Enabling Trusted Scheduling in Embedded Systems

Ramya Jayaram Masti[†], Claudio Marforio[†], Aanjhan Ranganathan[†],
Aurélien Francillon[‡], Srdjan Capkun[†]
[†]Institute of Information Security, ETH Zurich, Switzerland
[‡]Eurecom, Sophia-Antipolis, France
[†]{rmasti, maclaudi, raanjhan, capkuns}@inf.ethz.ch
[‡]aurelien.francillon@eurecom.fr

ABSTRACT

The growing complexity and increased networking of security and safety-critical systems expose them to the risk of adversarial compromise through remote attacks. These attacks can result in full system compromise, but often the attacker gains control only over some system components (e.g., a peripheral) and over some applications running on the system. We consider the latter scenario and focus on enabling on-schedule execution of critical applications that are running on a partially compromised system — we call this *trusted scheduling*. We identify the essential properties needed for the realization of a trusted scheduling system and we design an embedded system that achieves these properties. We show that our system protects not only against misbehaving applications but also against attacks by compromised peripherals. We evaluate the feasibility and performance of our system through a prototype implementation based on the AVR ATmega103 microcontroller.

1. INTRODUCTION

Today, security- and safety-critical systems are being increasingly networked to facilitate their remote configuration, control and monitoring. As a result, they face an increased risk of adversarial compromise and therefore have to be designed to meet their real-time constraints even if they are partially compromised. More specifically, it is necessary to architect them such that they can guarantee the execution of certain critical functionality despite the presence of other misbehaving system components (e.g., compromised applications, peripherals). We refer to this property of preventing applications and components under the attacker's control from changing the execution times of other applications as *trusted scheduling*. Recent examples of compromised embedded systems [10], control systems [16] and peripherals [15] show that this is an emerging problem.

Most safety-critical systems include a real-time operating system (RTOS) [5, 6] or similar system management software (e.g., microkernel [18]) whose primary goal is to ensure that their real-time constraints are met. More recently, these RTOS also include mechanisms to contain the effects of other misbehaving software components/applications. However, RTOS do not address threats by untrusted peripherals (e.g., an RTOS cannot prevent a compromised peripheral from making the peripheral bus unusable by not adhering to the bus protocol). Furthermore, their complexity makes them prone to vulnerabilities that can be exploited to force the system to deviate from its expected behavior [4].

In this work, we address the problem of enabling trusted scheduling in the context of security- and safety-critical embedded systems. These are specialized devices that typically include a CPU, memory and some peripherals connected to the CPU via the peripheral bus. They usually run a fixed set of applications whose resource requirements are well-known in advance. We first identify three essential components of a trusted scheduling architecture, namely, secure scheduling, secure resource allocation and application state protection. This is in contrast to conventional scheduling that only focuses on the CPU allocation.

Second, we describe an embedded system architecture that achieves trusted scheduling and analyze its security. Our architecture includes five main hardware components, namely, a scheduler which decides the order in which applications execute, time-slice and atomicity monitors that ensure CPU availability, an application-aware memory protection unit which mediates memory access and a peripheral bus manager which controls access to the peripheral bus. These components, together with a thin layer of software, ensure that misbehaving applications and peripherals cannot influence the system's expectation for other applications. We show that our architecture provides strong guarantees against remote attacks that exploit software vulnerabilities which we believe is crucial for today's safety-critical systems. We then evaluate the feasibility of realizing such an architecture through a prototype implementation based on the AVR ATmega103. Finally, we discuss how the design of system components (e.g., bus, peripherals) can affect the feasibility of achieving trusted scheduling on a particular architecture.

The rest of the paper is organized as follows. In Section 2 we discuss the problem of enabling trusted scheduling and identify the functions needed to achieve it in a system. In Section 3, we describe a trusted scheduling architecture for embedded systems and analyze its security. In Section 4, we discuss several practical security issues involved in realizing a trusted scheduling architecture. We present preliminary performance considerations in Section 4.2. Finally, we discuss related work in Section 5 and conclude in Section 6.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

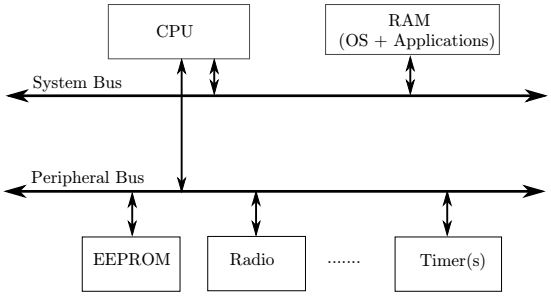

Figure 1: Most embedded systems consist of a *CPU*, *memory* (for code and data) and a set of *peripherals* that interact over a *system bus* and a *peripheral bus*. Typically, the peripherals are connected on a separate bus for efficiency reasons.

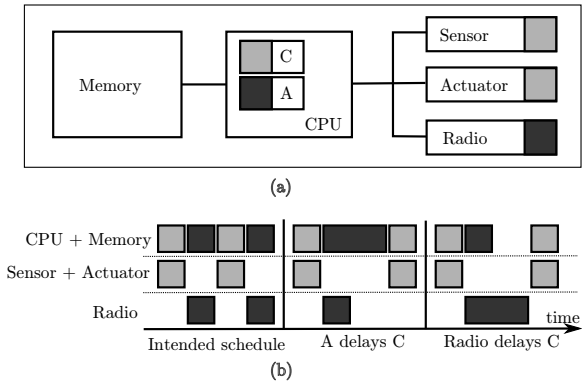

Figure 2: The execution schedule of a critical application *C* can be disrupted by a misbehaving (non-critical) application *A* that does not release the CPU on-time or exhausts the memory or by the *radio* peripheral that does not release the bus on-time.

2. TRUSTED SCHEDULING

In this section, we define trusted scheduling and identify the requirements that a system must satisfy in order to enable trusted scheduling.

2.1 Problem Statement

Trusted scheduling is the property of ensuring adherence to the intended execution schedule in adversarial settings.

To illustrate the problem of enabling trusted scheduling, we consider a simple embedded system $\mathcal{S}$ (Figure 1) that consists of a CPU, memory unit(s) and peripherals that are connected over one or more buses. Most embedded systems have two such buses: a system bus that is used to connect main components (e.g., CPU) to the memory unit(s) and a slower peripheral bus (e.g., SPI [9], I^2C [12]) that is used to connect the CPU to the peripherals (e.g., EEPROM, real-time clock, radio and other I/O interfaces). The system hosts a number of applications that are entirely independent and self-contained.

We consider an attacker $\mathcal{M}$ who controls a subset of applications and system components and is interested in interfering with the execution schedule of other (critical) applications that it does not control. For example, $\mathcal{M}$ could compromise an application and use it to gain control over the network card on the peripheral bus. We assume that $\mathcal{M}$ does not have physical access to the system and hence cannot launch physical attacks. We further assume that the attacker cannot influence any external inputs that affect the system's execution schedule.

This model corresponds to systems where critical and non-critical applications/peripherals co-exist. For example, a system can consist of a critical control (sensing/actuating) application and a non-critical communication application used for the sole purpose of reporting (Figure 2(a)). In this example system, if the radio peripheral or status reporting application is compromised, they could attempt to influence the execution schedule of the critical control application; this is illustrated in Figure 2(b). While conventional scheduling (or CPU scheduling) suffices to guarantee adherence to the intended schedule as long as all applications and peripherals are benign, it alone cannot provide similar guarantees in the presence of compromised applications and peripherals. Hence, in this work we focus on a broader notion of scheduling in adversarial contexts that we call trusted scheduling and define below.

We say that *a system enforces trusted scheduling if it prevents (possibly colluding) applications and components under the attacker's control from changing the execution times of other applications such that they do not correspond to the intended schedule.* In this work, we primarily focus on attacks that aim to delay or entirely prevent the execution of applications. We further assume that the applications do not fully depend on attacker-controlled applications or components for their execution; otherwise, little can be done to ensure trusted scheduling. Similarly, we also assume that the data influencing the execution schedule of applications either directly (e.g., as an input to the scheduler) or indirectly (e.g., persistent system data) cannot be modified by an attacker.

The problem that we want to solve in this work is that of designing an embedded system that enforces trusted scheduling, assuming that the attacker controls a subset of applications and system components. Existing real-time systems include software [6, 5, 18] and hardware [13, 19] to provide resilience against compromised applications, but do not consider misbehaving system components (peripherals).

2.2 Realizing Trusted Scheduling

Although systems are diverse and the application scenarios in which they are used largely differ, there are still some common functionalities that all systems must realize to support trusted scheduling which we discuss below.

To support trusted scheduling, a system should implement a robust scheduler, protect system resources and protect the applications. More precisely, the system must be designed such that the attacker (i) cannot modify the execution schedule of applications (ii) cannot interfere with the allocation of resources to the applications and (iii) cannot modify the state of applications (code and data).

This effectively implies that, a trusted scheduling system should implement a secure scheduler that schedules the execution of applications and enforces adherence to this schedule. Furthermore, this system should securely isolate appli-

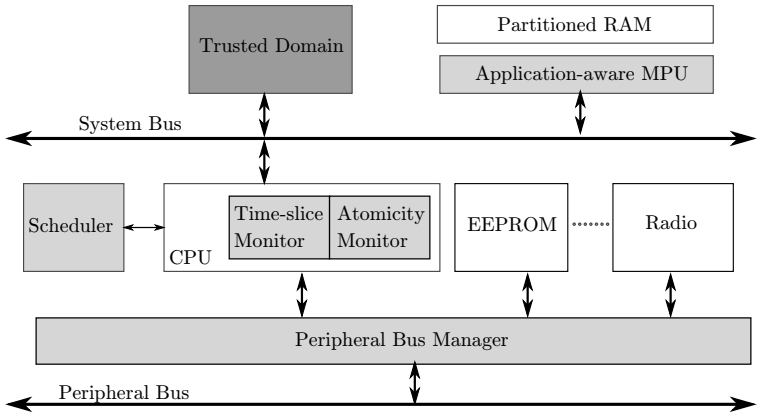

Figure 3: Our trusted scheduling architecture uses a thin layer of software (trusted domain) to initialize and configure its hardware components. The scheduler informs the trusted domain when a new application has to be executed. The trusted domain saves the current system state and transfers control to the new application. The CPU time-slice monitor and the atomicity monitor together guarantee that the trusted domain can regain control of the CPU when needed. Each application is allocated its own dedicated memory partition and application boundaries are enforced using an application-aware memory protection unit (MPU). Finally, the peripheral bus manager controls access to peripherals and also prevents misbehaving peripherals from denying applications access to the bus.

cations such that their code and data are protected, preventing the attacker from modifying applications or preventing their correct execution by modifying their data. Finally, the system should be able to securely multiplex shared system resources and ensure that applications are able to obtain all resources they need on-time for their correct execution. Ensuring the correct allocation of resources prevents internal Denial-of-Service (iDoS) attacks in which an attacker holds on to a system resource (e.g., a bus or a peripheral) required by another application or exhausts system resources to prevent other applications from running (e.g., dynamically allocating memory until it runs out).

In the next section, we describe our design of an embedded system that follows the above reasoning and supports trusted scheduling.

3. TRUSTED SCHEDULING SYSTEM

Our trusted scheduling system is designed for real-time systems with a pre-defined set of applications. It is tailored for embedded devices that are used in security- and safety-critical applications that have a well-defined and focused purpose. The system is initialized and configured by a trusted administrator and is not expected to be modified by the user during use.

3.1 System Overview

Our system is shown in Figure 3. It includes standard embedded system components (shown in white): CPU, RAM, system bus, peripheral bus, peripherals and trusted scheduling extensions (shown in gray): trusted domain, scheduler, application-aware MPU, time-slice and atomicity monitors and a peripheral bus manager.

The scheduler manages the execution of applications and informs the trusted domain when a new application must be executed. The scheduler triggers execution of applications according to its pre-determined schedule or in response to external events. The scheduler is therefore configured with the scheduling policy it must enforce and is aware of the applications that are running on the system. When a new application is scheduled to be executed, the trusted domain saves the current state of the system and transfers control to the new application. Before the actual execution of the application itself, the trusted domain configures and activates the desired CPU time-slice monitor (e.g., a timer interrupt) and the atomicity monitor that transfer control back to it from the application. The CPU time-slice monitor and the atomicity monitor together guarantee that the trusted domain can regain (with minimal latency) control of the CPU when needed. The scheduler is isolated from the applications and other system components; it acts directly on the CPU and thus can always stop and start the execution of the applications. In order to ensure application isolation, each application running on the system is allocated its own dedicated memory partition and application boundaries are enforced using an application-aware memory protection unit (MPU). Finally, to ensure that bus access to the peripherals is securely mediated, we introduce a peripheral bus manager. This peripheral bus manager controls access to the peripheral bus from the various peripherals and prevents misbehaving peripherals from denying the CPU (the running applications) and other benign peripherals access to the bus. It also ensures that each application can only access the set of peripherals to which it has been granted access by the trusted domain.

Every application starts executing at its first instruction and continues until it terminates, violates a security policy or is preempted. A security exception is raised if an application tries to access or modify code or data that does not belong to it. A security exception is also raised if an application tries to execute an atomic section that is larger than the preset limit or if its allocated CPU time-slice expires. Additionally, the peripheral bus manager also raises a secu-

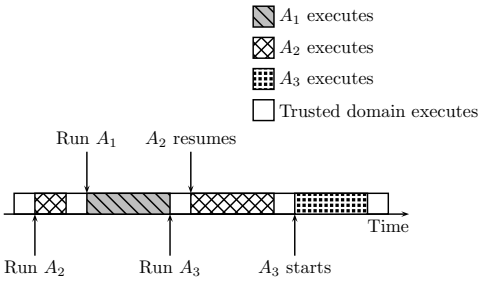

Figure 4: The trusted domain must save and restore state during a context switch to prevent applications from accessing and/or modifying each others state.

rity exception if any application tries to access a resource to which it has not been granted access or if the length of any bus transaction exceeds a pre-determined upper bound. Alternatively, the upper bound on the length of bus transactions can also be enforced by the atomicity monitor. When an application is preempted or forcibly terminated due to a security exception, control is transferred back to the trusted domain (via the hardware scheduler) which resumes execution of pending applications.

In terms of the system initialization, we assume that a trusted administrator supplies system initialization values to the trusted domain. When the system is powered-on, the trusted domain executes first and performs system initialization based on the inputs provided by the administrator. These inputs may include (but are not limited to) the number of applications, the peripherals they use, their memory layout, execution schedule, preemption policies (like the maximum CPU time-slice for each application, etc.). The trusted domain configures the scheduler for periodic and/or event-driven execution of applications. Furthermore, it also configures the resource allocator's CPU preemption policies and the application-aware MPU to create dedicated program and data partitions for each application. Finally, the trusted domain initializes the program memory of individual applications and configures the peripheral bus manager with information regarding the peripheral access requirements of the applications.

3.2 System Components

We now describe selected hardware and software components of the trusted computing base (TCB) in more detail.

Trusted Domain

The trusted domain is the only software component of the TCB and resides in ROM. The trusted domain is responsible for initializing the hardware scheduler, components that allocate system resources (described below), handling context switches between applications and the actual transfer of control from the scheduler to the individual applications (Figure 4). We chose this approach in order to prevent malicious applications from accessing or modifying the state of their predecessors which could result in a violation of trusted scheduling. The trusted domain uses a dedicated data partition in RAM that is not accessible by any other application or system component.

Scheduler

Our architecture includes a hardware-scheduler that uses information about external events and internal logic to decide the order in which applications execute. Although using a hardware scheduler limits flexibility (adding, removing applications, changing algorithms, etc.), we believe that this is an acceptable trade-off for security in real-time systems, as they are mostly single-purpose and run applications that do not change frequently (e.g., due to safety compliance issues). The scheduler transfers control to the trusted domain that in turn transfers control to the actual application.

Our scheduler implements *preemptive scheduling*. Although co-operative scheduling, where applications are allowed to execute to completion or co-operatively release the CPU, may be simpler and more efficient than preemptive scheduling, relying upon applications to release the CPU punctually can be dangerous in adversarial settings. This is because malicious applications may hold onto the CPU and hence delay (or in the worst case prevent) the execution of other applications. Hence, in order to achieve trusted scheduling, one must use preemptive scheduling schemes where the execution of any application can be interrupted at any point of execution except for those that the application explicitly labels as *atomic*.

Resource Allocation Components

The resource allocator must ensure that applications have access to at least three system resources while executing: CPU, memory and the bus(es).

CPU Availability

Atomic sections are required to ensure the correctness of certain operations (e.g., updating the stack pointer (SP)). The sequence of instructions in an atomic section (*(i)* decrementing the existing SP value and *(ii)* re-storing the new value in SP) should be executed without interruption. Otherwise, a corrupt system state (SP) may result. Hence, if an application is executing an atomic operation, the scheduler would have to wait until it completes the operation before suspending it. Although it is recommended that atomic operations should be as short as possible, a malicious application could declare large portions of code as atomic and hence delay, or in the worst case prevent the execution of another application. In order to prevent such attacks, the system must be able to bound the maximum duration (in time) of atomic operations.

Our architecture (Figure 3) includes an atomicity monitor that tracks the length of atomic code sections. The atomicity monitor terminates an erring application which exceeds a pre-defined execution upper bound. This requires that all applications are designed to respect this bound; otherwise they will fail to execute correctly. We note that this bound does not apply to the trusted domain. The atomicity monitor must also prevent nesting of atomic sections to increase their effective length. Furthermore, the trusted domain configures an additional CPU time-slice monitor (e.g., a timer) just before it transfers control to the individual application.

Memory Availability

Applications typically need two types of memory: program memory for their code and data memory for their stack and heap.

The use of shared stacks and heaps allows compromised applications to launch iDoS attacks by potentially exhausting stack or heap space. Recovery from such stack and heap overflow attacks requires invalidating current stack frames (e.g., by unwinding) and heap data. Although solutions that guarantee such secure stack sharing exist [24, 20], recovering from security violations can be complicated and expensive. This is because identifying and invalidating data memory regions that caused the violation can be both costly and time-consuming. For example, upon a security exception, one may have to unwind multiple stack frames which is time consuming compared to simply swapping the stack pointer. Enforcing access control policies in systems with shared data memory can also be complicated because it requires maintaining ownership information on individual stack frames and heap blocks as described in [20]. Therefore, we use dedicated data partitions for each application and depend upon the application-aware MPU (described below) to prevent iDoS attacks on data memory. A similar approach is used to ensure availability of program memory. Although such partitioned memory architectures limit flexibility, we believe they are still suitable for use in trusted scheduling architectures for such specialized systems whose memory requirements are typically known in advance.

In practice, most embedded systems use a memory protection unit (MPU) or a memory management unit (MMU) for protecting each application's state against unauthorized access or modification. Typically, such a unit only enforces access control policies (read, write, execute) at run-time on segments or pages of memory, and the operating system is responsible for separating memory regions of different applications. Our trusted scheduling architecture relies on a specialized MPU that combines these two functions, i.e., an application-aware MPU that not only checks for the type of access to memory but also whether the entity initiating such an access has the appropriate privileges. For specialized embedded devices with relatively long lifetimes, enhanced MPUs and MMUs that are application-aware (e.g., ARM [1], NIOS II [7]) present a reasonable trade-off between flexibility and better security.

In our system, the trusted domain configures the application-aware MPU with information regarding the boundaries of different applications. Every memory access (both program and data memory) is then mediated by this MPU.

Mediation of the Bus Access

If an application cannot gain access to the memory or to the peripherals that it needs for its correct execution, it will fail to execute. A compromised system component with access to a bus can cause such failures by holding on to the bus — preventing any communication among other system components that are connected to the same bus. We call this attack an iDoS attack on the bus.

In our system, we do not consider DMA-capable peripherals and assume that all peripherals access memory only through the CPU. We assume that the atomicity monitor enforces the upper bound on the length of bus transactions. This in turn prevents iDoS attacks on the system bus. We discuss this further in the security analysis (Section 3.3).

In addition to components connected to the system bus, components on the peripheral bus are also often crucial for the operation of the system or of individual applications, e.g., EEPROMs containing system configuration data may need to be accessed in a timely manner, an alarm application needs to have access to the radio peripheral to transmit alarm messages. In order to ensure the availability of the peripheral bus, we propose a secure hardware bus manager that mediates bus access requests.

In our system, we consider a multi-master (multi-)slave bus (e.g., I^2C). In most multi-master bus architectures, a typical data exchange consists of three phases: bus-arbitration, data-exchange and bus-release. Bus-arbitration is used to establish a bus-owner when there are several contenders (or masters). Arbitration is followed by data exchange and by an explicit bus-release phase in which other bus masters are informed about the availability of the bus. These three phases constitute a bus transaction and it is always executed atomically.

The shared nature of the multi-master bus allows a misbehaving peripheral to affect (by delaying or denying bus access) the execution of the applications that do not directly depend upon it. A misbehaving peripheral could deny bus access in the following ways:

(i) A misbehaving master peripheral may not respect the rules of bus arbitration and may continue its transmission beyond what it is allowed, thereby indirectly disrupting or modifying data on the bus.

(ii) A misbehaving master peripheral could gain access to the bus by sending carefully crafted data to win bus-arbitration every time and then sending data continuously without releasing the bus.

(iii) A misbehaving slave could delay its master node for arbitrary lengths of time.

In our system, we prevent these attacks by introducing a secure peripheral bus manager. This manager is configured by the trusted domain at start-up with information regarding the list of peripherals that must be accessed by each application. The manager uses this information at run-time to ensure that only peripherals that are needed by the executing application have access to the bus.

The manager further allows an application to selectively enable and disable peripherals at run-time. Such fine-grained control allows an application to only enable the peripheral that it is currently using. As a result, a misbehaving peripheral cannot influence any execution sequence during which it is not enabled and an application's interaction with any other peripheral that is not concurrently active. Together, these two features enable graceful degradation in the functionality of the application while maintaining tight security guarantees. However, little can be done if the misbehaving peripheral is critical for the application's correct execution.

Finally, the manager ensures that if an application terminates as a result of a security violation, then all the devices (peripherals) to which it had access are reset before they are re-used. We present a realization of a peripheral bus manager for the I^2C bus in Section 4.

3.3 Security Analysis

In this section, we analyze the security of our system. As described in Section 2, our attacker does not have physical access to the system, but can only remotely compromise selected applications and system components.

We assume that our trusted scheduling extensions (scheduler, trusted domain, application-aware MPU, peripheral

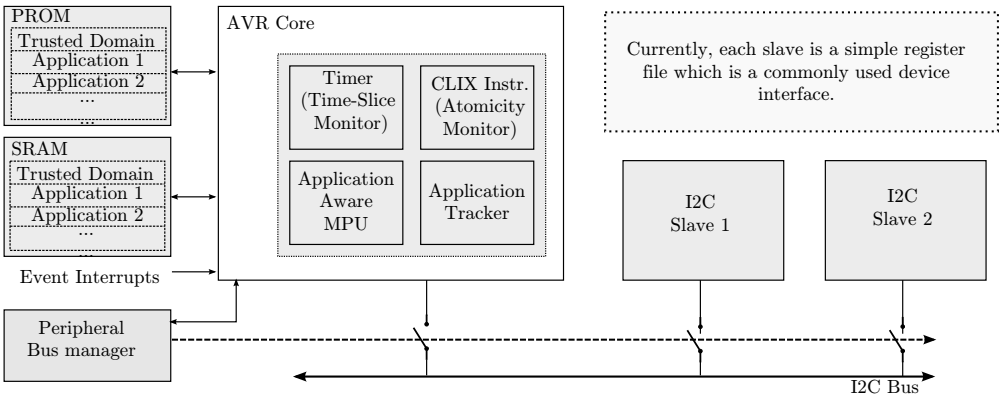

Figure 5: Our prototype implementation consists of a modified AVR core, partitioned data and program memories and a peripheral I^2C bus manager. The trusted domain also resides in its own dedicated RAM partition which other applications cannot access. The time-slice monitor is implemented using one of the AVR's timers and the atomicity-monitor is implemented as a custom instruction (`clix`). The application-aware MPU enforces memory boundaries of applications using information from the application tracker. The peripheral I^2C bus manager mediates access to the I^2C bus according to its access control map.

bus manager, time-slice/atomicity monitors) are implemented correctly and were initialized properly in a trusted initialization step that is free from adversarial interference. Furthermore, all of these components have simple logic, which reduces the risk of vulnerabilities in their implementation. We assume that the attacker cannot control inputs to the scheduler either directly (as external input) or indirectly (through applications that it controls). Allowing only the trusted domain to access the trusted scheduling components eliminates the risk of compromise of these secure elements by other applications. We assume that any system update (applications or configuration) involves a similar trusted initialization step and is adversary-free.

We analyze our system assuming that the attacker controls (up to) all of the applications on the system, except the critical application and (up to) all of the peripherals except the ones needed by the critical application. We show that even in this scenario, the execution schedule of the critical application will not be affected by the attacker. We further show that even if the attacker controls a subset of the peripherals that are used by the critical application, our system still ensures that the application executes on time, albeit with a reduced or compromised functionality due to the compromise of the peripheral.

Compromised Applications

The CPU time-slice monitor and the atomicity monitor ensure that misbehaving applications cannot indefinitely deny CPU access to other applications — no application can occupy the CPU longer than its assigned duration or execute bus transactions/atomic operations longer than a predefined length. This prevents iDoS attacks on the system bus by misbehaving applications since they can occupy the CPU (which is the only master of the system bus) only for a limited amount of time. Since an application can use a peripheral only as long as it has CPU-context, the time-slice monitor also prevents iDoS attacks against peripherals from misbehaving applications.

The application-aware MPU isolates applications such that they are restricted to using only their own code and data memory. This ensures that they do not occupy more than their share of memory space or interfere with other applications. Furthermore, the application-aware MPU and the trusted domain mediate context switches and prevent unauthorized cross-application state access and modifications.

The above countermeasures prevent one or more misbehaving applications from delaying or preventing the execution of the critical application.

Compromised Peripherals

Our system enforces that all peripherals can access memory only through the CPU (no DMA) and hence through the application-aware MPU; given this, there is no threat of iDoS attacks by peripherals on the system bus. However, a peripheral can attempt to disrupt trusted scheduling by launching iDoS attacks against the peripheral bus. This attack is prevented by the use of the peripheral bus manager that fully mediates the access of peripherals to the bus. The peripheral bus manager ensures that only the peripherals required by the currently executing application are active. This prevents other compromised peripherals from interfering with the execution of an application.

Furthermore, the system prevents peripherals from executing a long bus transaction — the length of any bus transaction (which is an atomic operation) is bounded by the atomicity monitor. Hence, misbehaving peripherals alone cannot disrupt trusted scheduling or the communication between the critical application and its peripherals.

Finally, even in the case of misbehaving applications and peripherals that are under the control of the same attacker, the combination of the above mechanisms ensures that the execution schedule of the critical application is not modified provided it does not depend on the misbehaving peripherals.

4. IMPLEMENTATION AND EVALUATION

In order to demonstrate the feasibility of realizing a trusted scheduling architecture, we implemented a prototype embedded system based on the AVR ATmega103 core. Our prototype (Figure 5) is a simplified instance of the architecture shown in Figure 3. It consists of a modified AVR core

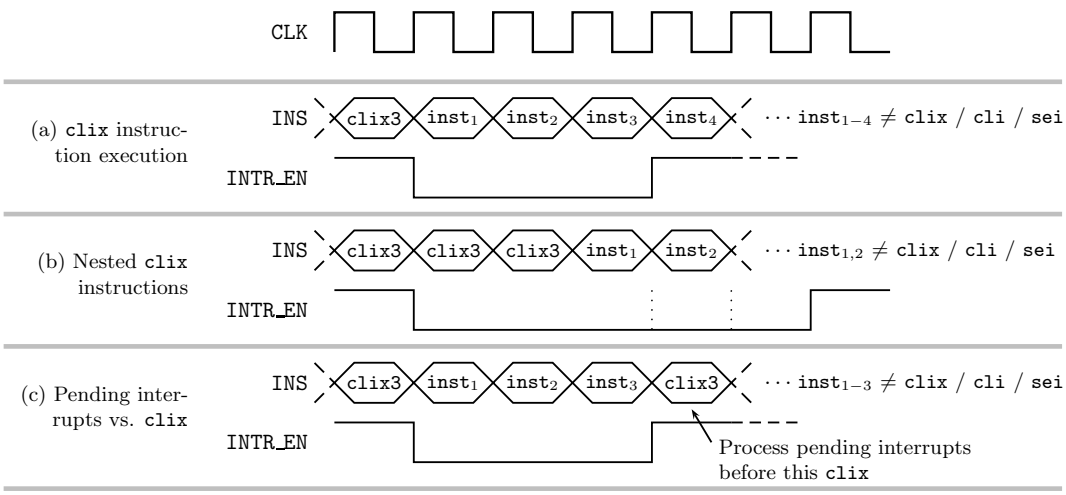

Figure 6: In order to limit the number of cycles for which interrupts can be disabled by untrusted applications, we introduce a new processor instruction (`clix`) which can be used to bound the maximum duration for which interrupts are disabled. We also implemented additional safeguards to prevent an application from disabling interrupts for a longer period by executing `clix` continuously or at regular intervals.

that is connected to two slave devices (register files) on an I^2C bus. We chose the AVR ATmega core and the I^2C bus due to their widespread use in embedded systems. In what follows, we describe our implementation in detail and then present initial results on the performance of our prototype.

4.1 Our Prototype

In our prototype, the trusted domain, which manages the scheduler, time-slice and atomicity monitors, application-aware MPU and the peripheral bus manager, is hosted in its own dedicated portion of RAM (instead of ROM) which is not accessible by other applications. The trusted domain also has exclusive access to a set of registers that hold security-critical information. This was necessary to ensure that misbehaving applications do not access or modify security-critical system parameters, e.g., the maximum length of an atomic section that is stored in a register and initialized during system start-up by the trusted domain. We added secure registers to store information regarding the currently executing application (in the application tracker), the trusted domain's stack pointer and the boundaries of different applications. We also modified the interrupt mask register (that is used to determine the set of active interrupts and hence, the set of applications that may interrupt the current application) and the timer corresponding to the time-slice monitor to be secure registers.

The scheduler is implemented using the set of prioritized interrupts available in the AVR core. It supports priority-based scheduling of applications with fixed and unique priorities. The highest priority interrupt line is reserved for the trusted domain and applications are executed as interrupt handlers. A high priority panic interrupt is fired when a security violation (as detected by the application-aware MPU, time-slice monitor, atomicity monitor or peripheral bus manager) occurs. It is important that the security panic interrupt is non-maskable, i.e., it can never be disabled. CPU preemption occurs automatically based on the priority of an incoming interrupt. The CPU time-slice monitor is implemented as a hardware timer that fires a security interrupt when it expires. The atomicity monitor is implemented as a custom instruction.

Furthermore, since the original AVR does not have an exclusive MMU or MPU module, we extend the core with a custom MPU that is initialized by the trusted domain. During system start-up the trusted domain loads the memory map of applications into the MPU. Then the MPU enforces application boundaries in program and data memory using the information about currently executing application that it obtains from the application tracker.

Memory partitions that are enforced by the MPU are created at compile time, i.e., by compiling the trusted domain and applications into a custom executable. The executable contains information regarding the load addresses of applications in RAM and their data regions (stack, heap and globals) and hence facilitates mapping of applications to separate (program and data) partitions. The trusted domain multiplexes the stack pointer between different applications. We extended the AVR core with an additional stack pointer that is used exclusively by the trusted domain. The trusted domain is also responsible for saving and restoring application contexts. We store the context of an interrupted application on the trusted domain's stack, which is accessible only by the trusted domain. This prevents any malicious application from accessing and modifying the state of any other application.

Atomicity Monitor

Since our prototype uses interrupt-based preemption mechanisms, applications disable all interrupts before executing atomic operations. In order to limit the maximum length of atomic operations, we added a custom instruction `clix Y` to the AVR core that disables all interrupts for `Y` cycles (Figure 6). We extended `gcc` (version 4.3.2) and `binutils` (version 2.20.1) to enable support for this new instruction. The value of `Y` is fixed at compile-time based on the atomic section's declared length. The generated application binaries use the custom instruction `clix` in place of the conventional `cli` instruction and a security exception is raised if any ap-

plication (other than the trusted domain) executes the `cli` instruction.

It is important to derive a practical bound on the maximum duration for which interrupts can remain deactivated by an untrusted application using `clix`. We refer to this upper bound as `max_clix`: the largest argument (maximum value for `Y`) that can be processed by a single `clix` instruction. This value is held in a dedicated register initialized by and accessible only to the trusted domain. Furthermore, enforcing the upper bound using `clix` requires the following additional safeguards:

(i) No application other than the trusted domain is allowed to execute the `cli` instruction.

(ii) Misbehaving applications may try to nest the execution of critical sections, i.e., they may execute the `clix` instruction consecutively and hence increase the effective number of cycles for which interrupts remain disabled (Figure 6). We prevent this in our prototype by ignoring `clix` instructions that occur while an older `clix` is being processed.

(iii) Pending interrupts should always be processed in precedence to `clix` instructions (Figure 6). This is important in the case when an interrupt occurs between a `clix a` and `clix b` instruction that are exactly `a` cycles apart. It must also be ensured that the `clix b` instruction is processed once the interrupt handler has completed.

Peripheral I²C Bus Manager

Our prototype includes a peripheral bus manager (Figure 7) that controls access to an I^2C bus that connects the CPU to peripherals. The I^2C bus is a two-wire serial bus. One of the wires is used for data and the other for control or clock signals. In practice, peripherals are connected to the bus using tri-state buffers (one each for the data and clock lines). Each peripheral typically controls its own tri-state buffer.

In our prototype (Figure 7), the access control map of the peripheral bus manager is initialized by the trusted domain with information regarding the set of peripherals that each application is allowed to access. Additionally, each application can choose to enable only a subset of all the peripherals to which it has been granted access using the peripheral selection register (PSR). The bus manager restricts access to the bus by controlling the enable signal of the tri-state buffer that connects the peripheral to the bus, i.e., the peripheral can access the bus only when the enable signal from the bus manager is also low. Finally, on the occurrence of a security violation (panic is high), the bus manager resets all the currently active devices so that they are ready for use by the next application that executes.

4.2 Preliminary Evaluation

In this section, we present an initial evaluation of our prototype with respect to its timing properties and hardware resource utilization.

Application Activation Latency

We evaluate the timing properties of our prototype in terms of its activation latency. Activation latency refers to the time elapsed between the arrival of a request for application

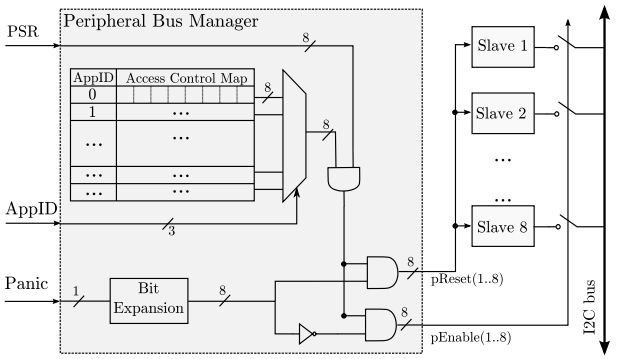

Figure 7: The secure I^2C bus manager (shown in gray above) restricts access to the bus by controlling tri-state buffers that are used to connect peripherals to the I^2C bus. The trusted domain initializes the peripheral access control map with information regarding the peripherals which each application can access. Each application can in turn enable a subset of these peripherals using the Peripheral Select Register (PSR).

execution and the actual execution of the application. When the system is idle, the activation latency is equal to the context switch time and we call this the *ideal activation delay*. In our prototype, this includes time required to save the current program counter, all the general purpose registers (32 registers in the case of the AVR ATmega103), the current stack pointer, timer register representing the remaining execution time, peripheral selection register and the interrupt mask register. This takes about 114 CPU cycles ($\simeq$9.1 μs when the processor runs at 12.5 MHz) and is our system's ideal activation time.

In the non-idle state of the system, the activation latency depends on whether execution control rests within another application or a context switch. If execution control rests within an atomic section of a lower-priority application, then the activation delay increases by the number of remaining cycles in the atomic section. Similarly, if the system is currently executing a context switch, then the activation delay increases by the number of remaining cycles in the context switch. Our measurements on the prototype implementation resulted in a worst case activation latency of 222 CPU cycles ($\simeq$18 μs at 12.5 MHz).

Finally, we calculate the system recovery time, i.e., the time taken by the system to exit an erring application and begin restoring a previous context. Our prototype implementation took 12 CPU cycles ($\simeq$1 μs at 12.5 MHz) to recover from a security panic and restore last known stable system state.

Application Execution Latency

In our prototype, the only component that directly affects the total application execution time is the MPU, which checks every memory access to guarantee its validity. The ATmega 103 has two types of data memory access instructions: direct and indirect. While indirect memory access instructions (`load` and `store`) do not incur additional delay, the direct memory access instructions require one extra cycle (i.e., they take 3 cycles instead of 2). This is because in the latter case,

the actual memory address which is fetched during the second cycle needs to be validated before the actual load/store operation.

Hardware Complexity

We implemented the trusted scheduling hardware modules as an extension to the AVR ATmega103 core in VHDL and synthesized it for a Xilinx Virtex 5 FPGA. The application-aware MPU, the time-slice monitor and the atomicity monitor together occupied 34.7% more logic units compared to the original AVR core. A major portion of the increase (about 22%) is the application-aware MPU which contains the memory map of application boundaries. However, we note that many CPUs today are already equipped with process-aware or domain-aware MMUs (e.g., NIOS II, ARM) which can potentially be used to realize application-aware MMUs at no additional hardware cost. The logic units utilized by the peripheral bus manager was insignificant (less than 0.1% of the whole system).

4.3 Discussion

In this section, we discuss the implications of choosing a bus protocol for a trusted scheduling enabled embedded system. Broadly, bus protocols can be classified as either *(i)* node-oriented (e.g., I^2C) or *(ii)* message-oriented (e.g., CAN [2]). In a node-oriented protocol, only one master and slave are active at any point in time. A simple bus manager design would be to allow access to the bus based on the priority of the master node. In addition, as implemented in our prototype, the master node can selectively enable the slave(s) that it requires for functioning. However, in message-oriented protocols like CAN, bus arbitration depends upon the priority of the message being broadcasted. Since multiple nodes can send out messages of the same priority, it is non-trivial to formulate secure bus access control policies for message-oriented protocols without any modifications to the protocols. Therefore, bus manager designs for message-oriented protocols requires further exploration.

Furthermore, in case the bus protocol uses a bi-directional bus line, peripherals may either connect to it using a single bi-directional pin [11] or using separate pins for input and output. Bus isolation circuits that allow control of physical access to the bus are much simpler in the latter scenario because it is easier to identify when a peripheral is actually transmitting. We intend to investigate this and other aspects of bus-interface designs that affect trusted scheduling as future work.

5. RELATED WORK

Given the feasibility of compromising the firmware of peripherals [15], there have been efforts to detect [22] as well as defend applications [28] against such compromised devices. The work that comes closest to ours in terms of protection against malicious peripherals is CARMA [28], which relies on Cache-as-RAM mechanism to securely sandbox applications. However, CARMA focuses on reducing the trusted computing base rather than providing trusted scheduling guarantees as presented in this work. Furthermore, our work addresses iDoS attacks by peripherals unlike CARMA which addresses attacks against confidentiality and integrity of application code and data.

Although there has been no direct work that provides guarantees similar to those of our trusted scheduling architecture, individual components of the architecture have been explored extensively in previous work. We summarize previous work on CPU scheduling, memory management and bus isolation in real-time embedded systems.

Most previous work on scheduling in real-time systems focused on optimizing the design [23, 26] and implementation of schedulers in hardware and software [30]. Today, scheduling in real-time systems is usually done by a real-time operating system (RTOS) [5, 6] or a separation kernel [3, 8]. An overview of contemporary RTOS and their performance can be found in [21, 27]. Most RTOS support the use of both co-operative and preemptive scheduling using priority- and round-robin-based algorithms. However, unlike our solution, none of these works explicitly include mechanisms to limit the length of atomic sections in code.

Process/Domain-aware MMUs are available in some of today's processors (e.g., ARM [1], NIOS II [7]). While most RTOS support the use of MMUs and MPUs, it is unclear whether they also support use of such application-aware MMUs as described in this work. Furthermore, commercial RTOS [5, 6] assign separate program and data memory partitions to each of the applications [27]. RTOS for memory constrained embedded devices ensure more efficient use of memory by sharing stack, heap and global data sections. Solutions for secure stack sharing [20, 24] and stack overflow prevention in such constrained devices [14, 17, 20] also exist.

The work in [25, 31] discusses DoS attacks against the system bus and defense mechanisms in the context of shared-memory multi-processor systems. However, these works only consider attacks by misbehaving applications running on one or more CPUs and do not take into account other misbehaving system components and peripherals. The need for fault tolerant bus design has led to the design of bus isolation solutions [11, 29]. These bus isolation solutions by themselves only provide a mechanism to physically isolate faulting devices and therefore useful for trusted scheduling only when they are configured and controlled by a context-aware bus controller as described in our design.

6. CONCLUSION

In this work, we investigated the problem of enabling trusted scheduling on embedded systems in adversarial settings. First, we identified the essential properties of a trusted scheduling system and presented an embedded system design that satisfies these properties. Our design includes a software-based trusted domain that manages the other hardware components. We analyzed the security of our proposal and showed that it achieves trusted scheduling in the presence of not only misbehaving applications but also misbehaving peripherals. Our prototype implementation based on the AVR ATmega103 shows the feasibility of realizing such an architecture through simple hardware extensions.

Acknowledgments

The research leading to these results was supported, in part, by the Hasler foundation (project number: 09080) and European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 258754.

7. REFERENCES

[1] ARM Architecture Reference Manual. http://infocenter.arm.com/help/index.jsp.

[2] CAN Specification v2.0. http://esd.cs.ucr.edu/webres/can20.pdf.

[3] Codezero. http://www.l4dev.org/.

[4] IBM X-Force 2010 Trend and Risk Report. ftp://public.dhe.ibm.com/common/ssi/ecm/en/wgl03003usen/WGL03003USEN.PDF.

[5] Integrity for Embedded Systems. http://www.ghs.com/products.html.

[6] Lynx Embedded RTOS. http://www.lynuxworks.com/rtos/rtos.php.

[7] NIOS II Processor Reference Handbox, chapter 3. http://www.altera.com/literature/lit-nio2.jsp.

[8] OKL4 Microvisor. http://www.ok-labs.com/products/overview.

[9] SPI BlockGuide v03.06. http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf, 2003.

[10] Attacks on Mobile and Embedded Systems: Current Trends. https://mocana.com/pdfs/attacktrends_wp.pdf, 2009.

[11] Designing an isolated I2C Bus interface by using digital isolators. http://www.ti.com/lit/an/slyt403/slyt403.pdf, 2011.

[12] I^2C bus specification and user manual. http://www.nxp.com/documents/user_manual/UM10204.pdf, 2012.

[13] J. Adomat, J. Furunas, L. Lindh, and J. Starner. Real-time Kernel in Hardware RTU: A Step Towards Deterministic and High-performance Real-time Systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pages 164 –168, 1996.

[14] S. Biswas, T. Carley, M. Simpson, B. Middha, and R. Barua. Memory Overflow Protection for Embedded Systems Using Run-time Checks, Reuse, and Compression. *ACM Transactions on Embedded Computing Systems*, 5(4):719–752, Nov. 2006.

[15] L. Duflot, Y.-A. Perez, and B. Morin. What If You Can't Trust Your Network Card? In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 378–397, 2011.

[16] N. Felliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier, 2011.

[17] A. Francillon, D. Perito, and C. Castelluccia. Defending Embedded Systems Against Control Flow Attacks. In *Proceedings of the 1st ACM Workshop on Secure execution of untrusted code*, SecuCode '09, pages 19–26, 2009.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, 2009.

[19] P. Kohout, B. Ganesh, and B. Jacob. Hardware Support for Real-time Operating Systems. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS'03, pages 45–51, 2003.

[20] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava. A System for Coarse Grained Memory Protection in Tiny Embedded Processors. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 218–223, 2007.

[21] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber. A Comparison of Partitioning Operating Systems for Integrated Systems. In *Computer Safety, Reliability, and Security*, volume 4680 of *Lecture Notes in Computer Science*, pages 342–355. 2007.

[22] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the Integrity of PERipherals' Firmware. In *Proceedings of the 18th ACM conference on Computer and Communications security*, CCS '11, pages 3–16, 2011.

[23] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[24] B. Middha, M. Simpson, and R. Barua. MTSS: Multitask Stack Sharing for Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 7(4):46:1–46:37, Aug. 2008.

[25] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *Proceedings of the 16th Usenix Security Symposium*, pages 257–274, 2007.

[26] K. Ramamritham and J. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-time Systems. *In Proceedings of the IEEE*, 82(1):55 –67, Jan. 1994.

[27] S. Tan and B. Tran Nguyen. Survey and Performance Evaluation of Real-time Operating Systems (RTOS) for Small Microcontrollers. *Micro, IEEE*, (99), 2009.

[28] A. Vasudevan, J. M. McCune, J. Newsome, A. Perrig, and L. van Doorn. CARMA: A Hardware Tamper-Resistant Isolated Execution Environment on Commodity x86 Platforms. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, ASIACCS'12, 2012.

[29] N. Venkateswaran, S. Balaji, and V. Sridhar. Fault Tolerant Bus Architecture for Deep Submicron Based Processors. *SIGARCH Computer Architecture News*, 33(1):148–155, Mar. 2005.

[30] M. Vetromille, L. Ost, C. Marcon, C. Reif, and F. Hessel. RTOS Scheduler Implementation in Hardware and Software for Real Time Applications. In *17th IEEE International Workshop on Rapid System Prototyping*, pages 163 –168, 2006.

[31] D. H. Woo and H.-H. S. Lee. Analyzing Performance Vulnerability due to Resource Denial-of-Service Attack on Chip Multiprocessors. In *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnects*, CMP-MSI'07, 2007.