Device Driver and System Call Isolation in Embedded Devices

Maja Malenko
Graz University of Technology
Graz, Austria
malenko@tugraz.at

Marcel Baunach
Graz University of Technology
Graz, Austria
baunach@tugraz.at

***Abstract*—The number of low-end embedded devices in today's Internet of Things and Cyber-Physical Systems is increasing along with their security concerns. Memory isolation mechanisms are often absent, programming flaws lead to malfunctioning applications, which in turn can crush the whole system. A common design approach in these devices is to have applications, operating system components, and device driver libraries reside in a single non-isolated address space, which represents one vast attack surface. Furthermore, with increasing network connectivity and frequent dynamic updates, new or modified applications and services are uploaded, opening space for even more attacks. Isolating the execution of applications in these systems is still a challenge.**

In this work we provide a holistic hardware/software co-designed approach for memory isolation, which prevents corruption of the state of the operating system and applications from a buggy software, including device drivers, interrupt service routines, and misused system calls. We implemented low-cost architectural extensions in a RISC-V-based microcontroller which work together with kernel-based protection concepts. Our evaluation shows that applications as well as the kernel can enjoy the benefits of the proposed memory isolation with minimal impact on performance and an insignificant increase in the area of the MCU.

***Index Terms*—memory isolation, memory protection, device driver isolation, secure system calls, RISC-V**

I. Introduction

Embedded systems are reactive systems with very frequent and security critical interaction with the environment: waiting for sensor inputs, making calculations, and generating outputs. Their dependability is a major issue since failures, as many attacks have shown [10] [12] [24], can have immediate, at times catastrophic impact on the environment.

The software even on small microcontrollers (MCUs) consists of a diversity of concurrent applications, operating system services, and libraries, all residing in a single linear memory address space. When different parties are involved in developing applications and device driver libraries, these systems can easily become very complicated, inconsistent, and even buggy. Without proper memory isolation, they represent one big attack surface.

Even though many low-end MCUs rarely implement any security mechanism, others provide different memory isolation techniques to isolate all or some sensitive applications. Unlike commodity operating systems which usually use virtual address spaces and Memory Management Unit (MMU)-based isolation, low-end systems use some form of a Memory Protection Unit (MPU) [15] [23] [30] [31] to protect different regions in a single address space with lightweight hardware-enforced checks.

Device drivers have historically proven to be one of the most vulnerable part of the OS. Usually they are provided by third-parties, have more bugs than any other code, and threaten OS dependability. Fault isolation in device drivers prevents from global system misbehavior and failure. Thus, one goal of this paper is isolation of the memory accessible to device drivers. Another isolation aspect focuses on protection of security critical kernel data structures accessed by malformed system call arguments.

By co-designing the hardware and the software, the system becomes more efficient as it can easily adapt to new kernel-imposed isolation scenarios which are enforced by hardware extensions. We offer lightweight hardware extensions in a RISC-V implementation [28] [29] and a kernel run-time library in microkernel-based *Smart*OS, which work together to isolate device drivers, Interrupt Service Routines (ISRs), and system calls. These protection mechanism do not impose any run-time performance overhead and are suitable for maintaining the existing real-time properties of the system.

The main contributions of our hardware-enforced isolation approach are the following:

- Creation of a general structure for device drivers according to the description of hardware I/O devices received from AutoIO register design tool [21]. Creation of necessary synchronization and resource protection primitives when loading device drivers.
- Memory isolation of device drivers and ISRs by introducing hardware checks triggered on specific memory addresses.
- Protection of system call kernel entry points which use synchronization and resource management kernel data structures by introducing hardware checks on specific processor instructions.

The paper is structured as follows. First we give an overview of the operating system concepts that are used in the implementation and are relevant for the protection mechanisms we provide (Section II). Next, we give a description of the implementation of the hardware modules which are enforcing the memory isolation (Section III). Then, measurement results are presented (Section IV) and our work is related to similar approaches (Section V). Finally, we draw a conclusion (Section VI).

II. Threat Model

The main security objective of our system is to prevent malicious code execution in one task from affecting other tasks or the OS.

The goal of the attacker is (1) to corrupt the task data from within a device driver or (2) to change important kernel data structures, thus bypassing any security mechanisms.

Since device drivers in our OS are executed in user mode, in the context of the running task, we specifically focus on malicious code execution in device drivers from affecting the data of the running task. Thus, we want to protect against data corruption attacks caused by faulty drivers, not necessarily control-flow hijacking attacks. The system already implements task and kernel isolation which makes it difficult for an attacker who has compromised one task to launch memory corruption attacks to other tasks or to the kernel due to the restricted memory access. On the other hand, we also consider threat models that use system calls as an interface to change security critical kernel data structures by passing badly formed arguments. We assume that a task or device driver will change the data and control flow by maliciously accessing and modifying unintended kernel data structures via system calls.

We consider the kernel to be trusted and securely booted [9] [25]. We also assume that an attacker can successfully install a malicious device driver, including an ISR executed in kernel mode. Also, the attacker is able to install a task which tries to circumvent correct system call execution by passing well crafted malicious arguments. Physical attacks as well as DoS attacks are out of scope.

III. System Overview

In this section we are going to give a short description of the necessary kernel services, data structures, and interfaces to the hardware, which are relevant for the rest of the paper.

*Smart*OS [6] is a microkernel-based operating system which provides only basic functionality in privileged mode, as shown in Fig. 3. All other services and applications run in user mode as tasks or libraries. The main execution unit in *SmartOS* are tasks. They are preemptive, have unique static priorities defined at compile time and active priorities which are dynamically modified by the resource manager and used by the scheduler. Kernel manages tasks by using Task Control Blocks (TCBs) of type *Task_t*. For mutually exclusive access to shared resources, tasks use the resource manager which implements the Priority Inheritance Protocol (PIP) [13]. In this work we are interested in resources for physical devices. The kernel manages resources by using Resource Control Blocks (RCBs) of type *Resource_t*. Synchronization is achieved using events, controlled by using Event Control Blocks (ECBs) of type *Event_t*. The memory is organized by the linker to contain a well structured arrangement for TCBs, RCBs, and ECBs

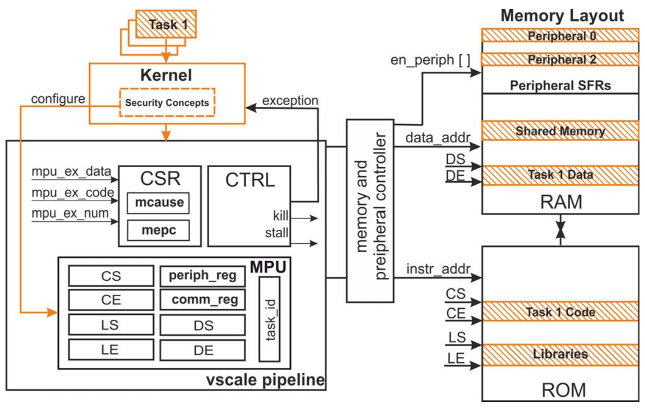


Fig. 1. Overview of the basic memory protection system.

as well as regions for task stacks, task data, and task entry functions.

Memory isolation is an effective technique frequently used in computer systems. The isolation mechanisms presented in this paper are built upon our previous work presented in [18] and [19], in which each task is an isolated module and has a particular memory layout structured in three sections: code, data, and stack, each of which is a contiguous range of memory with different access control permissions. The basic memory protection concept for isolating tasks is extended to prevent them from performing illegal I/O operations and to enable authenticated and protected shared memory for communication. The permissions for accessing on-chip peripherals are granted after executing the resource management system calls *getResource(Resource_t *)* and *setResource(Resource_t *)*. They are stored in a bitmap register which identifies the peripheral ports that a task has access to. If a task has permissions to access a port, a ”1” is stored in the corresponding peripheral index. This concept does not prevent failures in device drivers, ISRs, and incorrect system calls from propagating to the rest of the system. The overview of the system is shown in Fig. 1.

The access control mechanisms that are used in our system are largely OS dependent, since the kernel mechanisms dictate what, when, and how is protected. Hardware extensions are co-designed appropriately and help in enforcing them. In order to be suitable for this kind of protection, the underlying architecture must support at least two different privilege levels as well as privileged instructions (for configuring security critical Control and Status Registers (CSRs)). Our MPU already offers basic W$\oplus$X protection, by allowing tasks only to read and write from/to the data region, and only to execute instructions from the code region.

A. Device Drivers and ISRs

Each peripheral device has a set of memory-mapped registers as an interface that is provided to the device driver software. By writing to and reading from these registers the driver initializes, configures, and communicates with the device. While in commodity operating systems drivers are executed in kernel mode, in our system, like in many bare-metal and microkernel-based systems, task and drivers, can directly access the peripheral devices. In addition to our previous attempt to protect on-chip peripheral devices which are not accessed according to the OS's resource management protocol [19], in this paper we further protect task's data from being modified from malicious drivers.

Device drivers in *Smart*OS are libraries called and executed in the context of the running task. Incorrect or malicious drivers can cause silent memory corruption which often can lead to incorrect results or system crashes difficult to diagnose. While a task is executing it can implicitly access peripheral devices through device drivers, and they in turn can access only the peripheral's registers they are written for. No illegal I/O operation should be performed from the drivers.

Interrupts are used for environment-to-task synchronization. Under *Smart*OS, all enabled hardware interrupts will be processed by a common handler, which depending on the interrupt source, calls specific ISR registered for each source. RISC-V categorizes interrupts into three classes: software, timer, and external interrupts. The external interrupts are routed to the core via a Platform Level Interrupt Controller (PLIC). The cause of the interrupt is stored in predefined CSR which is inspected by the kernel before jumping to the appropriate ISR. ISRs in *Smart*OS are executed in kernel context and have complete access to all kernel code and data structures. Since interrupts break the program flow, they must be fast. That is the reason why in our system they are only setting appropriate events for synchronization with the tasks.

The idea behind the device driver and ISR isolation is to limit their access permissions only to the memory locations dedicated for the peripheral device the driver is written for. That is realized by using a combination of well structured Device Driver Blocks (DDBs) of type *Driver_t*, per-driver isolation policies, and run-time granting of peripheral device memory. The kernel creates the isolation policy at the moment when the device drivers are first loaded into memory, while its enforcement is done at run-time by a hardware module which checks for correct accesses to peripheral registers and other allowed data regions. All peripherals are abstracted as resources and all interrupt requests from a peripheral are coupled with events. On system start up, before the scheduler starts running, we are loading all low-level drivers from a special driver constructors. Fig. 2 shows how device drivers communicate with hardware peripherals and how events and resources are propagated.

B. Resources

To maintain temporal, exclusive, and controlled access to one or more shared peripheral devices, *Smart*OS uses a resource management concept. Each peripheral in software is represented as a resource and each resource is managed with its individual RCB. Whenever a low-level driver is loaded, the peripheral is initialized, and all the necessary RCBs are created. When a resource is assigned to a task, the task becomes its owner, and has complete and exclusive access to

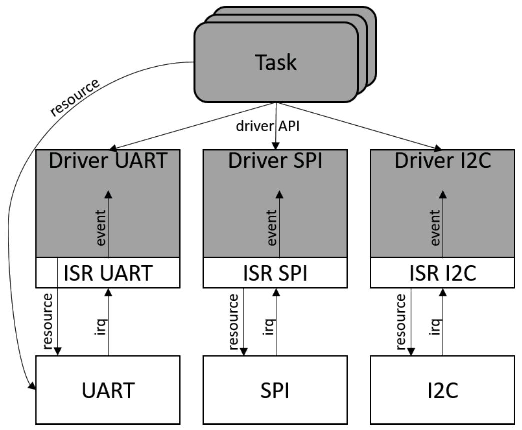


Fig. 2. Relationship between tasks, device drivers, and peripheral blocks.

the registers that belong to the peripheral device represented by that resource. Resource owners remain preemptive, while resources are non-preemptive. They are managed according to the PIP which dynamically modifies the priorities of tasks that cause a blocking condition. A task always inherits the highest priority of all tasks it blocks due to a resource allocation. After it releases a resource, its priority is re-adjusted.

The resource management concept itself is not enough to physically protect the address range of a peripheral device. It works well only if every task or driver in the system obeys to it. The kernel has no means to prevent illegal use of any peripheral component by a task which does not hold the corresponding resource. Hence, we are using the approach described in [19] which introduces hardware-enforced protection in the resource management concept and works well even with partitioned (de-multiplexed) peripherals devices [18]. The peripheral (IO) operations are always handled by the MPU, which locks the access to every on-chip peripheral available in the system, unless it is explicitly requested by the task and granted by the OS. Any illegal access results in an exception. At the application side, resources are granted/released and protected by using *getResource(Resource_t *)* and *releaseResource(Resource_t *)* functions which are mapped into system calls.

C. Events

*Smart*OS uses events to synchronize tasks with each other and with the interrupt handlers which notify them when certain interrupts have happened. At the application side, waiting for an event is done by using *waitEvent(Event_t *)* function, while setting an event is done by using *setEvent(Event_t *)* function, both mapped into system calls.

In this work, events correspond to interrupts and are created and provided by the driver. The kernel makes sure there are as many different events as the number of interrupts the device can emit.

D. System Calls

System calls are an interface between tasks and the kernel. *Smart*OS has limited number of system calls which provide access to several important kernel data structures, including system calls for resource management which modify RCBs, system calls for task synchronization and notification which modify ECBs, and system calls for communication which modify shared memory addresses. Changes to these kernel data structures is commonly initiated by application code by using the system call API functions and is always performed by kernel functions. The system call API functions internally call a system call wrapper (*__syscall_<name>*) which switches to kernel mode and eventually calls the actual system call (*syscall_<name>*). When finished, the scheduler selects the next task for running.

When accessing kernel data structures via system calls, tasks are passing a pointer to the structure as an argument. By making system calls with bad casted address arguments for kernel control blocks (events and resources are considered in this paper), a task or a device driver might maliciously modify other critical memory location and change the state of other tasks, cause inconsistency in the execution, or even crush the system, as shown in Listing1.

Listing 1. Example of modifying unintended control structure

```
Event_t *ev;
Resource_t *res;
setEvent((Event_t *)res);
```

To avoid that, we introduce system call argument checks as described in Section IV-B, which allow easy hardware monitoring of the passed arguments and aim to limit the damage that might be caused by passing wrong arguments.

IV. Design and Implementation

In order to protect the system from malfunctioning device drivers and system call arguments, we developed Protection Module (PM) placed inside the main core of the *vscale*[1] RISC-V implementation. The PM is programmed by the kernel, but performs the checks on its own. In this way we not only enable secure device drivers execution in user mode without additional context switches, but we also minimize the runtime overhead of the system.

In this section we are going to describe the implementation of two types of hardware components, the Device Driver Isolation Module (DDIM) which is activated whenever a task enters into a device driver code memory region, and the System Call Tracing Module (SCTM) which is invoked when an *ecall* machine instruction takes place. When data memory access, control-flow, or an *ecall* instructions are executed by the processor, the hardware modules inspect the data memory address, the program counter, or the argument value, respectively, and make decisions as described in the following sections. These checks take place in the decode/execute stage of the three-stage *vscale* pipeline. The overview of the system

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

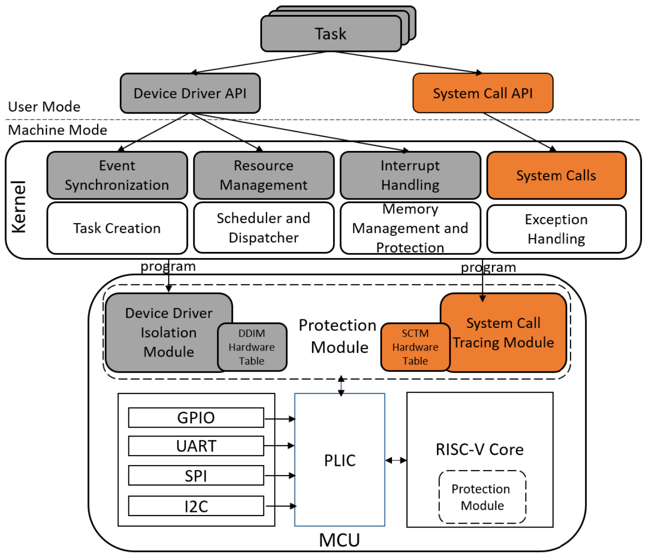


Fig. 3. System Overview.

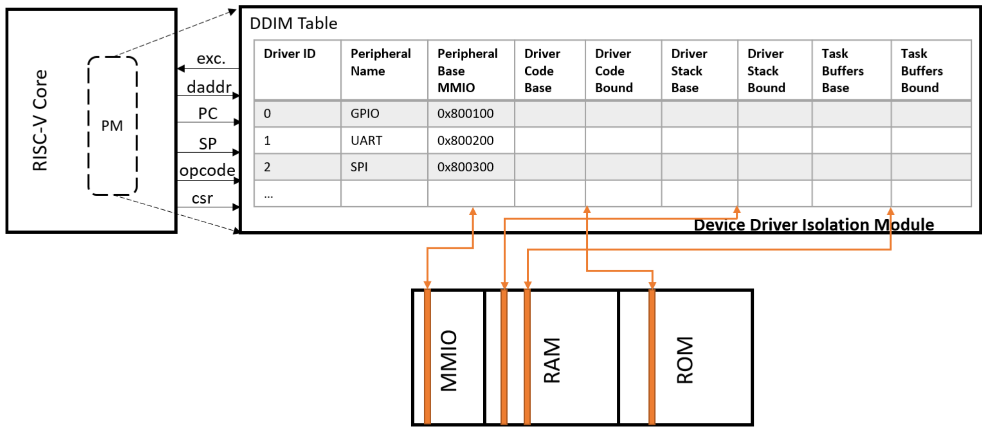
is shown in Fig. 3. If an irregularity is detected, an exception is raised and the execution jumps to predefined kernel entry point.

A. Device Driver Isolation Module

Every peripheral device is represented in software by a data structure containing the registers that belong to it. We are using a register design tool, called *AutoIO* [21], which generates hardware and firmware header files for all hardware components in an MCU, including device drivers. The base addresses of peripheral devices are static and are fixed during synthesis of the system. If dynamic address are to be used, the device driver can determine the current configuration and make adjustment to the base address of the peripheral. Every peripheral block has some spare memory space within its address range to add new registers in the future. Peripherals in our MCU implementation are placed along 256-byte boundaries, meaning each peripheral device has room for 64 32-bit register addresses.

Each peripheral has exactly one low-level device driver associated with it. During the boot process all device driver constructors located in well defined memory regions are executed and the corresponding low-level device drivers are loaded. The structure of an example driver control block and the initializing constructor is shown in Listing2. Each driver initializes its device and sets the corresponding address of the ISR in a kernel interrupt handler table. It also creates all the necessary resources (RCBs) requested by tasks or device drivers and directly involved in the physical protection of the device as well as all the events (ECBs) corresponding to the number and type of interrupts the device emits.

We extended the hardware with a Device Driver Isolation Module (DDIM). Whenever a new low-level device driver is



Driver ID	Peripheral Name	Peripheral Base MMIO	Driver Code Base	Driver Code Bound	Driver Stack Base	Driver Stack Bound	Task Buffers Base	Task Buffers Bound
0	GPIO	0x800100						
1	UART	0x800200						
2	SPI	0x800300						
...								

Fig. 4. Device Driver Isolation Module

loaded, the hardware table of this module is programmed by the kernel with the relevant information necessary for isolated execution of the device driver as shown in Fig. 4. The DDIM keeps track of the memory regions which are allowed to be accessed by the device drivers, including memory-mapped on-chip registers, events, resources, and stack memory. It provides execution aware memory protection, as described in [15] by binding data to code region.

Listing 2. Example of a driver constructor for a simplified SPI peripheral

```
typedef struct {            typedef struct {
   uint32_t SPI_SR;            uint32_t driver_start;
   uint32_t SPI_CR;            uint32_t driver_end;
   uint32_t SPI_DR;            uint32_t isr;
}SPI_Device_Reg_t;             Resource_t *resources;
                               uint8_t numResrources;
                               Event_t *events;
                               uint8_t numEvents;
                            }Driver_t;

SPI_Device_Reg_t *SPI_reg = SPI1_BASE_ADDR;

OS_CONSTRUCTOR(ctor_SPI, 1) {
   Driver_t SPI_Driver;
   Result_t result = create_DDB(&SPI_Driver, __driver_SPI_S
        , __driver_SPI_E, spi_ISR);
   SPI_Init(SPI_reg);
   return result;
}
```

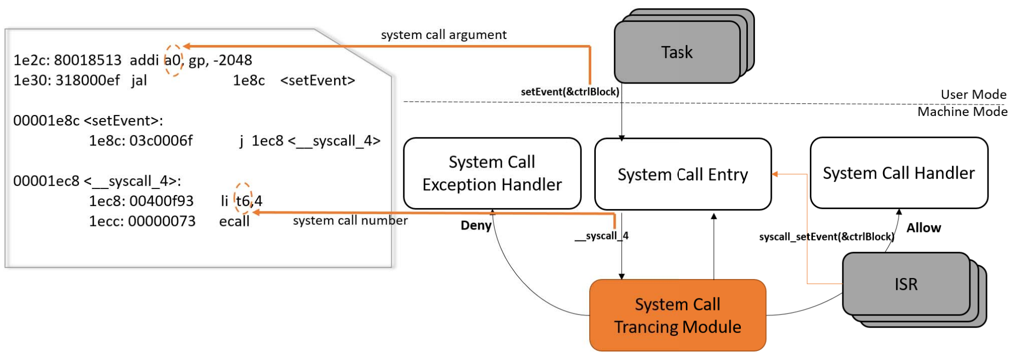
The DDIM detects when the code execution is within a device driver code region by inspecting the program counter value and comparing it to the address ranges of the available device driver code regions stored in the hardware table. It easily finds the device that is being accessed and checks whether the data addresses binded to that code region are allowed. Specifically it checks three data regions: (1) memory-mapped region of the corresponding peripheral, (2) stack region which is opened from the moment the task enters in a device driver code, i.e., isolated stack addresses from the running task stack, and (3) data address for the buffers with which the driver communicates with the task. These buffers should be shared with the DDIM upon getting the resource for the corresponding peripheral and must be consecutively aligned in memory.

ISRs are part of the device driver code. They execute in privilege mode and have unlimited access to the whole memory. Interrupts are routed to the core from the PLIC. Interrupt bits are used to determine which event has occured. The ECBs created when the driver is loaded correspond to the event bits usually located in an Interrupt Pending register. In order to prevent them from accessing and modifying crucial kernel information, we are isolating the memory portions they are allowed to access in the same manner as for the device drivers. From within the ISR only *setEvent(Event_t *)* is allowed to be called, and the hardware opens only the memory-mapped registers of the peripheral which issued the interrupt request. The DDIM has knowledge of the device which issued the interrupt and accordingly allows only the events mapped to that interrupt to be set. The SCTM described in IV-B has knowledge of the context from which the *setEvent(Event_t *)* was called (task or interrupt), and makes the appropriate checks on the ECB address argument.

If the MCU contains more than one instantiation of the same peripheral device/block, multiple tasks can concurrently operate on different instantiation. Even though the same device driver code can be used to access several instantiations, each device gets its own set of resources and events. If the platform supports extensible on-chip peripherals [26], by means of reconfigurable hardware, existing peripherals might be updated or new peripherals might get installed. With the device driver approach it is possible to enable protection for new or updated driver software.

B. System Call Tracing Module

When system calls are issued, we monitor the system call interface to the hardware and deduct important information about the underlying data being accessed. The system calls in which we are interested in this work are shown in Table I, along with their corresponding decoding numbers. The System Call Tracing Module (SCTM) binds each type of system call only with the kernel data structures it is allowed to access (e.g., *syscall_setEvent(Event_t*)* is allowed to access only



```
1e2c: 80018513  addi a0, gp, -2048
1e30: 318000ef  jal          1e8c   <setEvent>

00001e8c <setEvent>:
          1e8c: 03c0006f        j  1ec8 <__syscall_4>

00001ec8 <__syscall_4>:
          1ec8: 00400f93     li  t6,4
          1ecc: 00000073     ecall
```

Fig. 5. System Call Tracing Module.

TABLE I
SYSTEM CALL MAPPING TABLE

System Call	System Call #	Type of Control Block
getResource	1	RCB (1)
setResource	2	RCB (1)
getEvent	3	ECB (2)
setEvent	4	ECB (2)

ECBs). The SCTM monitors the addresses of ECBs and RCBs passed as arguments to capture incorrect references. It contains hardware table in which each type of system call is binded to the allowed addresses of control blocks it can access. The kernel data structures and their corresponding locations in memory are exposed to the SCTM by the kernel whenever a new kernel data structure which needs to be monitored is created.

The system call mechanism uses a single kernel entry point for every system call. Depending on the system call number, the execution is passed to the appropriate system call handler. The calling convention in our system puts the function arguments in registers *a0* and *a1*, while the number of the system call is stored in register *t6*. When a task wants a system service from the kernel it first calls a system call API function which then calls a wrapper function for the system call (Fig. 5). The wrapper function saves the number of the system call in a register *t6* and executes an *ecall* instruction. Authorization of the system call is done by the SCTM as soon as the pipeline decodes an *ecall* instruction. SCTM knows which system call is invoked, it knows the address of the kernel data structure passed as an argument, and it inspects the validity of that address according to the type of system call. The process is shown in Fig. 5. By knowing the size of each data structure, the hardware can easily check whether a well aligned address is passed as an argument.

SCTM can distinguish between several contexts in which a system call is executed: device driver, task, and ISR context. When the system call is called from within the task code region, the SCTM does not authorize resources or events individually, but it only checks whether the kernel data structures belong to that type of system call. When an ISR or a device driver is executing, kernel data structures are explicitly and individually checked and authorized. When calling the *setEvent(Event_t *)* system call, the SCTM checks if the context from which it was called is an ISR or device driver, and immediately checks the ECB addresses that belong to that context.

So far the SCTM expects all related control blocks to appear in consecutive memory region. When they are to be randomly put in memory as a future goal we are going to extend this mechanism by attaching and inspecting metadata to the control blocks and directly checking it in hardware.

V. RESULTS

In order to evaluate the correctness and the hardware footprint of our hardware-enforced fault isolation concept, we implemented a system prototype on the Basys3 Artix-7 evaluation board from Digilent[2]. The prototype consists of a *vscale*-based MCU (with the additional hardware logic for the described memory access control) with several on-chip memory-mapped peripherals (e.g., UART, GPIO, SPI), on which *Smart*OS is running.

To evaluate the used FPGA resources of our tracing modules we measured the hardware footprint of the baseline system and the increase when the hardware components for protection are enabled. We considered only three on-chip peripheral devices as well as 4 different system calls expecting two different types of control blocks as arguments. The overall utilization of lookup tables for logic (LUTs) and flip-flops (FFs) as reported after synthesis is shown in Table II.

Regarding execution overhead, both the Device Driver Isolation Module (DDIM) and System Call Tracing Module (SCTM) perform the corresponding checks in one clock cycle, adding no execution overhead. The DDIM just uses the DDIM Hardware Table to check whether the address of the PC counter belongs to a device driver, and immediately checks whether a data access is performed in the memory area belonging to that driver. The SCTM is activated on an *ecall*

[2]https://reference.digilentinc.com/reference/programmable-logic/basys-3/start

TABLE II
SYNTHESIS RESULTS OF OUR HARDWARE EXTENSIONS.

Category	Components	LUTs (Util%)	FFs (Util%)
Baseline System	*vscale*	2387 (29.84%)	1005 (6.28%)
Extensions	Basic MPU and Resource Protection	147 (1.85%)	320 (2%)
	Device Management Module	98 (1.25%)	150 (0.93%)
	System Call Tracing Monitor	80 (1%)	133 (0.83%)

instruction and looks up into the SCTM Hardware Table for valid address arguments corresponding to the type of the system call. The setup of driver regions as well as system call data structures is done once for each driver and newly created control block, so its execution time is not counted as an overhead.

The amount of newly introduced code is also negligible. When a new driver is loaded the kernel configures the hardware table by executing two store instructions. The stack space the DDIM allows to be accessed from within a driver is directly tracked by the DDIM, whenever a new entry into a driver region is detected. Regarding the data buffers for communication between a task and the driver, the task is obliged to share its own data address space when requesting the peripheral. One additional store instruction is executed when a new control block is loaded, presuming they are consecutive in memory.

VI. Related Work

In contrast to our architecture, some security architectures like [3] [11] [23] have domain separation between secure and non-secure domains, but usually do not consider multitasking protection within domains. Our architecture does not make that domain distinction, but tries to isolate every task from other tasks as well as from device drivers, ISRs, and incorrectly formed system call arguments.

In x86-based systems [7], separate I/O instructions can be executed only from kernel space. This necessitates the use of kernel space drivers to access peripherals. In this mainly general purpose systems many malicious activities often use system calls to execute privileged operations on system resources. In ARM-based systems, like in our system, peripherals can be directly accessed from a user application. In ARM TrustZone [3], peripherals as well as applications can be separated into secure and non-secure domains.

Our system call monitoring approach is different from the interposition-based sandboxing and intrusion detection solutions [16] that rely on system call interposition. We are not tracing the system calls, since we are not interested in determining the behavior of the application (all tasks are allowed to call any system call).

If we compare our system to other microkernel-based systems [2] [4] [5], we can find many similarities in basic protection as well as managing resources and synchronization. They all use an MPU with fixed number of protected regions. In particular, in FreeRTOS-MPU [1] tasks can access only their stack and up to three user defined regions for communication with other tasks. By using the solution from our previous work we allow more fine-grained protected communicaton. In order to be protected from other task, a task must be started in privileged mode. If it drops the mode, it can not elevate it again during its execution. FreeRTOS also distinguishes between system and standard peripherals. System peripherals can only be accessed from privileged mode, while standard peripherals (e.g., UART, SPI, etc.) are accessible from user mode, but must be explicitly protected using a user definable MPU region. Resources in FreeRTOS are protected by means of mutexes. It is the job of the application developer to take care that a mutex is not accessed by more than one task simultaneously, as simultaneous access can corrupt the resource. Neither hardware nor software enforcement of protection is present. When tasks make calls that internally influence and change kernel data structures, a handle to those structures is passed, but no checks similar to our SCTM are executed. In *embOS* [27] a device driver consists of two parts, an unprivileged and a privileged part. The applications do not have direct access to peripherals and they must call driver functions in the unprivileged part. The actual peripheral access is performed in the privilege part thus increasing the runtime overhead of every peripheral access.

There is a considerable amount of research in hardware-assisted kernel integrity monitoring [14] [17], where several regions of immutable and mutable parts of the kernel memory are monitored during runtime using whitelisting. Our monitoring of critical kernel data structures used for synchronization and resource management are considered as part of kernel integrity monitoring. Also, the whitelisting approach we use for device driver execution monitoring, ensures task integrity instead.

In contrast to our microkernel-based system, in [20] the authors are tackling similar concept but trying to isolate certain kernel memory regions in a monolithic system from being accessed by malicious kernel extensions. Many works are also tackling different approaches in device driver isolation, primarily in commodity operating systems, by moving them in user mode, running them in separate virtual machines [8] [22] [32].

VII. Conclusion

In this paper, we have proposed a hardware-enforced approach for providing run time protection by memory isolation of device drivers and system call entry points in embedded systems. Two new hardware monitoring modules are implemented, a Device Management Module, which limits the access scope of device drivers and ISRs, and System Call Tracing Module, which intercepts the arguments of crucial system calls and checks whether a tampering of some kernel data structures takes place. The hardware extensions provide increased security as defined by kernel specifications in an efficient manner, regarding hardware usage and performance.

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] Freertos-mpu, https://www.freertos.org/freertos-mpu-memory-protection-unit.html.

[2] mbedos, https://os.mbed.com/.

[3] ARM Limited. *ARM Security Technology - Building a Secure System using TrustZone Technology*, 2009.

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

[5] R. Barry. *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.

[6] Marcel Baunach. Towards collaborative resource sharing under real-time conditions in multitasking and multicore environments. In *ETFA*, pages 1–9. IEEE, 2012.

[7] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[8] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[9] George Bricker. Unified extensible firmware interface (uefi) and secure boot: Promise and pitfalls. *J. Comput. Sci. Coll.*, 29(1):60–63, October 2013.

[10] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, San Diego, CA, 2014. USENIX Association.

[11] R. de Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede. Secure interrupts on low-end microcontrollers. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 147–152, June 2014.

[12] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 378–397, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[13] John Goodenough and L Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority ada tasks. *ACM SIGAda Ada Letters*, VIII:20–31, 11 1988.

[14] D. Hwang, M. Yang, S. Jeon, Y. Lee, D. Kwon, and Y. Paek. Riskim: Toward complete kernel protection with hardware support. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 740–745, March 2019.

[15] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM.

[16] Samuel Laurén, Sampsa Rauti, and Ville Leppänen. A survey on application sandboxing techniques. In *Proceedings of the 18th International Conference on Computer Systems and Technologies*, CompSysTech'17, pages 141–148, New York, NY, USA, 2017. ACM.

[17] Hojoon Lee, Minsu Kim, Yunheung Paek, and Brent Byunghoon Kang. A dynamic per-context verification of kernel address integrity from external monitors. *Computers and Security*, 77:824 – 837, 2018.

[18] Maja Malenko and Marcel Baunach. Hardware/software co-designed peripheral protection in embedded devices. In *ICPS*, 2019.

[19] Maja Malenko and Marcel Baunach. Hardware/software co-designed security extensions for embedded devices. In *Architecture of Computing Systems - ARCS 2019 - 32nd International Conference, Copenhagen, Denmark, May 20-23, 2019, Proceedings*, pages 3–14, 2019.

[20] Valentin J.M. Manès, Daehee Jang, Chanho Ryu, and Brent Byunghoon Kang. Domain isolated kernel: A lightweight sandbox for untrusted kernel extensions. *Computers and Security*, 74:130 – 143, 2018.

[21] Paul Nagele. Design and Implementation of an I/O Specification Tool for MCU Architectures. Master's thesis, Graz University of Technology, 2017.

[22] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 116–132, New York, NY, USA, 2013. ACM.

[23] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 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)*, pages 479–498, Washington, D.C., 2013. USENIX.

[24] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin OFlynn. Iot goes nuclear: Creating a zigbee chain reaction. *IEEE Security and Privacy*, 16:54–62, 01 2018.

[25] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[26] B. Sukhwani, A. Forin, and R. N. Pittman. Extensible on-chip peripherals. In *2008 Symposium on Application Specific Processors*, pages 55–62, June 2008.

[27] S. Tan and T. Anh. Real-time operating systems for small microcontrollers. *IEEE Micro*, 29(05):30–45, sep 2009.

[28] Andrew Waterman, Lee, and Krste Asanović. The risc-v instruction set manual volume i: User-level isa version 2.2. Technical report, EECS Department, University of California, Berkeley, May 2017.

[29] Andrew Waterman, Lee, and Krste Asanović. The risc-v instruction set manual volume ii: Privileged architecture version 1.10. Technical report, EECS Department, University of California, Berkeley, May 2017.

[30] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *Network and Distributed System Security Symposium (NDSS) 2019*, 2 2019.

[31] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[32] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *2014 IEEE Symposium on Security and Privacy*, pages 308–323, May 2014.