data, but it allows for greater flexibility and more fine-grained protection regarding shared peripherals and shared memory for communication. The above mentioned architectures don't do much to improve the security of the software that runs in the secure world, except to prevent unwanted access by normal world software. Therefore, it is the developer who determines that software is trusted, typically through rigorous development processes, testing, and certification.

Regarding the communication between tasks, in many operating systems like L4 [10] and FreeRTOS [3] the communication for security reasons goes through the kernel, which is a very costly operation. In L4, each time a communication is performed, a task loads the receiver's id, the message, and executes a system call. Then, the kernel switches the address space to the receiver task's, loads the sender id, and returns to user mode. Only after that, the receiver task gets the message. On the other hand, in TrustLite, the ability to achieve fine-grained communication via shared memory is limited by the number of MPU regions the platform offers. In order to avoid expensive context switches every time tasks need to communicate, we are proposing hardware-checked authenticated communication, which the kernel establishes only once, and afterwards is performed in user mode. The communication memory that tasks use is configurable and fine-grained, in terms of consecutive memory locations.

6 Conclusion

In this paper we present a hardware/software co-designed architecture for isolated execution of tasks in low-cost embedded devices. Besides the basic isolation of private code and data regions, in an environment where tasks have frequent peripheral access and mutual communication, we allow for efficient protection of shared resources. By installing task-awareness into the MPU, kernel's operations are supported by inexpensive hardware-enforced security checks.

Acknowledgment. This work was conducted within the Lead-Project "Dependable Internet of Things in Adverse Environments", subproject "Dependable Computing" (funded by TU Graz).

References

1. embOS: Real-Time Operating System User Guide and Reference Manual. SEGGER Microcontroller GmbH (2018)
2. ARM Limited. ARM Security Technology - Building a Secure System using TrustZone Technology (2009)
3. Barry, R.: FreeRTOS reference manual: API functions and configuration options. Real Time Engineers Limited (2009)
4. Baunach, M.: Towards collaborative resource sharing under real-time conditions in multitasking and multicore environments. In: ETFA, pp 1–9. IEEE (2012)
5. Brasser, F.F. Mahjoub, B.E., Sadeghi, A.R., Wachsmann, C., Koeberl, P.: Tytan: tiny trust anchor for tiny devices. In: DAC, pp. 34:1–34:6. ACM (2015)
6. Berkay Celik, Z., McDaniel, P., Tan, G.: Soteria: automated IoT safety and security analysis. In: 2018 USENIX Annual Technical Conference (USENIX ATC 2018), Boston, MA, pp. 147–158. USENIX Association (2018)
7. Checkoway, S. et al.: Comprehensive experimental analyses of automotive attack surfaces. In: Proceedings of the 20th USENIX Conference on Security, SEC 2011, Berkeley, CA, USA, p. 6. USENIX Association (2011)

8. Costan, V., Lebedev, I., Devadas, S.: Sanctum: minimal hardware extensions for strong software isolation. In: 25th USENIX Security Symposium (USENIX Security 2016), Austin, TX, pp. 857–874. USENIX Association (2016)
9. Defrawy, K.E., Perito, D., Tsudik, G., et al.: Smart: secure and minimal architecture for (establishing a dynamic) root of trust. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium, pp. 5–8 (2012)
10. Heiser, G., Elphinstone, K.: L4 microkernels: the lessons from 20 years of research and deployment. ACM Trans. Comput. Syst. **34**(1), 1:1–1:29 (2016)
11. Humayed, A., Lin, J., Li, F., Luo, B.: Cyber-physical systems security-a survey. IEEE Internet Things J. **4**, 1802–1831 (2017)
12. Koeberl, P., Schulz, S., Sadeghi, A.-R., Varadharajan, V.: Trustlite: a security architecture for tiny embedded devices. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys 2014, New York, NY, USA, pp. 10:1–10:14. ACM (2014)
13. Maene, P., Götzfried, J., de Clercq, R., Müller, T., Freiling, F.C., Verbauwhede, I.: Hardware-based trusted computing architectures for isolation and attestation. IEEE Trans. Comput. **67**, 361–374 (2018)
14. Noorman, J. et al.: Sancus: low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13), Washington, D.C., pp. 479–498. USENIX (2013)
15. Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., Piessens, F.: Secure compilation to protected module architectures. ACM Trans. Program. Lang. Syst. **37**(2), 6:1–6:50 (2015)
16. Waterman, A., Lee, Y., Asanović, K.: The RISC-V instruction set manual volume i: User-level ISA version 2.2. Technical report, EECS Department, University of California, Berkeley, May 2017
17. Waterman, A., Lee, Y., Asanović, K.: The RISC-V instruction set manual volume ii: Privileged architecture version 1.10. Technical report, EECS Department, University of California, Berkeley, May 2017
18. Zhang, T., Guan, N., Deng, Q., Yi, W.: Start time configuration for strictly periodic real-time task systems. J. Syst. Archit. **66**(C), 61–68 (2016)