Lightweight and Flexible Trust Assessment Modules for the Internet of Things

Jan Tobias Mühlberg(✉), Job Noorman, and Frank Piessens

iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
jantobias.muehlberg@cs.kuleuven.be

Abstract. In this paper we describe a novel approach to securely obtain measurements with respect to the integrity of software running on a low-cost and low-power computing node autonomously or on request. We propose to use these measurements as an indication of the trustworthiness of that node. Our approach is based on recent developments in Program Counter Based Access Control. Specifically, we employ Sancus, a light-weight hardware-only Trusted Computing Base and Protected Module Architecture, to integrate trust assessment modules into an untrusted embedded OS without using a hypervisor. Sancus ensures by means of hardware extensions that code and data of a protected module cannot be tampered with, and that the module's data remains confidential. Sancus further provides cryptographic primitives that are employed by our approach to enable the trust management system to verify that the obtained trust metrics are authentic and fresh. Thereby, our trust assessment modules can inspect the OS or application code and securely report reliable trust metrics to an external trust management system. We evaluate a prototypic implementation of our approach that integrates Sancus-protected trust assessment modules with the Contiki OS running on a Sancus-enabled TI MSP430 microcontroller.

Keywords: Internet of Things · Wireless sensor networks · Trust assessment · Trust management · Protected software modules

1 Introduction

In the past decades, security research and security practice has focused on desktop and server environments. While threats to these systems grew with increased interconnectivity and deployment in safety-critical environments, elaborate security mechanisms were added. Of course these mechanisms impose certain costs in terms of a performance decrease on the host system. However, with the availability of more potent hardware, these costs quickly became acceptable to a degree where virus scanners, firewalls and intrusion detection systems can operate in the background of every modern off-the-shelf PC.

Ongoing developments in our ever-changing computing environment have lead to a situation where every physical object can have a virtual counterpart on the Internet. These virtual representations of things provide and consume

G. Pernul et al. (Eds.): ESORICS 2015, Part I, LNCS 9326, pp. 503–520, 2015.
DOI: 10.1007/978-3-319-24174-6_26

services and can be assigned to collaborate towards achieving a common goal. While this Internet of Things (IoT) brings us unpreceded convenience through novel possibilities to acquire and process data from our environment, the situation with respect to the safe and secure deployment and use of such extremely interconnected devices is quite different from the server-and-desktop world [26].

Devices in the IoT may be equipped with inexpensive low-performance microcontrollers that provide just enough computing power to periodically perform their intended tasks, i.e. obtain sensor readings and communicate with other nodes. Many nodes are required to operate autonomously for extended periods of time, solely relying on battery power as they are deployed in environments where maintenance is difficult or even impossible. Yet, all these devices are interconnected and thereby exposed to physical as well as virtual attacks. Even if we do not consider malicious attempts to disrupt a node's function, the autonomous mode of operation, exposure to harsh environmental conditions and the resource scarceness of small microcontrollers, make these systems prone to malfunction and the effects of software aging [7] – while other systems may critically depend on the reliability and timeliness of information obtained from these devices.

The problem of trustworthiness and trust management of low-power low-performance computing nodes has been discussed in previous research, in particular in the context of Wireless Sensor Network (WSNs) [12,16,19]. Importantly, most techniques proposed in this field focus on observing the communication behaviour and on validating the plausibility of sensor readings obtained from network nodes so as to assess the trustworthiness of these nodes. This approach to trust management is suitable to detect the systematic failure or misbehaviour of single nodes. However, failures or misbehaviour of a node may not be detected immediately: the quality of readings from a sensor may degrade gradually, software failures may lead to non-deterministic behaviour or a node may be captured by an attacker, exposing benign and malicious behaviour alternately. In all these cases the malfunctioning node may produce a number of measurements that are accepted as trustworthy by the network before the network will begin to distrust the node. We believe that this shortcoming can be mitigated by employing light-weight security mechanisms that guarantee the integrity and secrecy of programs and data directly on WSN or IoT nodes. An approach to do so with only marginal interference with legacy code is presented in this paper.

Our Contribution. We describe a novel approach to securely obtain measurements with respect to the integrity of the software that runs on a minimalist computing node autonomously or on demand. We use these measurements as an indication of the trustworthiness of that node. Our approach is based on Sancus [23], a light-weight hardware-only Trusted Computing Base (TCB) and Protected Module Architecture (PMA) [28]. Sancus allows us to integrate trust assessment modules into a largely unmodified and untrusted embedded Operating System (OS) without using techniques such as virtualisation and hypervisors, which would incur unacceptable performance overheads for many embedded applications.

Sancus targets low-cost embedded systems which have no virtual memory. Recent research on Program Counter Based Access Control (PCBAC) [30] shows

that, in this context, the value of the program counter can be used unambiguously to identify a specific software module. Whenever the program counter is within the address range associated with the module's code, the module is said to be executing. Memory isolation can then be implemented by configuring access rights to memory locations based on the current value of the program counter.

Sancus also provides attestation by means of built-in cryptographic primitives to provide assurance of the integrity and isolation of a given Protected Module (PM) to a third party. By using this feature, our trust assessment modules can be deployed dynamically, limiting memory consumption and restricting attacker adaptation. The module may then inspect the OS or application code and securely report trust metrics to an external trust management system.

Beyond trust assessment, our approach can be used to remotely test and debug code on a node and to facilitate the deployment of formally verified code in an untrusted context [1]. We describe and evaluate a prototypic implementation of our approach that integrates Sancus-protected trust assessment modules with the Contiki [9] OS, running on a Sancus-enabled TI MSP430 microcontroller, a single-address-space architecture with no memory management unit. The source code of the evaluation scenario is available at http://distrinet.cs.kuleuven.be/software/sancus/esorics15/.

2 Background

This section provides background information on the IoT and the Contiki OS, which enables extremely light-weight hardware such as TI MSP430 microcontrollers to be active components in the IoT. We emphasise on safety and security limitations of this setup and outline key features of the Sancus PMA as a way to cope with these limitations.

2.1 Contiki and the IoT

Contiki [9] is one of the most used OSs in the IoT. It is open source and designed for portability and to have a small memory footprint. Contiki readily runs on a range of different hardware platforms, including a number of small 8-bit and 16-bit microcontrollers, including the TI MSP430. On these machines, a Contiki system that supports full IPv6 networking can be deployed in less that 10 KiB of RAM and 30 KiB of ROM. While the IoT certainly requires the use of light-weight software on similarly light-weight, low-cost and low-power hardware, the use of this kind of configurations comes at the expense of safety and security. That is, microcontrollers such as the TI MSP430 do not feature the hierarchical protection domains, virtual memory and process isolation that are known as key mechanisms to implement safe and secure operation in the server and desktop world. Moreover, implementing computationally expensive cryptography and complex secure networking protocols is often contradictory with the constraints on power consumption and computation power on tiny autonomous devices. As a result, one would expect WSNs or the IoT in general to become a

safety hazard and a key target for attacks in the near future [26]. Recent attacks on Internet connected light bulbs [6] already give an outline of this future.

In particular the lack of protection domains on extremely light-weight hardware makes it very difficult to implement extensible systems securely since software components cannot easily be isolated from each other. In the remainder of this section we present the features of Sancus, a hardware-only TCB and PMAs that aims to mitigate this difficulty.

2.2 PMAs and Sancus

Sancus [23] guarantees strong isolation of software modules, which are generally referred to as Protected Modules or PM, through low-cost hardware extensions. Moreover, Sancus provides the means for remote parties to attest the state of, or communicate with, the isolated software modules.

Isolation. Like many PMAs [28], Sancus uses Program Counter Based Access Control (PCBAC) [30] to isolate PMs. Software modules are represented by a *public text section* containing the module's executable code and a *private data section* containing data that should be kept private to the module. The core of the PCBAC model is that the private data section of a module can only be accessed from code in its public text section. In other words, if and only if the program counter points to within a module's code section, memory access to this module's data section is allowed.

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

Attestation. Sancus allows external parties to verify the correct isolation of a module as well as to securely communicate with it. For this, Sancus extends the underlying MSP430 processor with a cryptographic core that includes symmetric

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

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

authenticated encryption and key derivation primitives. Sancus also defines a key hierarchy to ease the establishment of a shared symmetric key.

The root of this hierarchy is a *node master key*, K_N. This node-unique key is known only to the owner of the node and is not accessible by software. Each software provider that wants to deploy modules on a node gets assigned a unique ID, *SP*, by the node's owner. This ID is then used to derive a *software provider key*, $K_{N,SP}$, from K_N and the software provider is provided with this key along with its ID. The last level in the key hierarchy is the *software module key*, $K_{N,SP,SM}$. This key is derived from $K_{N,SP}$ using the *module identity SM*. The identity of a module is defined as the concatenation of the contents of its text section and the load addresses of its text and data sections.

When a module is isolated, the hardware will first derive $K_{N,SP}$ and then use that key to derive $K_{N,SP,SM}$. Sancus enforces that this key is only accessible by the newly isolated module. This construction ensures that (1) the key $K_{N,SP,SM}$ can *only* be used by a module with identity *SM* deployed by software provider *SP* on node N; and (2) isolation has been enabled for this module.

Since the software provider has access to $K_{N,SP}$, it can also calculate $K_{N,SP,SM}$. The latter key can then be used as the basis for attestation and secure communication. Indeed, because of the properties listed above, if the software provider receives a message created with $K_{N,SP,SM}$, it will have strong guarantees that this message was created by a module with identity *SM* isolated on node N.

3 Trust Assessment Modules

Our approach to trust assessment is designed to integrate seamlessly with the deployment of low-cost and low-power hardware in WSNs and in the IoT. In particular, we make use of a Sancus-enabled CPU to run a protected trust assessment module and to facilitate secure and authenticated communication with a remote operator of this module. This operator can be, for example, a human operator with a particular interest in inspecting a specific device, or a trust management system that keeps track of the integrity and trustworthiness of a larger network of devices. Our trust assessment module executes as a PM, in isolation from a base of largely unmodified and generally untrusted OS and application code. Yet, our approach partially relies on services provided by this untrusted code, e.g. networking, scheduling and memory management, in a way such that failure is detected by the trust assessment module or by the remote operator. Trust assessment modules are capable of inspecting and modifying the state of the untrusted OS and applications autonomously or on request, giving the operator a trustworthy means of assessing the integrity of the software on a node and to take actions accordingly.

In this section we describe the process of deploying and communicating with Sancus-protected trust assessment modules and discuss inspection targets and trust metrics. We further outline weaknesses and attack scenarios to our approach. While the examples in this section are given with respect to the Contiki OS and its internals, we believe that our approach can be easily adapted to support other OSs in the domain of the IoT, such as TinyOS [18] or FreeRTOS [5].

3.1 Module Deployment

This section describes how the operator of a trust assessment module can deploy such a module on a Sancus-enabled computing node. We focus on getting assurance of correct deployment and on establishing a secure and authenticated communication channel with the module. The principal deployment process is originally described in [23], where details on the underlying cryptographic machinery are given. Figure 1 illustrates the process and highlights the TCB. In summary, each Sancus-enabled computing *node* N possesses a unique *node master key*, K_N, which is managed by the hardware, not directly accessible by the software running on the node, and shared only with the Infrastructure Provider (IP). It is the responsibility of the IP to manage the hardware and deployment of the nodes, and to derive a *software provider key*, $K_{N,SP}$, for each party that is to install PMs on the node N. We refer to these parties as Software Provider (SPs); they are identified by a unique public ID SP. $K_{N,SP}$ is computed using a key derivation function that takes K_N and SP as input. The computing node includes a hardware implementation of this derivation function so as to independently compute $K_{N,SP}$. Thus, $K_{N,SP}$ is shared between the IP, an SP and a specific node N.

The SP, in our scenario equivalent with the operator or Trust Management System, may now deploy a trust assessment module on N. This module can be sent as a binary program over an untrusted network and be loaded by an untrusted loader on the node. Each software module has a unique identity SM, which comprises of the module's text section (code) and the effective start and end-addresses of the

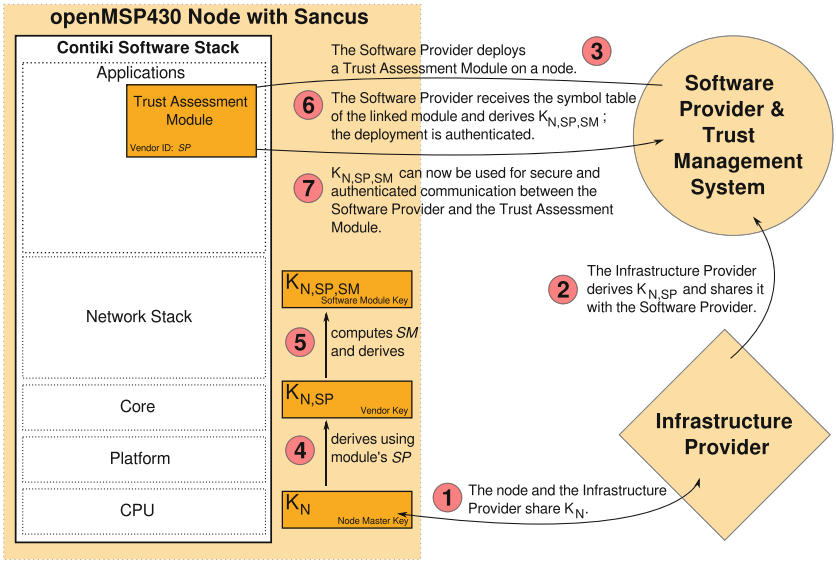

Fig. 1. Deployment of a trust assessment module on a Sancus node. The TCB, from the perspective of the operator, is shaded in orange.

loaded module's text- and protected data sections. As the module is loaded, the Sancus-enabled hardware computes a secret *software module key* $K_{N,SP,SM}$, which is derived from $K_{N,SP}$ and SM, and stored in hardware. Software cannot access $K_{N,SP,SM}$ directly but may use it to encrypt or decrypt data. The SP may derive $K_{N,SP,SM}$ if he is provided with a symbol table of the linked module, containing the start and end-addresses of the module's text- and protected data sections and the effective addresses of library code used by the module. This layout information is not confidential and may be transferred by the module loader back to the SP without integrity protection.[1] As can be seen, the proceeding outlined above establishes a shared secret between the SP and the correctly deployed PM SM on node N. All further data exchanged between these two parties can be encrypted with $K_{N,SP,SM}$, providing a secure and authenticated channel. Nonces may be used to guarantee freshness of messages.

The trust assessment module SM is now ready to execute on the computing node and may access all data and code on that node, with the exception of data belonging to other PMs. Consequentially, the module may inspect arbitrary address ranges and report its findings to the operator as an indication of the trustworthiness of the node. In the following section we discuss a number of these trust indicators in detail.

3.2 Trust Indicators

Our approach to trust assessment readily supports measuring a number of trust indicators as listed and explained in detail below. Importantly, our system is not limited to these indicators and we believe that additional or alternative indicators may be more suitable for specific application scenarios. Research, in particular in the context of software aging and software rejuvenation [7] names many such indicators that may be securely measured using our approach.

Code Integrity. A particularly useful measurement is code integrity. Sancus-enabled hardware features a keyed cryptographic hash function to compute a Message Authentication Code (MAC) of a section of memory using the module's secret key. This MAC may then either be reported to the remote operator or be compared with a MAC stored in the secret section of the trust assessment module in autonomous operation. Code integrity checks with a MAC are used by the trust assessment module to establish whether a particular section of code has been modified, which is then securely communicated to the operator. Unexpected code modifications may be caused by an attack against the device or by a malfunction. Candidates for integrity checks are core functions of the OS such as the scheduler, the memory management system or the network stack, or application code. MACing all code sections is technically feasible but may impose unacceptable computational overheads.

[1] It is possible for an attacker to modify the module or layout information during loading. However, this will be detected as soon as SM communicates with SP. Successful communication attests that SM has not been compromised during deployment to N and that the hardware protection has been correctly activated.

OS Data Structures. Trust assessment modules are further capable of inspecting and reporting the content of internal data structures of the OS. Interesting candidates for this are the process table or the interrupt vector table. Similar to code integrity checks, unexpected changes of these data structures are a strong indication of a malfunction or a successful attack against a device.

Available Resources. A group of indicators that is heavily used in the domain of software aging is the availability of resources such as memory and swap space: as software runs for extended periods of time, small memory leaks can accumulate and degrade performance, eventually leading to failure. In the context of Contiki and the MSP430 we use the general availability of program memory and data memory and the size of the largest available chunks of these as trust indicators. The chunk size is an important characteristic as our architecture does not feature a Memory Management Unit that could mitigate the fragmenting effect of repeated allocation and deallocation. Importantly, reliably measuring the availability of program and data memory requires implementing part of the allocator, typically a OS component, as part of the trust assessment TCB.

Application Data Structures. Similar to monitoring OS data structures, we have experimented with using application data as trust indicators. For example, on WSN nodes that run a webserver, activity can be measured by monitoring the length of the request queue. Also static content that is used to compile dynamic websites can be inspected to detect modification due to a bug or a malicious attempt. Generally all these measures are highly specific with respect to critical use cases of a node.

Event Occurrence and Timing. A key feature of our trust assessment infrastructure is to monitor and attest intentional activity on a node. More specifically, by integrating part of the OS's scheduler into the TCB, our approach can attest when critical code on a node has been executed. This allows an operator to infer which parts of a node are behaving within expected parameters.

Combined Indicators. In particular in the context of autonomous operation of a trust assessment module, combining trust indicators is desired so as to automatically adapt to changing deployment scenarios. In particular, we have experimented with modules that combine the inspection of OS data structures, i.e. the process table, and periodically performing integrity checks on the functions associated with each process. This can be interleaved with measuring the frequency of process invocation and execution times, giving the operator a detailed picture of the behaviour of a computing node and allowing for specific autonomous responses to faults.

3.3 Fault Recovery

As a trust assessment module or the operator detect anomalies on a node, the module is even capable of responding to the situation. Responses may range from a

simple reset of the node over a more thorough investigation of the fault up to actively manipulating the system state and restoring damaged code and data.

4 Evaluation

We have implemented the approach described in the previous section as a number of flexibly configurable trust assessment modules that can be loaded into a Contiki OS at runtime. In this section we evaluate our implementation with respect to overheads in terms of module sizes and runtime. We further discuss security gains, attack scenarios and their mitigation.

4.1 Scenario and Implementation

Our prototypic implementation is based on a developmental version of Contiki 3.x running on a Sancus-enabled openMSP430 [15,23]. We evaluate an application scenario in which the trust assessment module regularly reports on the application processes running on a node, periodically checks the integrity of a number of code sections of these processes and integrates with Contiki's scheduler to detect and log process invocations. We have further added a public entry point to the trust assessment module that allows an application to register invariant address ranges, which are then included in periodic integrity checks. This section gives an overview of entry points and the internal behaviour of our trust assessment module and the demo scenario.

As outlined in Table 2, our example module provides a number of entry points to be called from unprotected code. Most importantly, the `TAMainFunc` is invoked by the scheduler. In a first run, it will initialise internal data structures of the module and then populate these data structures with initial measurements from the unprotected OS. This involves shadowing part of the scheduler's process list and MACing the process functions and the interrupt vector table. Subsequent

Table 2. *Entry Points* of our trust assessment module.

Function Name	Description
`TAMainFunc`	Main entry point controlling initialisation and periodic behaviour
`TARegisterInvar`	Can be used by application code and internally to register an address range for regular integrity checks
`TASecureCallProcess`	Used by the OS scheduler to invoke application functions; the trust assessment module extends the call with counting the number of invocations and measuring time
`TAInvarsStatus`	Returns an encrypted status report on the integrity checked address ranges
`TAProcessStatus`	Returns an encrypted status report on the processes currently running on a node

Table 3. Size and execution time of different trust assessment components on an MSP430 running at 20 MHz: 1 cycle corresponds to 50 ns. Function sizes include protected helper functions.

Function	Size in Bytes	Runtime in Cycles	Description
`TACoreEnable`	58	236,440	Enables module protection and initiates key generation
`TAMainFunc`	430	578	Main function, initialisation
		73,678	... validation run (5 processes, 9 integrity checks)
`TARegisterInvar`	402	1,242	Stores meta-data and MACs of 32 B
		10,762	... 199 B
		19,930	... 399 B
`TACheckInvars`	498	69,659	Checks integrity of 9 address ranges (1833 B)
`TAAddProcess`	568	$\leq$ 18,374	Shadows and entry from the process list and determines length of process function
`TACheckProcesses`	288	2,371	Checks shadowed process data against process list (5 processes)
`TASecureCallProcess`	392	266	Process invocation with no logging
		$\leq$ 731	... logs time and number of invocations
`TAInvarsStatus`	202	10,254	Encrypts meta-data on integrity-checked code and data (160 B + 16 B nonce)
`TAProcessStatus`	202	17,488	Encrypts meta-data on running processes (320 B + 16 B nonce)
total	3,742	n/a	Code (`.text`) and data (part of `.bss`)

invocations result in the current state of the unprotected OS being compared with the internal state of the module. In addition, `TASecureCallProcess` is used by the scheduler to start process functions. As this function is part of the trust assessment module, it can securely log which process is invoked and keep track of meta data. Of course, all data, including MACs and meta data on process invocations is stored in the trust assessment module's private data section. The functions `TAInvarsStatus` and `TAProcessStatus` return a snapshot of this data, encrypted with the module's $K_{N,SP,SM}$ and using a nonce to guarantee freshness. Thus, the module's state can be reported to the operator for further assessment.

To test the effectiveness of our trust assessment module, our scenario integrates a number of trivial application processes and a "malicious" process that aims to perform alterations to OS data and application code. Specifically, our *attacker* is invoked by an event timer. With every invocation it performs one of the following random actions: do nothing, modify a function pointer in the process list, remove an entry from the process list, overwrite a process function, or modify an entry in the interrupt vector. Event timing and the Contiki's scheduler typically result in alternating invocation of the attacker and the trust assessment module. Expectedly, all changes performed by the attacker are detected with the next invocation of the trust assessment module.

4.2 Overheads

Our evaluation shows at what expenses the alterations made by the attacker are detected. In Table 3 we list measurements of the size of our trust assessment

components and these components' execution time. All code of the demo scenario is compiled either with MSP430-GCC 4.6.3[2] if no Sancus features are involved, or with the LLVM-based Sancus toolchain[3]. The trust assessment module is executed on an MSP430 configured with 41 KiB of program memory[4] and 16 KiB of data memory, running at 20 MHz. In Table 3 we report execution times in terms of CPU cycles. With the given clock speed, 1 cycle corresponds to 50 ns and 10,000 cycles correspond to 0.5 ms. For our evaluation, the MSP430 CPU is programmed on a Xilinx Spartan-6 FPGA. This renders a precise assessment of overheads in terms of power consumption infeasible. For a discussion of the power consumption of the Sancus extensions we refer the reader to [23].

As can be seen from Table 3, our approach does imply non-negligible overheads. Whether these overheads are acceptable depends largely on the constraints on reactivity and energy consumption versus safety and security requirements in a specific deployment scenario. Our trust assessment module is designed to keep the cost of periodic validation tasks small, typically below 70,000 cycles (3.5 ms), at the expense of incurring higher initial overheads. Overall, most overheads are caused by the use of Sancus-provided cryptographic operations. The performance and security provided by these operations is evaluated in [23].

As mentioned in Sect. 3.2 certain trust indicators, such as logging process invocations, required us to modify the Contiki core. These modifications are always very small, i.e., replacing a call to a Contiki internal function with a PM-equivalent. Yet, the resulting overhead is considerably high due to switching protection domains – 26 cycles for an unprotected call and return versus 160 cycles for calling a protected entry point function. Due to passing arguments, return values, and logging the function invocation with a time stamp, process invocations through TASecureCallProcess incurs an overhead of 731 cycles.

With respect to runtime performance it is important to mention that Sancus does not support interruption of protected code execution. Thus, protected modules run with interrupts disabled, which may lead to important interrupts not being served by the OS and certain properties of the unprotected code potentially being broken. Examples for this could be real-time deadlines not being met due to extensive integrity checks. This issue can be mitigated by splitting up periodic validation tasks, e.g., do not perform all integrity checks but only one per scheduled invocation of the trust assessment module. Similar approaches have been used to perform expensive validation tasks in desktop and server environments [14]. Ongoing research aims to resolve this issue by making Sancus PMs fully interruptible and re-entrant. Mechanisms for securely handling interrupts in the context of PMAs have been discussed in [8,17]. The non-interruptibility of Sancus PMs also makes it necessary to use trampoline functions that re-enable and again disable interrupts when transferring control to an application

[2] http://www.ti.com/tool/msp430-gcc-opensource/.
[3] http://distrinet.cs.kuleuven.be/software/sancus/.
[4] ROM is often used as program memory in embedded devices. On platforms that support module deployment at runtime, as we do, program memory is writable.

process in `TASecureCallProcess`, incurring relatively high overheads for scheduled process invocations.

We do neither evaluate nor provide an integration with a trust management system. In particular, we do not evaluate the infrastructure that has to be in place to load a software module at runtime, and to communicate with a PM on the OS level. This infrastructure performs fairly generic tasks, yet its implementation is highly dependent on the deployment scenario. Contiki and many other embedded OSs provide module loaders and a network stack that is fully sufficient to implement the required functionality. Yet, the performance of these components depend on the storage and communication hardware connected to the MSP430 and is, thus, beyond the scope of this paper.

4.3 Security Evaluation

Bootstrapping Autonomous Operation. An obvious issue of the scenario presented and evaluated here is with respect to the suggested autonomous mode of operation: the trust assessment module automatically discovers running processes and then periodically checks the discovered data structures and code sections for unexpected changes. Of course, an attacker may tamper with these sections at or before boot time, effectively preventing detection in regular checks. In our scenario it would be the responsibility of the operator to request and evaluate the output of `TAInvarsStatus` and `TAProcessStatus` to detect such modifications. Alternatively, a trust assessment module may also be provided with a list of expected processes and MACs by the operator at runtime, using encrypted communication.

Communication Failure. While the code and the internal state of the PM cannot be tampered with, it is of course possible that malfunctions or an successful attack against the node prevent the trust assessment module from successfully communication with the operator or from executing altogether. Yet, this is detected by the operator who then may conduct actions appropriate for the deployment scenario.

Preventing Invocation of the Trust Assessment Module. In the evaluated application scenario, the trust assessment module is invoked by the scheduler and its entry point is stored in the unprotected process list. This gives the attacker process the opportunity to tamper with the pointer to the entry point, allowing it to disable execution of the trust assessment module. Alternatively, an attacker or a malfunction may disable interrupts while preventing control flow from returning to the scheduler. In our evaluation scenario these attack would not be detected by the trust assessment module directly but rather by the operator who would not be able to communicate with the module.

For autonomous operation, this attack can be easily mitigated by configuring the trust assessment module to be invoked as an interrupt service routine for a non-maskable timed interrupt. We can simulate this behaviour by using the watchdog as a source of timed interrupts, which we have implemented as an

optional configuration option in our evaluation scenario. To ensure that the module will complete its tasks, this approach requires the worst-case execution time of the trust assessment module to be smaller than the interrupt rate. It is possible to guarantee that the watchdog configuration is not modified by an attacker by making the control register and the respective entry in the interrupt vector table part of the secret section of a PM. In combination with extensive integrity checks, this approach also hinders stealthy attacks where malicious code would attempt to restore a valid system state before the trust assessment module is executing. Yet, using a non-maskable interrupt to invoke the trust assessment infrastructure requires some consideration: It must be possible to determine the worst-case execution time of the trust assessment module and it must be acceptable to interrupt application code for that time as the PM itself is non-interruptible. Using a scheduler to invoke the trust assessment module allows for more permissible policies that prevent starvation of applications.

Attacker Adaptation. As mentioned in the previous paragraph, a stealthy attacker that is well adapted to a specific trust assessment module may be able to hide code or data in address ranges that are not inspected by the module. The attacker may also restore inspected memory content to the state that is expected by the trust assessment module right before inspection takes place. Our approach to trust assessment counters these attacks by allowing the operator to deploy trust assessment modules at runtime, confronting the attacker with an unknown situation. Alternatively, a generic module may inspect targets by request from the operator rather than controlled by a deterministic built-in policy.

Process Accounting. Our trust assessment module features logging and reporting a time stamp of the latest invocation and the total number of invocations of scheduled processes. Of course, these numbers are only exact as long as processes are called through the scheduler, which passes the call through our trust assessment module. As processes may be invoked without using the scheduler, the numbers reported by our module represent a lower bound on the actual number of invocations. If more precise measures are needed for a particular process, this process should be implemented as a PM and perform its own accounting.

Extending the TCB. Of course, the safety and security of a node could be improved greatly by implementing larger parts of the OS, e.g. the scheduler, or applications as PMs. PMAs imply a number of complications that are a direct consequence of the strong isolation guarantees provided: resource sharing between components is generally prohibited, yet it is often desired for efficiently implementing communication between components. In Sancus, for example, one would have to explicitly copy the protected state of a module so as to share it with another module or unprotected code. While technically feasible, we believe that it is not trivial to re-implement a more complex code base as a set of neatly separated PMs. Alternatively one could think of compiling an entire embedded OS together with its applications as a single PM. This would ensure integrity but

does not provide for isolation between components and severely restricts runtime extensibility and the use of dynamic memory. Thus, the trust assessment modules provided here present a pragmatic approach to measure and improve safety and security of an IoT node while not interfering with the existing code on that node. This results in low development overheads and runtime overheads that should be acceptable for many deployment scenarios.

5 Related Work

This section discusses some related work in the domains of WSNs, trust assessment on high-end systems, and PMAs. Where applicable, we compare the work with our contributions. Note that this section is not meant as an exhaustive exposition on trust assessment – a domain that can be interpreted rather broadly – but as an overview of the work that we consider closely related to ours.

5.1 Trust Management in Wireless Sensor Network

Many schemes for trust management in WSNs have been devised by researcher over the years [12,16,19]. Most of these schemes deal with the problem of distributing trust management over a network of sensor nodes. Individual nodes usually obtain trust values about neighboring nodes by observing their externally visible behavior. These trust values are then propagated through the network allowing nodes to make decisions based on the trustworthiness of other nodes.

Although our approach does not deal directly with distributed networks, it can be used to enhance trust metrics used by existing trust management systems. Indeed, our trust assessment modules can provide nodes with a detailed view on the internal state of their neighbors; allowing them to make better informed decisions about their trustworthiness. Moreover, since the produced trust metrics are attested, the bar is significantly raised for existing attacks on trust management systems where malicious nodes try to impersonate good nodes.

5.2 Trust Assessment on Desktop and Server Systems

Techniques similar to our trust assessment modules have been described for the domain of desktop and server systems. Copilot [24] and Gibraltar [4] employ specialised PCI hardware to access OS kernel memory with negligible runtime overhead. Both systems detect and report modifications to kernel code and data.

A number of approaches use virtualisation extensions of modern general purpose CPU. Here, a hypervisor is employed to inspect a guest operating system. SecVisor [27] protects legacy OSs by ensuring that only validated code can be executed in kernel mode. Similarly, NICKLE [25], shadows physical memory in a hypervisor to store authenticated guest code. At runtime, kernel mode instructions are then only loaded from shadow memory and an attempt to execute code that is not shadowed is reported as an attack. Hello rootKitty [14] inspects guest memory from a hypervisor to detect and restore maliciously modified kernel data

structures. Due to frequent transitions between execution hypervisor and guest code, and expensive address translation between those domains, these inspection systems typically incur significant performance overheads. HyperForce [13] mitigates this problem by securely injecting the inspection code into the guest and forcing guest control flow to execute this code.

Our approach to trust assessment using PMs on a Sancus-enabled TI MSP430 provides isolation guarantees that are equivalent to executing the trust assessment code in a hypervisor. Yet, our PM executes in the same address space as the OSs and application, which makes expensive domain switches and address mapping unnecessary. In addition, Sancus provides attestation features in hardware that the above systems do not employ. On modern desktop architectures, these features can be implemented using the Trusted Platform Module.

Sancus enables the implementation of effective security mechanisms on extremely light-weight and low-power hardware. In terms of inspection abilities and isolation guarantees, these mechanisms are similar to the state-of-the-art in the desktop and server domain. Our approach to trust assessment modules illustrates that, using Sancus, comprehensive inspection mechanisms can be implemented efficiently, incurring runtime overheads that should be acceptable in many deployment scenarios with stringent safety and security requirements.

5.3 Alternatives to Sancus

The trust assessment infrastructure proposed in this paper is built upon Sancus [23], a PMA [28]. A number of PMAs have been proposed and can be used to implement our approach, as long as memory isolation and attestation features are provided, which we discuss below.

A PMA is typically employed as a core component of a TCB. The key feature of all PMA is to provide *memory isolation* for software components. That is, to enable the execution of a security sensitive component, a PM, so that access to the component's runtime state is limited to the TCB, the component itself, and if supported, to other modules specifically chosen by the protected component. In addition, execution of the module's code is guaranteed to happen in a controlled way so as to prevent code misuse attacks [2]: a module may specify a public Application Programming Interface (API) to be used by other modules. A range of PMAs for general purpose CPU has been presented in the last years, including Intel SGX [21], ARM TrustZone [3], TrustVisor [20] and Fides [29].

Recent research [10,17] has brought PMA techniques to small embedded microprocessors at an acceptable cost. PMAs such as SMART [10], TrustLite [17] as well as Sancus [23] utilise the PCBAC [30] approach to provide isolation an thereby guarantee the integrity of software modules executing on low-power embedded processors.

A second crucial feature of many trusted computing platforms is the ability to provide assurance of the integrity and isolation of a given PM to a third party. This party can be, e.g. another module on the same hosts or a software component on a remote host. We refer to the process of providing this assurance as *attestation*,

which is typically implemented by means of cryptographic primitives that operate on the PM's identity. Attestation can be reused to establish a shared secret for secure communication between the PM and a third party. To the best of our knowledge, only Intel SGX, SMART and Sancus readily implement attestation.

6 Conclusions

In this paper we present an approach to trust assessment for extremely light-weight and low-power computing nodes as they are often used in the Internet of Things (IoT). Instead of relying on the externally observable behaviour of a node, we deploy flexible trust assessment modules directly on the node. These modules are executing in isolation from an unprotected OS and application code. Yet, the modules are capable of inspecting the unprotected domain and report measurements that are indicative for the trustworthiness of a node to a trust management system. We employ Sancus [23] to guarantee isolation, to facilitate remote attestation of the correct deployment of a trust assessment module, and to secure communication between a module and a trust management system. Sancus is a Protected Module Architecture as well as a minimal hardware-only Trusted Computing Base. In terms of inspection abilities and isolation guarantees, Sancus-protected trust assessment modules are similar to using virtualisation technology or specialised security hardware in the desktop and server domain.

We have implemented our approach to trust assessment modules on a Sancus-enabled TI MSP430 microcontroller. Our results demonstrate that, using Sancus, comprehensive inspection mechanisms can be implemented efficiently, incurring runtime overheads that should be acceptable in many deployment scenarios with stringent requirements with respect to safety and security. Indeed, we believe that our approach enables many state-of-the-art inspection mechanisms and countermeasures against attacks to be adapted for IoT nodes and in the domain of Wireless Sensor Networks, which are in dire need of modern security mechanisms [26]. These mechanisms include integrity checks and data structure inspection as discussed in this paper. Yet, more complex mechanisms such as automatic invariant detection and validation [14], stack inspection [11] or protection against heap overflows [22] are in scope for our approach.

In the future we aim to improve performance and scalability of the inspection and reporting process by making Sancus modules fully interruptible and re-entrant. We are further interested in investigating alternative trust indicators and fault recovery mechanisms, and integrate our trust assessment modules with a trust management system. Finally we will investigate the deployment of formally verified code in an untrusted context [1] for Sancus, which can lead to proving the absence of runtime errors for Sancus-protected security critical code that runs on an IoT node.

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

References

1. Agten, P., Jacobs, B., Piessens, F.: Sound modular verification of c code executing in an unverified context. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, pp. 581–594. ACM (2015)
2. Agten, P., Strackx, R., Jacobs, B., Piessens, F.: Secure compilation to modern processors. In: 2012 IEEE 25th Computer Security Foundations Symposium (CSF 2012), pp. 171–185. IEEE, August 2012
3. Alves, T., Felton, D.: Trustzone: integrated hardware and software security. ARM white paper **3**(4), 18–24 (2004)
4. Baliga, A., Ganapathy, V., Iftode, L.: Detecting kernel-level rootkits using data structure invariants. IEEE Trans. Dependable Secure Comput. **8**(5), 670–684 (2011)
5. Barry, R.: FreeRTOS: A portable, open source, mini real time kernel (2010). http://www.freertos.org/
6. Chapman, A.: Hacking into internet connected light bulbs (2014). http://www.contextis.com/resources/blog/hacking-internet-connected-light-bulbs/
7. Cotroneo, D., Natella, R., Pietrantuono, R., Russo, S.: A survey of software aging and rejuvenation studies. J. Emerg. Technol. Comput. Syst. **10**(1), 8:1–8:34 (2014)
8. de Clercq, R., Piessens, F., Schellekens, D., Verbauwhede, I.: Secure interrupts on low-end microcontrollers. In: 2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 147–152. IEEE (2014)
9. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: 29th Annual IEEE International Conference on Local Computer Networks, pp. 455–462 (2004). http://www.contiki-os.org/
10. Eldefrawy, K., Francillon, A., Perito, D., Tsudik, G.: SMART: secure and minimal architecture for (establishing a dynamic) root of trust. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, USA (2012)
11. Feng, H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: 2003 Symposium on Security and Privacy, pp. 62–75. USENIX Association (2003)
12. Fernandez-Gago, M., Roman, R., Lopez, J. : A survey on the applicability of trust management systems for wireless sensor networks. In: Third International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing, SECPerU 2007, pp. 25–30 (2007)
13. Gadaleta, F., Nikiforakis, N., Mühlberg, J.T., Joosen, W.: HyperForce: hypervisor-enforced execution of security-critical code. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 126–137. Springer, Heidelberg (2012)
14. Gadaleta, F., Nikiforakis, N., Younan, Y., Joosen, W.: Hello rootKitty: a lightweight invariance-enforcing framework. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011. LNCS, vol. 7001, pp. 213–228. Springer, Heidelberg (2011)
15. Girard, O.: openMSP430 (2009). http://opencores.org
16. Granjal, J., Monteiro, E., Silva, J.S.: Security in the integration of low-power wireless sensor networks with the internet: a survey. Ad Hoc Netw. **24**(Part A), 264–287 (2015)
17. Koeberl, P., Schulz, S., Sadeghi, A.-R., Varadharajan, V.: Trustlite: a security architecture for tiny embedded devices. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys 2014, pp. 10:1–10:14. ACM (2014)

18. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: Tinyos: an operating system for sensor networks. In: Weber, W., Rabaey, J.M., Aarts, E. (eds.) Ambient Intelligence, pp. 115–148. Springer, Heidelberg (2005)
19. Lopez, J., Roman, R., Agudo, I., Fernandez-Gago, C.: Trust management systems for wireless sensor networks: best practices. Comput. Commun. **33**(9), 1086–1093 (2010)
20. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: Trustvisor: efficient tcb reduction and attestation. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 143–158. IEEE (2010)
21. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP 2013, pp. 10:1–10:1. ACM (2013)
22. Nikiforakis, N., Piessens, F., Joosen, W.: HeapSentry: kernel-assisted protection against heap overflows. In: Rieck, K., Stewin, P., Seifert, J.-P. (eds.) DIMVA 2013. LNCS, vol. 7967, pp. 177–196. Springer, Heidelberg (2013)
23. Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I., Piessens, F.: Sancus: low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In: Proceedings of the 22nd USENIX Conference on Security, SEC 2013, pp. 479–494. USENIX Association (2013)
24. Petroni Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot-a coprocessor-based kernel runtime integrity monitor. In: USENIX Security Symposium, pp. 179–194. USENIX Association (2004)
25. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
26. Roman, R., Najera, P., Lopez, J.: Securing the internet of things. Computer **44**(9), 51–58 (2011)
27. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 335–350. ACM (2007)
28. Strackx, R., Noorman, J., Verbauwhede, I., Preneel, B., Piessens, F.: Protected software module architectures. In: Reimer, H., Pohlmann, N., Schneider, W. (eds.) ISSE 2013 Securing Electronic Business Processes, pp. 241–251. Springer, Heidelberg (2013)
29. Strackx, R., Piessens, F.: Fides: selectively hardening software application components against kernel-level or process-level malware. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012, pp. 2–13. ACM (2012)
30. Strackx, R., Piessens, F., Preneel, B.: Efficient isolation of trusted subsystems in embedded systems. In: Jajodia, S., Zhou, J. (eds.) SecureComm 2010. LNICST, vol. 50, pp. 344–361. Springer, Heidelberg (2010)