TrustLite: A Security Architecture for Tiny Embedded Devices

Conference Paper · April 2014

DOI: 10.1145/2592798.2592824

CITATIONS
174

READS
1,283

4 authors, including:

Patrick Koeberl
Intel
24 PUBLICATIONS **819** CITATIONS
SEE PROFILE

Steffen Schulz
Intel
28 PUBLICATIONS **573** CITATIONS
SEE PROFILE

Vijay Varadharajan
Macquarie University
485 PUBLICATIONS **5,548** CITATIONS
SEE PROFILE

TrustLite: A Security Architecture for Tiny Embedded Devices

Patrick Koeberl Steffen Schulz *
Intel Corporation
{patrick.koeberl,steffen.schulz}@intel.com

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
ahmad.sadeghi@trust.cased.de

Vijay Varadharajan
Macquarie University
vijay.varadharajan@mq.edu.au

Abstract

Embedded systems are increasingly pervasive, interdependent and in many cases critical to our every day life and safety. Tiny devices that cannot afford sophisticated hardware security mechanisms are embedded in complex control infrastructures, medical support systems and entertainment products [52]. As such devices are increasingly subject to attacks, new hardware protection mechanisms are needed to provide the required resilience and dependency at low cost.

In this work, we present the TrustLite security architecture for flexible, hardware-enforced isolation of software modules. We describe mechanisms for secure exception handling and communication between protected modules, enabling seamless interoperability with untrusted operating systems and tasks. TrustLite scales from providing a simple protected firmware runtime to advanced functionality such as attestation and trusted execution of userspace tasks. Our FPGA prototype shows that these capabilities are achievable even on low-cost embedded systems.

1. Introduction

Embedded systems increasingly permeate our information society. Usage profiles, credentials and sensitive documents accumulate in various devices around us, from footwear and media players to smart TVs and smartphones [16, 45]. Printers, network routers and other inconspicuous equipment are repeatedly the subject of exploitation and abuse, jeopardizing the security of large IT infrastructures and the data entrusted to them [9, 10, 52, 53]. In emerging applications such as electronic cars and medical implants, the security of tiny, resource-impoverished embedded devices can be critical for the safety of users and bystanders [8, 18]. In this context, the strong isolation of critical software from complex peripheral drivers and protocol stacks, the binding of sensitive data to known system states and the automated validation of remote platform software stacks can provide fundamental security capabilities and facilitate the automated, secure operation of interdependent embedded systems.

While technologies such as secure co-processors [50], firmware security services [32] or verifiable execution of measured code (trusted execution) [4, 17] are commonly deployed in more capable systems such as laptops and even smartphones, for many embedded systems economic incentives drive towards lower-cost solutions requiring minimal resources. Indeed, even well-established features such as caching, virtual memory and user/superuser separation are often not included at these design points to minimize production cost and energy consumption. Instead, the software requirements are typically determined at design time and implemented using the cheapest available hard- and software.

To provide strong security assurance in low-cost embedded systems, a recent line of research [2, 11, 47] investigates an extension of memory access control to depend on the currently executing CPU instruction address. In particular, the authors of SMART [11] demonstrate that a simple measurement routine in ROM with exclusive access to a protected secret key can provide remote attestation and trusted execution. On the other hand, Sancus [38, 47] proposes additional CPU instructions that can be used to setup trusted software modules at runtime. For this purpose, they implement multiple memory protection regions, each containing a code and data section. An extended processor instruction set enables dynamic measurement and loading of code into protected regions in order to query the protection status of modules and request tokens for authenticated communication between processes.

However, neither SMART nor Sancus solve the problem of handling memory access violations and hardware interrupts, instead relying on the platform to reset the CPU and sanitize all memory. Moreover, the rather basic access control logic of SMART supports neither the update of the attestation code nor its key and interaction between multiple protected modules is very slow. On the other hand, Sancus restricts applications and incurs rather high hardware cost by

* Also affiliated with Ruhr-University Bochum, Germany and Macquarie University, Australia at the time of writing.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

implementing the loading, measurement and runtime identification of protected modules in the trusted CPU.

With TrustLite, we present a generic security architecture for low-cost embedded systems that allows OS-independent isolation of specific software modules with different trade-offs for assurance vs. flexibility. This allows manufacturers and security providers for the first time to implement custom security services such as remote management, secure updates, remote attestation and authentication services for a broad range of devices and independently of the respective deployed OS and application software. This is particularly interesting for low-cost embedded devices, where the hardware and software environment is highly dynamic, the cost of secure operating systems or hardware protection is a significant factor and a case-by-case evaluation of the individual target platforms is prohibitively expensive.

Contribution. We present TrustLite, a generic hardware security architecture for flexible and efficient software isolation on low-cost embedded devices. We introduce an Execution-Aware Memory Protection Unit (EA-MPU) as a generalization of recent memory protection schemes for low-cost devices. Programmed in software, our EA-MPU allows a flexible allocation and combination of memory and peripherals I/O regions without burdening the CPU. We solve the problem of information leakage on platform reset by introducing a simple Secure Loader sequence, which can optionally also provide a root of trust for attestation and trusted execution. We also propose a modified CPU exception engine, enabling the preemptive scheduling of trusted tasks by an untrusted OS. Our architecture supports update of software and security policy in the field and works independently of the CPU instruction set, facilitating code re-use and fast deployment.

A major strength of TrustLite is that it can be instantiated in several ways, providing different security features at different cost points, all of which can be programmed and evaluated independently of the main OS and software stack.

2. Problem and Assumptions

We consider a low-cost System on Chip (SoC) in the range of 100.000 gate equivalents, including on-chip memory and basic peripherals such as timers, interfaces for external communication peripherals and possibly cryptographic accelerators. As is the case with most embedded devices, we assume that the CPU boots from a hardwired, well-known location in non-volatile memory, such as Programmable ROM (PROM), and that any peripherals access is implemented with Memory-Mapped I/O (MMIO).

Observe that, in contrast to more costly and relatively power-hungry mobile or desktop systems, the considered target platform has no hardware support for virtualization, a secure firmware runtime or enhanced security features such as trusted execution. In particular, we do not assume a secure co-processor or CPU security extensions that provide additional privilege levels and execution modes (e.g., [3, 4, 17, 40]). Instead, the class of devices considered in this work may potentially be used *as part* of a programmable secure co-processor or smart card.

Despite strong economic incentives to minimize development and production costs, it is desirable that such platforms offer certain security services such as remote reporting of the software, secure software updates and possibly the ability to deploy and update security-sensitive services such as authentication or e-payment systems. Current platforms do not offer this capability, and a trusted OS approach incurs high costs for security evaluation of new drivers or protocols.

2.1 Terminology

We denote static (machine) code and associated data and meta-data as a *program*. Programs are typically designed for a particular functionality of some overall *application scenario*. In comparison, a *task* describes the runtime state of a program, software or firmware, including its CPU state, call stack and other volatile data. The *Trusted Computing Base (TCB)* of a task is the set of components (hardware and software) that must be secure to assure the unmodified execution of that task. Tasks which are designed and believed to implement a particular security mechanism are *trusted tasks* and we refer to them as *trustlets*.

The *owner* of a platform is authorized to install and modify the TCB, tasks and trustlets of that platform as desired, but may not (always) be the entity that is in physical control of the platform.

2.2 Adversary Model

The adversary's goal is to compromise trustlets on the platform which are not owned by the adversary and do not have any trustlets issued by the adversary in their TCB.

For this purpose, we assume that the adversary has full control over the untrusted OS and tasks running on the platform. Furthermore, the adversary can convince the platform owner to deploy arbitrary code in form of additional trustlets. However, trustlets and bootstrapping routine are assumed to operate correctly, i.e., their API contains no exploitable software bugs. The adversary also controls all communication with the platform and can eavesdrop, manipulate and intercept any communication messages. However, it is assumed that any deployed cryptographic mechanisms are secure.

We assume that the high levels of integration achievable with modern IC fabrication processes render chip-level invasive attacks such as tampering, on-chip bus probing, extracting keys from on-chip memory or fault injection out of scope for economically motivated attackers and that mitigations are in place against side-channel leakage through power, electromagnetic emissions or timing behaviour [15, 26, 27].

2.3 System Requirements

Low-cost embedded devices are used in many different scenarios. Typical requirements are fast cold start, low power

consumption and real-time processing constraints. In many cases it is desirable to update the software in the field, and the enabling of third party secure applications such as authentication and payment plug-ins is increasingly important. For security, the platform should prevent trustlets from tampering with each other and enable software environments with a minimal common TCB. In more detail, we pose the following security requirements:

Data Isolation: Trustlets are isolated in the sense that no other software on the platform can modify their code. Trustlet data can be read or modified by other trustlets according to the system policy.

Attestation: Trustlets can inspect and validate the local platform state without other software being able to manipulate the procedure.

Trusted IPC: Trustlets can inspect and validate other trustlets on the platform and establish a mutually authenticated and confidential communication channel.

Secure Peripherals: Trustlets can be provided with exclusive access to platform peripherals, such that other software cannot interfere in that interaction.

In particular, secure peripherals were shown to be essential for many applications scenarios, such as providing secure user input/output for explicit confirmation of online transactions [6, 13, 54]. Additionally, we believe that the following functional requirements are essential for providing a practical and versatile security solution:

Fast Startup: The security extensions should not significantly impact the bootstrapping delay, e.g., by having to measure large amounts of code or perform cryptographic operations at platform or trustlet initialization.

Protected State: To support fast invocation and switching between trustlets, the platform should allow trustlets to maintain a protected software state during inactivity.

Field Updates: The solution should allow updates to code, data and security policy after deployment.

Fault Tolerance: It should be possible to interrupt trustlets on unexpected errors or timeout.

3. The TrustLite Security Architecture

In the following we present the base components and procedures of the TrustLite security architecture and explain how they contribute to the memory protection and trusted execution of isolated software tasks.

3.1 Architecture Overview

The high-level hardware and software architecture of TrustLite is illustrated in Figure 1. The platform consists of a System on Chip (SoC) with (possibly insecure) external peripherals. Inside the SoC boundary, a CPU core is integrated with at least one PROM, alarm timer, and some limited amount

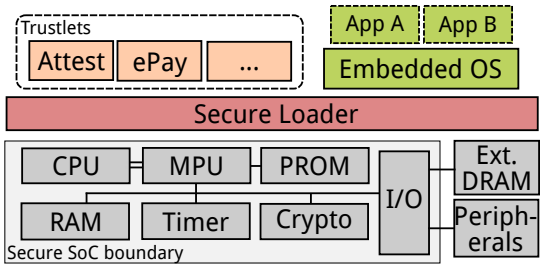

Figure 1. High-level architecture of a TrustLite platform.

of RAM. A Memory Protection Unit (MPU) enforces access control on all memory accesses, including regular memory as well as memory-mapped device I/O (MMIO). Depending on the application, a number of additional components may be included in the SoC, including cryptographic accelerators, display adapters and communication interfaces.

On top of the trusted SoC, a Secure Loader is responsible for loading all desired trustlets and their critical data regions into on-chip memory. Additionally, it programs the MPU to protect the trustlet memory regions as well as its own code and data regions from unauthorized access. This approach allows, for example, measured launch procedures [40] to be implemented without additional dedicated hardware support, although some usages may warrant hardware accelerators such as a signature verification engine.

The configured code and data regions are recorded in the *Trustlet Table*, a write-protected table in on-chip memory, such that they can be looked up and validated by individual trustlets or attestation routines. Only then does the Secure Loader continue to load and execute untrusted software, such as the embedded OS.

Note that external DRAM will not typically be used for confidential trustlet data in this approach. Instead this extended bulk memory can be used to support a larger untrusted OS and applications stack, or public code and data requiring only integrity protection.

Moreover, we emphasize that the Secure Loader itself is only active at initialization time. It only configures the hardware memory protection and optionally initializes a chain of trust for remote attestation and trusted execution before delegating control to actual runtime code, such as an untrusted OS.

In the following we discuss the key components of this architecture in more detail, before discussing the initialization and interaction of trustlets in Section 4.

3.2 Memory Protection Unit (MPU)

To realize flexible and OS-independent memory access control at low cost, we employ a generalized *execution-aware* MPU design, which also considers the address of the currently executing instruction when validating a particular data or code access.

An MPU can be seen as a lightweight Memory Management Unit (MMU). However, MMUs are primarily designed to implement paging and virtual memory, which in turn can be used to realize access control on physical memory. As such, MMUs impose a large management overhead in form of page tables, which map virtual to physical memory pages and manage their respective access rights. As the page tables typically reside in external, off-chip memory, lookups can incur significant and variable processing delays, which is unacceptable in many real-time application scenarios.

In contrast, an MPU is primarily designed for lightweight access control and does not provide virtual memory. For this purpose, available real memory is organized into a number of memory protection regions with associated access permissions. The access control rules are not kept in main memory but in local registers available to the MPU. Hence the number of protection regions is determined at production time, e.g., by instantiating an MPU with 12 or 16 such region registers [22, 49], or using faster, more expensive memory [5].

To support a larger number of protected OS tasks, an MPU is typically combined with CPU privilege levels such as supervisor and user mode execution. In this way, the OS can program the MPU rules for the next respective task to be scheduled. However, a major drawback of this approach is that the embedded OS and the facilities it implements, such as hardware drivers, communication protocols and other complex abstraction facilities, becomes a single point of failure for platform security enforcement, reducing the resilience of the overall platform to malicious attacks.

3.2.1 Execution-aware Memory Protection

Existing MPU implementations enforce execute and read-/write access control for different CPU privilege levels on a set of memory regions. For code execute permissions, addresses generated by the CPU's instruction fetch unit are checked against the programmed access control rules, while for data read/write permissions data addresses generated by the instruction execute unit are monitored.

We enhance this mechanism by providing a means to link code regions to data regions, thus making the permission check *execution-aware*. In addition to validating the data address generated by the instruction execute unit, the instruction address of the executing instruction is also considered. The resulting scheme is illustrated in Figure 2, where the MPU not only validates data accesses (*object*, read/write/execute) but additionally considers the currently active instruction pointer (*curr_IP*) as the *subject* performing the access.

Hence, our execution-aware MPU can be programmed to autonomically enforce a fine-grained access control based on individual executing code regions. Figure 3 illustrates an example access control matrix that can be programmed and enforced by our MPU, showing three subjects "TL-A", "TL-B", "OS" and several memory regions they can be given access to, including their own respective code and data regions as well as the memory-mapped registers of, e.g., the

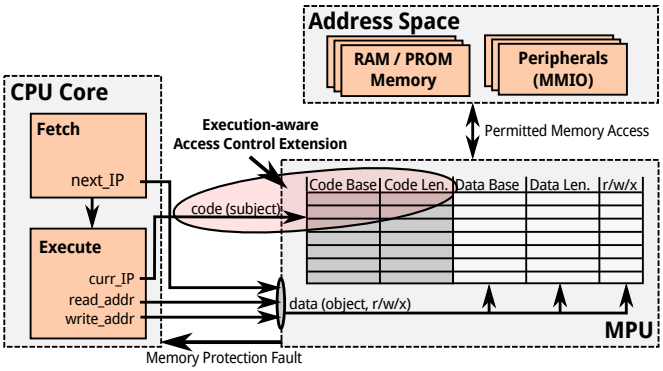

Figure 2. Block diagram of an Execution-Aware MPU.

MPU and Timer peripherals. By comparison, a regular MPU can only distinguish between user and supervisor access, and requires the OS to program the correct user-level access rules for the respective next scheduled task.

3.2.2 Memory Protection Faults

When the MPU detects a protection violation, a CPU exception is raised and handled as in regular MPU designs. In particular, the MPU protection fault invalidates the executing instruction and thus also any associated (speculatively allowed) data reads and instruction fetches. The CPU exception engine flushes the pipeline and diverts execution to the designated exception handler, providing the violating instruction address and requested data access as arguments.

3.3 Peripherals Access

The typical approach for interacting with platform peripherals is that access is limited to one or more privileged tasks. These components then implement a low-level hardware driver and provide a more abstract interface to other applications, regulating access and multiplexing available hardware resources.

TrustLite directly supports this paradigm and extends it to allow any trustlet to exclusively control an arbitrary set of hardware peripherals, without depending on the OS or other privileged code. Due to the flexible access control provided by our execution-aware MPU, trustlet code regions can be associated with a number of data regions. Since all peripherals access is implemented using Memory-Mapped I/O (MMIO), i.e., in form of read/write access to memory address space, access to individual hardware peripherals is provided by the MPU in the same way as any other memory access. For this purpose, the Secure Loader simply defines the MMIO address space of the respective peripheral as an additional read/write data region of the trustlet. This access is usually exclusive, enabling the trustlet to implement multiplexing and access control as desired.

As an example, consider the access control definitions for the MPU and Timer address regions in Figure 3. Using MMIO, the MPU is configured by writing to the appropriate

		Subject / Object		MPU Access Control Rules: TL-A (0x00-0A)	TL-B (0x0A-0B)	OS (0x0B-0F)
PROM / Flash	0x00..	Trustlet A	entry	rx	rx	rx
			code	rx	r	r
	0x0A..	Trustlet B	entry	rx	rx	rx
			code	r	rx	r
	0x0B..	OS	entry	rx	rx	rx
			code	r	r	rx
SRAM / DRAM	0x10..	Trustlet A	data	rw	-	-
			stack	rw	-	-
	0x1A..	Trustlet B	data	-	rw	-
			stack	-	rw	-
	0x1B..	OS	data	-	-	rw
			stack	-	-	rw
Peripherals	0x20..	MPU	flags	r	r	r
			regions	r	r	r
	0x2A..	Timer	period	r	r	rw
			handler(ISR)	r	r	rw
	0x2B..	...				

Figure 3. Example memory protection table, defining various memory regions and access permissions for two trustlets A and B as well as an OS.

memory regions denoted as "flags" and "regions" of the MPU peripheral. Hence, the MPU can also be configured to deny any further access to its own registers by configuring the appropriate MMIO region as read-only. We use this in the Secure Loader in Section 3.5 to prevent modification of the MPU by the OS or other software.

As another example, Figure 3 also shows a timer peripheral which can be programmed to call a particular function pointer (handler) after a configurable number of timer ticks (period). Timers are often used by an OS to interrupt the CPU and, e.g., switch to the OS task scheduler. By configuring the read-write access of this peripheral, the device can thus be setup to leverage or disable such an OS scheduler.

Observe that this capability can be applied to a variety of peripherals and usages, enabling trustlets that implement secure user I/O [54] or access cryptographic accelerators. By comparison, the Sancus task model requires that all memory and MMIO accessible for a trustlet are wired into the same continuous data region, which is unusual and requires close coordination between hardware design, software development and usage [38].

3.4 TrustLite Exception Engine

Previous works [11, 38, 47] on memory protection assumed that trusted applications are non-interruptible, resetting the platform on unexpected errors and operating in a cooperative multitasking environment. However, modern embedded systems often have high requirements with regard to fault tolerance and responsiveness, and fundamentally require efficient hardware interrupts and software fault handling.

In the following, we propose a modified CPU exception engine that maintains the memory isolation of tasks even in the case of hardware and software exceptions. This allows

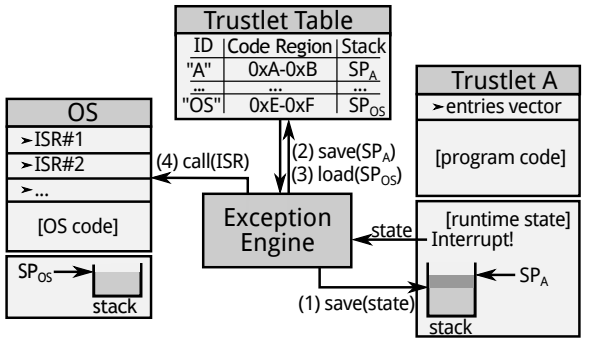

Figure 4. Exceptions save CPU state to current stack before launching handlers.

trustlets to leverage (trusted or untrusted) central scheduling services and exception handlers, without affecting the integrity and confidentiality of their data. In particular, TrustLite instantiations with a secure exception engine allow trustlets to be managed by an untrusted embedded OS, similar to other untrusted OS tasks executing on the platform.

3.4.1 Isolating Exceptions

When handling CPU exceptions such as faults, traps and interrupts, typical computing platforms only perform the minimal tasks of saving the stack and instruction pointer before executing the corresponding (software) exception handler. Depending on the type of exception, such handlers may then only save the particular CPU registers required to perform their operation, and restore them before handing control back to the task.

However, this procedure opens trustlets to information leakage attacks. In a preemptive multitasking environment, trustlets can be interrupted at any time, leaking potentially sensitive information from the CPU registers into the exception handlers and OS. To exclude the OS exception handlers from the TCB of our trustlets, we modify the CPU exception engine to store the stack and instruction pointer, as well as general purpose registers into the protected data region of the interrupted trustlet.

The detailed scheme is illustrated in Figure 4. When detecting an exception, existing exception handlers typically store the stack pointer, instruction pointer and CPU flags together with any additional exception information on the OS or other higher privileged stack. The address of this stack is taken from a well-known location, such as the Task State Segment (TSS) on current x86 CPUs. Depending on the type of exception, the control flow is then redirected to the appropriate Interrupt Service Routine (ISR) as indicated by the Interrupt Descriptor Table (IDT). To provide isolation of trustlets in the face of untrusted ISRs, we modify this scheme as shown in Figure 4. In particular, the hardware exception engine first (1) stores the CPU state to the current (task or other) stack SP_A, (2) stores SP_A to the Trustlet Table, into the row matching the currently executing instruction pointer

region, and then clears all general-purpose registers. Only then is the OS stack pointer SP_{OS} restored in step (3), unless already executing from the OS region, and the regular operation of the exception engine can continue by writing the faulting IP value as well as additional error codes onto the new stack SP_{OS}. The appropriate ISR in called step (4).

3.4.2 Return from Exception

Restarting an interrupted trustlet is performed simply by jumping to its respective entry point and does not require additional hardware logic. However, the trustlet must take care to restore its stack pointer SP_A as the very first instruction, since that instruction may already be followed by another exception leading the exception engine to store the CPU state to the wrong stack[1].

Note that the ISR and OS can distinguish the source of an exception simply by looking up the faulting IP in the trustlet table, and ensure an appropriate re-entry. Hence this design supports ISRs which directly return to the trustlet as well as approaches where the ISR defers to the task scheduler to potentially launch other trustlets and tasks before continuing the originally interrupted trustlet. Additionally, the reported faulting IP of trustlets can be sanitized to always point to the trustlet's entry vector to avoid information leakage. Our current analysis shows that the approach also works with nested interrupts, where an ISR itself may be interrupted by another ISR.

Observe that the Trustlet Table is similar to the Task State Segment (TSS) and hardware-accelerated task switching introduced in the Intel 80386 CPUs, and thus well within the scope of modern low-cost platforms. However, while x86 hardware task switching is designed for hardware acceleration, our Trustlet Table and modified exception handler are fundamental security features. In comparison to the previously proposed Intel SGX asynchronous exit [35], the entry and exit of individual trustlets is implicitly determined by matching the instruction pointer against the defined EA-MPU regions and does not require additional CPU instructions.

3.5 Secure Initialization and Reset

In the following we describe the Secure Loader, which performs the secure initialization and configuration updates of trustlets and allows an efficient platform reset.

The first process launched by the CPU upon platform reset is a platform initialization procedure that is typically loaded from a hard-wired location in memory. We realize a Secure Loader by having this initialization procedure protect itself using the MPU's memory access control. This ensures that the Secure Loader can at most be updated by itself, and not by any other software launched at a later time. The routine then loads one or more trustlets into memory and sets up the respective memory protection rules before launching the main OS.

[1] Since the MPU will typically not be configured to allow such accesses, this misbehavior leads to a memory protection fault, effectively terminating the trustlet.

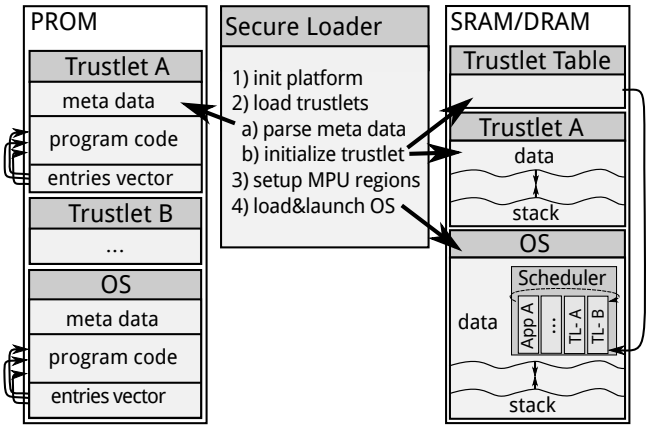

Figure 5. Bootstrapping Trustlets and OS from PROM.

Figure 5 illustrates the detailed interaction with the platform's Programmable ROM (PROM) and RAM. After performing basic platform initialization such as clearing the MPU access control registers in step (1), step (2) initiates the detection and loading any trustlets found in PROM. For each trustlet, the Loader parses the trustlets meta data and sets up the appropriate memory regions in RAM. The Loader then performs a static initialization of the trustlet, setting up its stack, instruction pointer and rewriting the code to restore its stack from the correct location in the Trustlet Table. Additionally, the global Trustlet Table is populated with the identifier, MPU regions and initial stack pointer of each loaded trustlet (cf. Figure 4).

In step (3), the Secure Loader programs the MPU in order to enforce the access control requested by the individual trustlets. For instance, code regions will usually be write-protected and data regions restricted to be read/writable only for the respective trustlet, as illustrated in Figure 3. Finally, the MPU regions themselves are locked from further unauthorized access by defining their locations in memory (as well as the MMIO locations of the corresponding MPU registers) as read-only. In this way, trustlets and even the OS are unable to interfere with the protection enforced by the MPU.

The OS may be started in step (4) to drive untrusted platform peripherals and implement any non-critical functionality. An OS can also be made trustlet-aware by inspecting the local Trustlet Table created by the Secure Loader, and registering any identified trustlets similarly to regular tasks managed by the OS. This allows the OS to invoke trustlets by calling into their appropriate entry points or manage the invocation of trustlets using the OS scheduler.

Observe that the initialization code is reliably executed at platform reset, and can efficiently reconstruct and re-establish the required memory protection rules. This allows a more efficient bootstrapping compared to prior solutions

such as SMART and Sancus, which require the hardware to sanitize all volatile memory on platform reset [11, 38].

3.6 Important Instantiations

We point out that the presented hardware architecture allows for several different instantiations, depending on the desired functionality, security level and performance.

Apart from the obvious aspect of scaling the number of MPU regions and thus available trustlets, designers may decide to hardwire certain MPU regions and memory locations to provide "hardware trustlets" similar to the ROM-based implementation of SMART. However, we expect that the secure initialization routine will be typically used to initialize load-time or "firmware trustlets" with the benefit that devices can be reprogrammed for different applications or even updated in the field. The secure exceptions engine can be included if interruptible "usermode trustlets" are desired, e.g., to enable preemptive scheduling or for the isolated execution of third party, perhaps not fully trusted code.

The Secure Loader can be extended to provide Secure Boot, i.e., to validate signatures of software components before their execution. Alternatively, designers may also use the Secure Loader to implement a minimal root of trust for measurement and reporting, e.g., by instantiating it as a hardware trustlet that also implements measurement and reporting facilities. In either case the platform may require additional secure key storage and cryptographic hardware accelerators to meet the performance and security requirements.

Naturally, trustlets may also be deployed without any attestation infrastructure at all, e.g., to provide an OS-independent remote management service similar to Intel AMT [32].

4. Trustlets Invokation and Communication

In the following we describe how trustlets are invoked and how they securely communicate with other trusted or untrusted tasks.

4.1 Memory Layout and Entry Vectors

The memory layout of a trustlet is defined by the set of memory regions and associated read, write and execute permissions granted by the MPU. A trustlet is therefore defined by a code region with associated entry vector, as well as one or more data regions which may be accessible to one or more trustlets at the same time and with different access rights. While most of these memory regions are fairly standard and self-explanatory, the entry vector requires a more detailed discussion as it is important for understanding how otherwise isolated trustlets can interact in TrustLite.

At its core, the entry vector is a code section of the overall trustlet code that may also be executed by *other* tasks or trustlets. It consists of one or more well-defined entry points which enable other tasks to call a particular sub-function of the trustlet, while at the same time assuring that a trustlet's own code execution (and thus the right to access private sub-functions and sensitive data) is limited to a particular set of entry functions. The entry vector is then defined and enforced as one of the memory regions in the MPU access control table, as illustrated by the "entry" areas for the trustlets A and B in Figure 3. In this way, only the trustlet itself can execute its complete program code region, while certain other tasks and trustlets may only execute code from the entry vector. Hence, the entry vector is the external interface of a trustlet and should be programmed with great care to avoid information leakage and other exploitation.

Figure 6 shows the two fundamental entry functions of a trustlet, `continue()` and `call()`. After initialization by the Secure Loader, `continue()` is used to continue the regular execution of the trustlet by replacing the current CPU state with previously stored values, as illustrated in the pseudo-code of "Trustlet A" in Figure 6. The functions are called simply by jumping to their position in code memory, with possible arguments in CPU registers.

Note that multiple `call()` entry points can be provided for more efficient Inter-Process Communication (IPC). Moreover, the entry functions of trustlets have similar security and functional requirements as ISRs implemented in typical operating systems. Consequently, we also list the ISRs of the OS as part of its entry vector in Figure 6.

4.2 Inter-Process Communication

Any isolation-based security system is limited by the need of applications (tasks) to communicate efficiently. In the following we discuss how untrusted tasks can communicate with trustlets while otherwise maintaining memory isolation. We also present a new mechanism for secure communication between trustlets, a local trusted channel protocol that allows efficient communication without trusted security kernels or hypervisors.

The fundamental assurance we require to achieve this kind of secure IPC, without a mutually trusted supervising entity (such as a security kernel), is that trustlets never exit. More specifically, a memory region assigned to a trustlet at boot time should not be re-used for other purposes until system reset. According to our Secure Loader setup (cf. Section 3.5), this assumption holds implicitly as no runtime entity is allowed to reconfigure the MPU[2].

4.2.1 Unprotected Communication

Typically, IPC is realized using message queues in the operating system, and tasks are notified by the OS when new messages are available. Messages can also be used to negotiate shared memory regions, allowing tasks to efficiently communicate large amounts of data.

Since any information transferred to or from untrusted tasks is implicitly already accessible to the respective un-

[2] Observe that such an entity may still be introduced, and in this case must have a notion of existing tasks and mediate IPC. However, in this case the trust relationships are similar to those of a microkernel OS, cf. [19, 41].

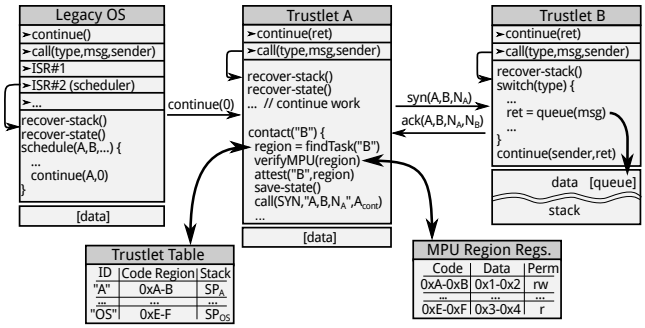

Figure 6. OS schedules Trustlet A using untrusted IPC. A then performs a local attestation of Trustlet B and establishes a trusted channel with B.

trusted parties, we can adopt the existing OS facilities to realize unprotected IPC in TrustLite. The main difference in our design is that, to maintain memory isolation between OS and trustlets, we realize signaling and short messages with trustlets in a Remote Procedure Call (RPC) fashion, by jumping to the respective trustlet entry points with arguments in CPU registers.

As illustrated in Figure 6, trustlets and OS are equipped with at least one IPC entry point `call(type, msg, sender)` for calling particular handler functions of type `type` to process message `msg`. The handler of the message may simply queue the signal in a message buffer reserved in the trustlet data region, or directly process received messages (synchronous IPC). The caller may also define a pointer `sender` to indicate the task and entry point to which the result of the call should be returned or which should be continued after queuing `msg`.

In addition to direct message transmission via CPU registers, larger messages can also be transmitted indirectly, by referring to their location in memory as part of `msg`. For this purpose, the trustlets meta-data should indicate the size and participating tasks for any desired shared memory regions, such that the Secure Loader can configure appropriate access rules in the MPU. Ideally, the program code of the desired participants should be in adjacent memory regions. In this way, only one code and data region register is needed to provide all authorized tasks with access to the particular memory region, whereas otherwise four or more MPU regions would have to be allocated.

Observe that the ability to realize fine-grained isolation and shared memory is heavily limited by the number of MPU region registers available in the particular platform. However, the nature of the trustlets suggests that the number of information flows per trustlet is fairly limited and predictable, and the considered low-cost application scenarios require only a limited flexibility.

4.2.2 Communication between Trustlets

Re-using OS facilities for IPC is cost-effective, but exposes exchanged data to manipulation, interception or injection attacks by untrusted components. Moreover, the receiver of a call has no assurance about the *sender* of an IPC. For secure and mutually authenticated communication between trustlets, we propose a simple handshake protocol to enable local trusted channels.

Our design allows trusted IPC to be initiated with a one round handshake, as illustrated in the communication between trustlet A and B in Figure 6. To start communication, the *initiator* ("Trustlet A") may first perform a local platform inspection to ensure the correctness of the platform configuration and MPU security policy enforcement for the task or trustlet to be contacted ("Trustlet B", *responder*). For this purpose, the initiator uses the responders identifier or other meta-data to look it up in the Trustlet Table and determine its memory location and relevant code entry points. Using this information, the initiator may perform an additional sanity check by validating the correct isolation of the responders memory regions in the MPU registers. The initiator may also validate a cryptographic hash of the responders program code to ensure that its code was not maliciously modified, has the latest patch level etc. This can be done by requiring read access to the responders program code for the initiator, or by having a trusted third party such as the Secure Loader measure the trustlets code region at load time and store it in a well-known location such as the Trustlet Table. Note that memory reads of the MPU registers, Trustlet Table or program code are secure from manipulation by third parties. Since the MPU does not provide virtual memory, an interruption followed by a remapping of the memory region is not possible.

Once the initiator completes its local attestation of the platform and the peer IPC trustlet, it sends a `syn`() message containing the identifier "A" of the initiator, "B" of the responder and a nonce N_A. On receiving the `syn`(), responder B may in turn perform a local attestation of the initiator A and on success respond with a corresponding `ack`(), containing the elements of the initial `syn`() as well as a nonce N_B chosen by the responder.

Since the identity of message *receivers* is ensured by the use of code entry points and secure exception handling, and since the peers can ensure with local attestation that their respective IPC receivers will not disclose the nonces, a cryptographic session token $tk_{A,B} \leftarrow \texttt{hash}(A, B, N_A, N_B)$ can then be used to authenticate subsequent messages in either direction.

Similar to untrusted IPC, trusted IPC can also be used to validate the configuration of a shared memory region.

5. Evaluation

Depending on the requirements of the application scenario, TrustLite may be deployed in a variety of configurations. In the following, we discuss a minimal and full instantiation of TrustLite in more detail and compare them against two

	TrustLite		Sancus	
	Regs	LUTs	Regs	LUTs
Base Core Size	5528	14361	998	2322
Extension Base Cost	278	417	586	1138
Cost per Module	116	182	213	307
Exceptions Base Cost	-	-	-	-
Except. per Module	34	22	-	-

Table 1. FPGA resource utilization of execution-aware memory protection per security module and comparison with Sancus [38].

previously proposed low-cost trusted computing solutions, SMART [11] and Sancus [38].

5.1 Prototype Implementation

We implemented our extensions on the Intel Siskiyou Peak research platform [43]. Siskiyou Peak is a 32-bit, 5 stage pipeline, single-issue architecture targeted primarily at embedded applications. The processor is organized as a Harvard architecture with separate buses for instruction, data and memory mapped IO spaces and is fully synthesizable. The target technology was a Xilinx Virtex-6 FPGA. In particular, we realized execution-aware memory protection by linking the respective code and data regions provided by the stock MPU as illustrated in Figure 2, using the first four bytes of each code region as its respective entry vector. Additionally, a 32 bit register storing the secure stack pointer location of each trustlet is associated with each code region to facilitate secure exception handling. Our modified exception engine logic required only minimal modifications, since the process of pushing the CPU state on the stack is largely similar to the regular exception engine behavior.

On the software side, we deployed a homegrown OS due to lack of available open source embedded OS with support for userspace tasks and memory protection. We fitted the kernel's bootstrapping routine to work as Secure Loader, programming the EA-MPU and protecting itself from later modifications by the runtime OS code. Currently, we use a linker script for the GNU C compiler to arrange code and data regions in the memory image such that they can be recognized and protected by the Secure Loader.

5.2 Hardware Extensions Cost

Table 1 lists the hardware cost of TrustLite in terms of FPGA registers and LUTs. In particular, we list the hardware overhead of the full execution-aware MPU extension in terms of cost per security module and extrapolated base cost with zero supported protection regions. To enable a better comparison with prior work, we consider a security module to consist of two associated MPU regions, one for code and one for the data of the security module, although our implementation also supports multiple code and data regions per module. We should point out that Sancus extends an open source version of the 16-bit TI MSP430 microcontroller while our implementation platform is a 32-bit architecture. In terms of FPGA resource utilization, the unmodified Sancus MSP430 core consumes 998 registers and 2322 LUTs in Xilinx Spartan-6. Our targeted 32-bit core consumes 5528 registers and 14361 LUTs in Xilinx Virtex-6 (this figure also includes a 16550 UART). Both Virtex-6 and Spartan-6 share a 6-input LUT architecture and organize these in 'slices' containing 4 LUTs with 8 registers; comparisons between Sancus and TrustLite at the LUT/register or slice level are therefore appropriate.

We now consider the base and per module cost of the security extensions. TrustLite's fixed costs are 50% of Sancus while the per module cost is roughly 40% less. It should be pointed out the disparities in hardware costs between TrustLite and Sancus are to some extent a result of differing architectural features. Sancus implements a set of instruction set extensions and instantiates a hardware hash implementation which accounts for the increased base cost. While a hash implementation (hardware or software) is not strictly required by TrustLite, there is ample base cost margin to absorb a hardware hash such as Spongent, which has been shown to consume 22 Spartan-6 slices in a representative implementation [30]. Note also that the TrustLite EA-MPU supports a wider 32-bit address space. Scaling our EA-MPU to support the narrower MSP430 16-bit datapath would roughly result in a further 50% saving in FPGA resources.

At the per module level a 128-bit MAC key is cached by Sancus resulting in a module storage cost of 128 bits which accounts for a significant portion of the register cost. For usages where performance is not a concern, moving to on-the-fly module key generation would result in a 128 register saving for Sancus, however the hardware cost for TrustLite remains competitive when scaling to a 16-bit data path is taken into account.

To better illustrate the impact of our proposed architecture, Figure 7 plots the total hardware cost in FPGA slices[3] depending on the number of supported security modules, irrespective on the employed underlying core. This is reasonable since our EA-MPU is portable to many other processors, performing essentially the same logic with similar effort, as evidenced by several regular MPUs implementations available today [5, 22, 49]. As can be seen in Figure 7, the overhead of Sancus protected software modules rises quickly to twice of the cost of their underlying open-MSP430 core, allowing them to fit only 9 protected modules at a design point where TrustLite supports 20.

Table 1 also shows that the additional overhead of secure exception handling is very low, with the additional fixed costs even staying within the error margin of the probabilistic FPGA synthesis. Figure 7 better illustrates the slightly increased cost of adding module protection with secure exceptions. Hence, our approach of managing trustlets in a multitasking environment with central scheduling and software fault handling appears quite practical.

[3] A slice in Xilinx Spartan-6 or Virtex-6 contains 4 6-input LUTs and 8 registers.

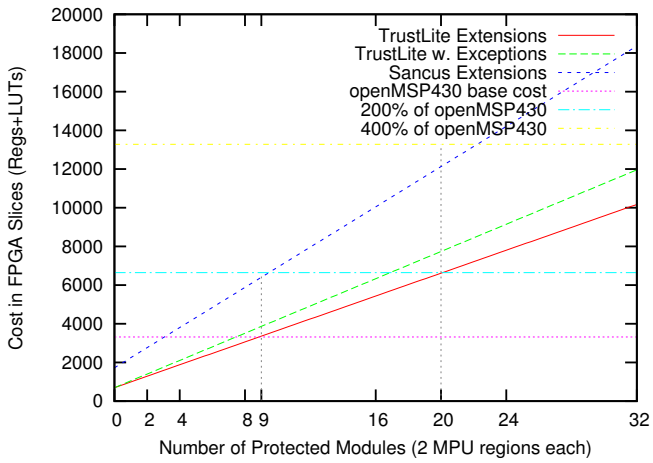

Figure 7. Hardware overhead of TrustLite and Sancus in total FPGA slices. Despite supporting a 32 bit address space, TrustLite incurs only about half the hardware overhead of Sancus in both, fixed cost and per module cost.

The rather low fixed cost of our extensions also justifies instantiations with only only one or two protected modules. In particular, a SMART-like system can be instantiated by merging the security-sensitive SMART code with that of our Secure Loader (which does not require entry points on its own), thus creating a protected attestation and trusted execution service using only a single protected module. With a hardware overhead of only 394 slice registers and 599 slice LUTs, such an instantiation can be very attractive compared to the original SMART instantiation that requires an extra 4kB ROM and does not allow software updates [11].

5.3 Runtime Overhead of Memory Protection

As already observed in prior work [38], memory region range checks can be parallelized such that they do not increase memory access time which is in the processor critical path. However, the logic which generates the collective memory access exception logarithmically increases in depth with the number of checked memory regions. We experienced no timing closure problems with up to 32 memory protection regions.

The overhead of initializing trustlets in the Secure Loader is also minimal, requiring only three additional writes to MPU registers for each protection region to define the start, end and permission of that region. Alternatively, the MPU access control rules may also be hard-wired into the MPU to further simplify the instantiation and provide additional security assurances. Note that even such limited instantiations may still be configured to allow software updates, by declaring the code region of a trustlet as writable to itself or to a separate software update service.

Overall, TrustLite enables multiple alternative instantiations with an overall bootstrapping and memory access overhead that is insignificant or not present at all.

5.4 Runtime Overhead of Exception Handling

The only hardware-dictated runtime overhead in TrustLite comes from the (optional) secure exception engine. Since the regular exception engine performs only the minimum work necessary and delegates the OS to store and restore general purpose registers, our modified flow of storing and clearing the complete CPU state before switching into the Interrupt Service Routine (ISR) yields a notable runtime overhead by the exception hardware.

In particular, the unmodified exception engine requires about 21 CPU cycles from recognizing the exception to executing the first ISR instruction, which includes the main work of restoring the OS stack and storing the exception error parameters together with the interrupted execution state onto that stack. On top of this flow, our secure exception engine flow takes another 2 cycles to recognize that a trustlet is being interrupted, 10 cycles to store all but the ESP registers onto the trustlet (instead of OS) stack, and 9 cycles to clear all general purpose registers and store the ESP into the Trustlet Table (cf. Section 3.4). Overall, our secure exception handler thus incurs a runtime overhead of 21 cycles or 100% of the regular exception flow when interrupting a trustlet, and 2 cycles otherwise.

Considering that a 32-bit i486 CPU takes at least 107 cycles for context switching and larger processors typically require significantly more, we believe this overhead is reasonable [20]. Furthermore, observe that task interruption is often followed by the invocation of the OS scheduler. For our hardware-protected trustlets, the software would be working to manage already cleared CPU registers, thus wasting CPU cycles in the time critical ISR and scheduler code. As part of our future work, we therefore plan to investigate how ISR and OS optimizations can further reduce the average overhead of trustlet exceptions.

6. Security Considerations

In the following we informally argue how TrustLite meets each of the system and security requirements in Section 2.3.

Data Isolation. As a low-end platform without multiprocessing and virtual memory, TrustLite achieves data isolation simply by memory access control. Hardware prevents unauthorized software from accessing trustlet code and data, restricting it to the explicitly provided code entry points of the respective trustlet. Our secure exception engine extends this separation into the CPU register files, ensuring that no data is unintentionally leaked on context switch. Like all known memory protection schemes, the approach is vulnerable to software exploitation attacks which may subvert trustlet code to leak information [51]. However, while TrustLite assists in reducing the vulnerability of trustlets by minimizing their overall software complexity (TCB), advanced software protection is outside the scope of this work and we instead refer to our assumption in Section 2.2 that trustlet code is correct.

Attestation. Building on a physical memory address space with full isolation of trustlets from other software, TrustLite ensures that the OS or other software cannot manipulate the outcome of memory read accesses of the MPU register set or other trustlet's code regions. Hence, any software with the required read permissions can inspect relevant system state and perform actions depending on the outcome of this inspection, without other software being able to interfere.

Trusted IPC. Since TrustLite establishes trustlets at platform initialization time and ensures their protection until system reset, a single inspection and validation of another trustlet's code and meta-data is sufficient to establish its identity, integrity and entry points for IPC. Moreover, since IPC with other tasks consist essentially of a *jump* instruction into their respective code entry region, the receiver identity and message confidentiality are directly enforced by the CPU. Trusted IPC, consisting of a confidential and mutually authenticated local channel with integrity verification of either endpoint can therefore be established using a single round-trip protocol, as described in Section 4.2.2.

Secure Peripherals. The unique memory isolation architecture of TrustLite enables a simple and direct extension to secure peripherals access based on Memory-Mapped I/O (MMIO). This enables the construction of secure device drivers and extends trusted software execution to assured interaction with device users or other aspects of a platform's environment. For future work, we want to extend this secure interaction to (possibly untrusted) devices with Direct Memory Access (DMA) capability, which were shown to be problematic for certain security architectures [42].

Fast Startup. While prior works [11, 38] require the hardware to purge all volatile memory on platform reset, our Secure Loader allows secure re-initialization of the memory protection rules before invoking untrusted software and only needs to clear data regions that should be re-allocated to other software. Moreover, TrustLite allows the measurement of trustlet code regions on demand, reducing the platform initialization overhead.

Protected State. Similar to other more recent approaches to trusted execution [35, 38], TrustLite allows trustlets to maintain a transient execution state accross invocations. This significantly reduces the activation latency of trustlets, resulting in reduced power consumption and better application performance and user experience.

Field Updates. TrustLite security extensions are independent of the CPU instruction set and completely programmable by software. This enables software updates to any trusted or untrusted software, security policy and potentially also the Secure Loader itself.

Fault Tolerance. Our secure exception engine and secure peripherals enable TrustLite instantiations that tolerate software faults in trustlets or even in the OS. In particular, TrustLite trustlets can cooperate with an untrusted OS but may also implement ISRs and hardware drivers on their own, thus preventing trivial denial-of-service attacks that affect several prior works [11, 34, 38, 48].

Overall, TrustLite achieves all security and functional requirements while being comparable or better than previously proposed solutions.

7. Related Work

Memory protection is perhaps the oldest hardware security mechanism in computing. Currently deployed systems typically use an MMU or MPU to prevent less privileged software from manipulating foreign memory. Alternatively, the Intel 80286 introduced memory segmentation where the hardware associates segments with a two bit privilege level and prevents access by software unless it is running at least at the same privilege level. Execution-Aware Memory Protection is fundamentally different from these concepts as it is not based on privilege levels but directly associates code and data regions in memory. This allows us to provide a large number of isolated software modules which are isolated independently of the OS or another trusted software runtime.

CPU Extensions. Several works also proposed CPU extensions to protect the execution of software. Aegis [48] loads code from untrusted memory and performs integrity verification and memory encryption in the CPU. Alternatively, it was proposed to extend the code of trusted software modules with integrity check values that are automatically verified by the CPU once loaded into its cache registers [12]. Intel Trusted Execution Technology (TXT) [17] and AMD Pacifica [1] introduce a secure re-initialization of the CPU at runtime. Intel SGX [35] provides hardware-protected execution of multiple concurrent userspace applications and transparent RAM encryption. However, these existing works focus on multi-core servers, PCs, and rich mobile platforms. At this point, it not clear how techniques employing memory encryption or code authentication can be reasonably downscaled to the low-cost embedded devices considered here.

Secure Hypervisors and Kernels. A large amount of research considers how to extend and improve software protection based on newly introduced privilege-based protection facilities [4, 37]. Higher privilege modes can be used by a secure hypervisor or OS kernel to provide strong isolation of certain critical software modules [19, 25, 31, 33, 36, 41], assuming that sufficient attention is paid to reduce the complexity of the privileged secure OS or hypervisor [7, 21, 54]. We believe that a secure and minimal OS kernel can be of great benefit for increasing the resilience of TrustLite systems, while at the same time TrustLite can be useful for security kernels to simplify and automate software isolation.

Software Attestation. Software-based attestation exploits the computational limitations of a platform to ensure that only a particular algorithm can be executed within a given

time frame. The scheme has been applied to a variety of platforms and use-cases, including sensor networks, voting machines and trusted key deployment [14, 28, 29, 46]. Since purely software-based attestation must assume any stored keys to be compromised, several works also consider the combination of software attestation with hardware trust anchors [23, 24, 28, 44, 47]. However, even if assuming a minimal hardware trust anchor, the scheme requires a temporary full utilization of the platform's computational capacity, and is unable to sustain more than one trusted execution environment. Hence, software attestation appears more suitable for attestation of legacy devices or initial trust establishment for platforms without trusted attestation keys and certificates.

SMART. SMART [11] enables remote attestation and trusted execution on low-cost systems using a custom access control on the memory bus. Specifically, they only allow access to a particular secret key in memory if the current CPU instruction pointer points to a trusted code region in ROM. The instruction pointer in turn may only point into this trusted code if it previously also pointed into the same region or if it currently points to the very beginning of that code. As a result, the secret key is only accessible by the trusted code in ROM, and can be used to prove to other parties that the ROM code has been executed correctly, e.g., has performed a measurement of some local software.

While our memory access control model is inspired by SMART, we extend and generalize it in several ways. Instead of deploying an attestation and trusted execution routine in ROM with custom instruction pointer restrictions, TrustLite realizes a generic, programmable access control logic, enabling a large variety of software programs that can be updated in the field. Moreover, TrustLite supports concurrent execution of trusted applications whereas SMART requires applications to store and restore their state on each invocation, resulting in significant overhead.

Sancus and SPM. Software-Protected Modules (SPM) [47] proposes new CPU instructions that allow tasks to be measured and loaded into protected memory regions and to query the status of other task's memory protection. Assuming publicly readable program code segments, protected tasks can then inspect and attest each other in physical memory. Sancus [38] implements and extends SPM for the openMSP430 CPU. They store measurements of loaded tasks in protected registers and provide special instructions to authenticate and accelerate task IPC.

Similar to TrustLite, SPM enforces certain code entry points for communication and multi-tasking. However, IPC and inspection of other tasks is simpler in TrustLite since our EA-MPU protection rules persist until platform reset. On the other hand, our more generic handling of memory protection regions allows trustlets to have multiple code and/or data regions, enabling secure peripherals access and simple shared memory constructions. In contrast to Sancus and SPM, TrustLite also supports the secure interruption of trusted tasks, and our Secure Loader solves the problem of sanitizing memory after platform reset by simply restoring the required access protection rules.

8. Limitations

We acknowledge that TrustLite also suffers from certain limitations. Notably, the limited number of supported memory protection regions in the MPU and the fact that trustlet memory regions cannot be re-allocated without a platform reset. Considering the typically very targeted and tight requirements on low-end embedded systems, we believe that these limitations are reasonable as they allow a simplified (low-cost) design of the memory protection and IPC subsystems. In particular, MPUs are well established in embedded systems and the software stack of such systems is usually very carefully adapted for the particular use case. Since most of the security mechanisms in TrustLite are programmable in software, a vendor may also deliberately deploy a hardware platform that does not support all possible usages simultaneously but instead detects the desired scenario and establishes the required software stack and protection facilities in a second boot phase during deployment.

9. Conclusion

We have presented TrustLite, a new security architecture for providing trusted computing functionality on low-cost embedded systems. Our design represents a new alternative for isolating secure applications, providing trusted execution, OS interoperability and secure peripherals access.

Our architecture is enabled by a generalized memory protection scheme, combining an execution-aware memory protection unit (EA-MPU) with a secure exception engine that protects the task state from untrusted exception handlers. Our Secure Loader provides a simple yet flexible establishment and update of the platform's security policy, prevents memory leakage after platform reset and can be extended to act as a root of trust for remote attestation and trusted execution. Depending on the application scenario and cost constraints, TrustLite can be instantiated in several configurations, from providing a single atomic firmware security service to isolating userspace tasks in a preemptive multi-tasking environment.

For future work, we want to investigate the integration of cryptographic accelerators with TrustLite and evaluate its impact on IPC performance and context switching.

References

[1] Advanced Micro Devices (AMD). *AMD64 Virtualization Codenamed "Pacifica" Technology - Secure Virtual Machine Architecture Reference Manual*, 2005.

[2] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2012.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Research in Security and Privacy (S&P)*. IEEE, 1997.

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

[5] ARM Limited. *ARMv7-M Architecture Reference Manual – Chapter B5, Protected Memory System Architecture (PSMA)*, 2009.

[6] F. F. Brasser, S. Bugiel, A. Filyanov, A.-R. Sadeghi, and S. Schulz. Softer smartcards - usable cryptographic tokens with secure execution. In *Financial Cryptography*. International Financial Cryptography Association (IFCA), Springer, 2012.

[7] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith. VM-based security overkill: A lament for applied systems security research. In *New Security Paradigms Workshop (NSPW)*. ACM, 2010.

[8] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. USENIX, 2011.

[9] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *Proceedings of the Network and Distributed Systems Security (NDSS)*. Internet Society, 2013.

[10] A. Cui, J. Kataria, and S. J. Stolfo. Killing the myth of Cisco IOS diversity: recent advances in reliable shellcode design. In *Workshop On Offensive Technologies (WOOT)*. USENIX, 2011.

[11] K. E. Defrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2012.

[12] J. Dwoskin and R. Lee. Hardware-rooted trust for secure key management and transient trust. In *Computer and Communication Security (CCS)*, 2007.

[13] A. Filyanov, J. M. McCune, A.-R. Sadeghi, and M. Winandy. Uni-directional trusted path: Transaction confirmation on just one device. In *Dependable Systems and Networks (DSN)*. IEEE, 2011.

[14] J. Franklin, M. Luk, A. Seshadri, and A. Perrig. PRISM: Human-verifieable code execution. Technical report, Carnegie Mellon University, 2007.

[15] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*. Springer-Verlag, 2001.

[16] A. Grattafiori and J. Yavor. The outer limits: Hacking the samsung Smart TV. Black Hat Briefings, 2013.

[17] J. Greene. *Intel Trusted Execution Technology*, 2012. White Paper.

[18] D. Halperin, T. Heydt-Benjamin, B. Ransford, S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, 2008.

[19] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza secure-system architecture. In *Collaborative Computing: Networking, Applications and Worksharing*. IEEE, 2005.

[20] G. Heiser. Critique of microkernel architectures. `www.cse.unsw.edu.au/~cs9242/04/lectures/lect05b.pdf`, 2004.

[21] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *SIGOPS Oper. Syst. Rev.*, 40(1):95–99, 2006.

[22] Infineon Technologies. *XC238xE User's Manual*, 2011. V1.0 2011-01.

[23] M. Jakobsson and K.-A. Johansson. Retroactive Detection of Malware With Applications to Mobile Platforms. In *Workshop on Hot Topics in Security (HotSec)*. USENIX, 2010.

[24] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *USENIX Security Symposium*. USENIX, 2003.

[25] 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 *Symposium on Operating Systems Principles (SOSP)*. ACM, 2009.

[26] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. Springer-Verlag, 1996.

[27] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. Springer-Verlag, 1999.

[28] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for Timing-Based attestation. In *Security & Privacy (S&P)*. IEEE, 2012.

[29] Y. Li, J. M. McCune, and A. Perrig. VIPER: verifying the integrity of PERipherals' firmware. In *Computer and Communications Security (CCS)*. ACM, 2011.

[30] R. Maes, A. V. Herrewege, and I. Verbauwhede. Pufky: A fully functional puf-based cryptographic key generator. In *CHES*. Springer, 2012.

[31] R. J. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun. Enabling trusted scheduling in embedded systems. In *ACSAC*. ACM, 2012.

[32] Matthew Gillespie. Architecture guide: Intel® active management technology, 2008.

[33] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: efficient TCB reduction and attestation. In Oakland [39].

[34] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *European Conference on Computer Systems (EuroSys)*. ACM, 2008.

[35] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[36] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *SIGPLAN/SIGOPS Virtual Execution Environments (VEE)*. ACM, 2008.

[37] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, 2006.

[38] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium,*. USENIX, 2013.

[39] *Proceedings of the Research in Security and Privacy (S&P)*. IEEE, 2010.

[40] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In Oakland [39].

[41] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner, and A. Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research, 2001.

[42] D. R. Piegdon and L. Pimenidis. Targeting physically addressable memory. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2007.

[43] J. Rattner. *Extreme scale computing*. ISCA Keynote, 2012.

[44] A.-R. Sadeghi, S. Schulz, and C. Wachsmann. Lightweight remote attestation using physical functions. In *Wireless Network Security (WiSec)*. ACM, 2011.

[45] T. S. Saponas, J. Lester, C. Hartung, S. Agarwal, and T. Kohno. Devices that tell on you: privacy trends in consumer ubiquitous computing. In *USENIX Security Symposium*. USENIX, 2007.

[46] A. Seshadri, M. Luk, and A. Perrig. SAKE: Software attestation for key establishment in sensor networks. In *Distributed Computing in Sensor Systems*. Springer, 2008.

[47] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks (SecureComm)*. Springer, 2010.

[48] E. Suh, C. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random function. In *International Symposium on Computer Architecture (ISCA)*, 2005.

[49] Texas Instruments. *KeyStone Architecture – Memory Protection Unit (MPU) User Guide*, 2013.

[50] Trusted Computing Group (TCG). *TCG Architecture Overview*, 2004.

[51] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory errors: the past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2012.

[52] J. Viega and H. Thompson. The state of embedded-device security (spoiler alert: It's bad). *Security Privacy, IEEE*, 10(5):68–70, 2012.

[53] K. Wilhoit. The SCADA that didn't cry wolf – who's really attacking your ICS devices – part deux! Black Hat Briefings, 2013.

[54] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Research in Security and Privacy (S&P)*. IEEE, 2012.