Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base

Job Noorman Pieter Agten Wilfried Daniels Raoul Strackx
Anthony Van Herrewege Christophe Huygens Bart Preneel Ingrid Verbauwhede
Frank Piessens
iMinds-DistriNet and iMinds-COSIC, KU Leuven
{*Job.Noorman, Pieter.Agten, Wilfried.Daniels, Raoul.Strackx,*
Christophe.Huygens, Frank.Piessens}*@cs.kuleuven.be*
{*Anthony.VanHerrewege, Bart.Preneel, Ingrid.Verbauwhede*}*@esat.kuleuven.be*

Abstract

In this paper we propose Sancus, a security architecture for networked embedded devices. Sancus supports extensibility in the form of remote (even third-party) software installation on devices while maintaining strong security guarantees. More specifically, Sancus can remotely attest to a software provider that a specific software module is running uncompromised, and can authenticate messages from software modules to software providers. Software modules can securely maintain local state, and can securely interact with other software modules that they choose to trust. The most distinguishing feature of Sancus is that it achieves these security guarantees without trusting *any* infrastructural software on the device. The Trusted Computing Base (TCB) on the device is *only* the hardware. Moreover, the hardware cost of Sancus is low.

We describe the design of Sancus, and develop and evaluate a prototype FPGA implementation of a Sancus-enabled device. The prototype extends an MSP430 processor with hardware support for the memory access control and cryptographic functionality required to run Sancus. We also develop a C compiler that targets our device and that can compile standard C modules to Sancus protected software modules.

1 Introduction

Computing devices and software are omnipresent in our society, and society increasingly relies on the correct and secure functioning of these devices and software. Two important trends can be observed. First, network connectivity of devices keeps increasing. More and more (and smaller and smaller) devices get connected to the Internet or local ad-hoc networks. Second, more and more devices support extensibility of the software they run – often even by third parties different from the device manufacturer or device owner. These two factors are important because they enable a vast array of interesting applications, ranging from over-the-air updates on smart cards, over updateable implanted medical devices to programmable sensor networks. However, these two factors also have a significant impact on security threats. The combination of connectivity and software extensibility leads to malware threats. Researchers have already shown how to perform code injection attacks against embedded devices to build self-propagating worms [18, 19]. Viega and Thompson [45] describe several recent incidents and summarize the state of embedded device security as "a mess".

For high-end devices, such as servers or desktops, the problems of dealing with connectivity and software extensibility are relatively well-understood, and there is a rich body of knowledge built up from decades of research; we provide a brief survey in the related work section.

However, for low-end, resource-contrained devices, no effective low-cost solutions are known. Many embedded platforms lack the standard security features (such as privilege levels or advanced memory management units that support virtual memory) present in high-end processors. Depending on the overall system security goals, as well as the context in which the system must operate, there may be more optimal solutions than just porting the general-purpose security features from high-end processors. Several recent results show that researchers are exploring this idea in a variety of settings. For instance, El Defrawy et al. propose SMART, a simple and efficient hardware-software primitive to establish a dynamic root of trust in an embedded processor [14], and Strackx et al. propose a simple program-counter based memory access control system to isolate software components [43].

In this paper we build on these primitives to propose a security architecture that supports secure third-party software extensibility for a network of low-end processors (the prototypical example of such a network is a sensor network). The architecture enables mutually distrusting parties to run their software modules on the same nodes in the network, while each party maintains strong assurance

that its modules run untampered. This kind of secure software extensibility is very useful for applications of sensor networks, for instance in the logistics and medical domains. We discuss some application areas in more detail in Section 2.4.

The main distinguishing feature of our approach is that we achieve these security guarantees without *any* software in the TCB on the device, and with only minimal hardware extensions. Our attacker model assumes that an attacker has *complete* control over the software state of a device, and even for such attackers our security architecture ensures that any results a party receives from one of its modules can be validated to be genuine. Obviously, with such a strong attacker model, we can not guarantee availability, so an attacker can bring the system down, but *if* results are received their integrity and authenticity can be verified.

More specifically, we make the following contributions:

- We propose Sancus[1], a security architecture for resource-constrained, extensible networked embedded systems, that can provide remote attestation and strong integrity and authenticity guarantees with a minimal (hardware) TCB.
- We implement the hardware required for Sancus as an extension of a mainstream microprocessor, and we show that the cost of these hardware changes (in terms of performance, area and power) is small.
- We implement a C compiler that targets Sancus-enabled devices. Building software modules for Sancus can be done by putting some simple annotations on standard C files, showing that the cost in terms of software development is also low.

To guarantee the reproducibility and verifiability of our results, all our research materials, including the hardware design of the processor, and the C compiler are publicly available.

The remainder of this paper is structured as follows. First, in Section 2 we clarify the problem we address by defining our system model, attacker model and the security properties we aim for. The next two sections detail the design of Sancus and some interesting implementation aspects. Section 5 reports on our evaluation of Sancus and the final two sections discuss related work and conclude.

2 Problem statement

2.1 System model

We consider a setting where a single infrastructure provider, *IP*, owns and administers a (potentially large) set of microprocessor-based systems that we refer to as *nodes* N_i. A variety of third-party *software providers* SP_j are interested in using the infrastructure provided by *IP*. They do so by deploying *software modules* $SM_{j,k}$ on the nodes administered by *IP*. Figure 1 provides an overview.

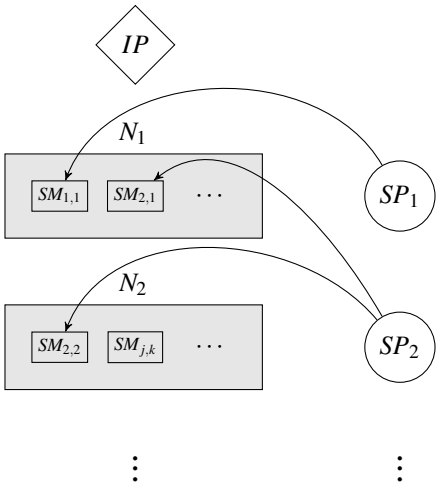

Figure 1: Overview of our system model. *IP* provides a number of nodes N_i on which software providers SP_j can deploy software modules $SM_{j,k}$.

This abstract setting is an adequate model for many ICT systems today, and the nodes in such systems can range from high-performance servers (for instance in a cloud system), over smart cards (for instance in GlobalPlatform-based systems) to tiny microprocessors (for instance in sensor networks). In this paper, we focus on the low end of this spectrum, where nodes contain only a small embedded processor.

Any system that supports extensibility (through installation of software modules) by several software providers must implement measures to make sure that the different modules can not interfere with each other in undesired ways (either because of bugs in the software, or because of malice). For high- to mid-end systems, this problem is relatively well-understood and good solutions exist. Two important classes of solutions are (1) the use of virtual memory, where each software module gets its own virtual address space, and where an operating system or hypervisor implements and guards communication channels between them (for instance shared memory sections or inter-process communication channels), and (2) the use of a memory-safe virtual machine (for instance a Java VM) where software modules are deployed in memory-safe bytecode and a security architecture in the VM guards the interactions between them.

For low-end systems with cheap microprocessors, providing adequate security measures for the setting sketched

[1] Sancus was the ancient Roman god of trust, honesty and oaths.

above is still an open problem, and an active area of research [16]. One straightforward solution is to transplant the higher-end solutions to these low-end systems: one can extend the processor with virtual memory, or one can implement a Java VM. This will be an appropriate solution in some contexts, but there are two important disadvantages. First, the cost (in terms of required resources such as chip surface, power or performance) is non-negligible. And second, these solutions all require the presence of a sizable trusted software layer (either the OS or hypervisor, or the VM implementation).

The problem we address in this paper is the design, implementation and evaluation of a novel security architecture for low-end systems that is inexpensive and that does not rely on any trusted software layer. The TCB on the networked device is *only* the hardware. More precisely, a software provider needs to trust only his own software modules; he does not need to trust any infrastructural or third-party software on the nodes, only the hardware of the infrastructure and his own modules.

2.2 Attacker model

We consider attackers with two powerful capabilities.

First, we assume attackers can manipulate *all* the software on the nodes. In particular, attackers can act as a software provider and can deploy malicious modules to nodes. Attackers can also tamper with the operating system (for instance because they can exploit a buffer overflow vulnerability in the operating system code), or even install a completely new operating system.

Second, we assume attackers can control the communication network that is used by *IP*, software providers and nodes to communicate with each other. Attackers can sniff the network, can modify traffic, or can mount man-in-the-middle attacks.

With respect to the cryptographic capabilities of the attacker, we follow the Dolev-Yao attacker model [11]: attackers can not break cryptographic primitives, but they can perform protocol-level attacks.

Finally, attacks against the hardware are out of scope. We assume the attacker does not have physical access to the hardware, can not place probes on the memory bus, can not disconnect components and so forth. While physical attacks are important, the addition of hardware-level protections is an orthogonal problem that is an active area of research in itself [2, 6, 25, 26]. The addition of hardware-level protection will be useful for many practical applications (in particular for sensor networks) but does not have any direct impact on our proposed architecture or on the results of this paper.

2.3 Security properties

For the system and attacker model described above, we want our security architecture to enforce the following security properties:

- *Software module isolation.* Software modules on a node run *isolated* in the sense that no software outside the module can read or write its runtime state, and no software outside the module can modify the module's code. The only way for other software on the node to interact with a module is by calling one of its designated entry points.
- *Remote attestation.* A software provider can verify with high assurance that a specific software module is loaded on a specific node of *IP*.
- *Secure communication.* A software provider can receive messages from a specific software module on a specific node with authenticity, integrity and freshness guarantees. For simplicity we do not consider confidentiality properties in this paper, but our approach could be extended to also provide confidentiality guarantees.
- *Secure linking.* A software module on a node can link to and call another module on the same node with high assurance that it is calling the intended module. The runtime interactions between a module *A* and a module *B* that *A* links to can not be observed or tampered with by other software on the same node.

Obviously, these security properties are not entirely independent of each other. For instance, it does not make sense to have secure communication but no isolation: given the power of our attackers, any message could then simply be modified right after its integrity was verified by a software module.

2.4 Application scenarios

This section illustrates some real-world application scenarios where the security properties above are relevant. Today's ICT environments involve many parties using shared resources. This is not different for the sensor space where applications have moved from the monolithic, often static, single application domain (such as wildlife [13] or volcano monitoring [46]) to a dynamic and long-lived setting characterized by platform-application decoupling [24] and resource sharing [33].

We present two illustrating scenarios. First, consider the logistics domain [48]. Given node cost and complexity, powerful nodes can be attached to containers, but nodes attached to packages are low-end and resource-constrained. The package is under control of the package

owner, the *IP*, a pharmaceutical company in this example scenario. This pharmaceutical wants a software module for continuous cold-chain visibility of the package.[2] In the warehouse, the shipping company wants to load a radio-location software module to expedite package processing. In the harbor, because of customs regulations like C-TPAT [44], the container owner needs to attest manifest validity and package integrity, requiring yet a different software module on the package node.

Another representative scenario is found in the medical domain, where a hospital is equipped with a variety of nodes used for many processes simultaneously, with most of those processes security sensitive [29]. Building nodes, for example, support facility management with software modules for Heating, Ventilation, and Air Conditioning (HVAC) or fire control and physical security, but are also used for patient tracking and monitoring of vital signals, an application where strong security requirements are present with respect to health information. The same nodes can even automate the supply chain by supporting asset and inventory management of medical goods through a localization and tracking software module.

The above scenarios establish a clear need for isolation, attestation, secure communication and secure linking of the various software modules reflecting the dynamic objectives of the various stakeholders. We believe these scenarios are strong evidence for the value of the Sancus architecture.

3 Design of Sancus

The main design challenge is to realize the desired security properties *without trusting any software on the nodes*, and under the constraint that nodes are low-end resource constrained devices. An important first design choice that follows from the resource constrained nature of nodes is that we limit cryptographic techniques to symmetric key. While public key cryptography would simplify key management, the cost of implementing public key cryptography in hardware is too high [31].

We present an overview of our design, and then we zoom in on the most interesting aspects.

3.1 Overview

Nodes. Nodes are low-cost, low-power microcontrollers (our implementation is based on the TI MSP430). The processor in the nodes uses a von Neumann architecture with a single address space for instructions and data. To distinguish actual nodes belonging to *IP* from fake nodes set up by an attacker, *IP* shares a symmetric key with each of its nodes. We call this key the *node master key*, and use the notation K_N for the node master key of node N. Given our attacker model where the attacker can control all software on the nodes, it follows that this key must be managed by the hardware, and it is only accessible to software in an indirect way.

Software Providers. Software providers are principals that can deploy software to the nodes of *IP*. Each software provider has a unique public ID *SP*.[3] *IP* uses a key derivation function *kdf* to compute a key $K_{N,SP} = kdf(K_N, SP)$, which *SP* will later use to setup secure communication with its modules. Since node N has key K_N, nodes can compute $K_{N,SP}$ for any *SP*. The node will include a hardware implementation of *kdf* so that the key can be computed without trusting any software.

Software Modules. Software modules are essentially simple binary files containing two mandatory sections: a *text section* containing protected code and constants and a *protected data section*. As we will see later, the contents of the latter section are not attested and are therefore vulnerable to malicious modification before hardware protection is enabled. Therefore, the processor will zero-initialize its contents at the time the protection is enabled to ensure an attacker can not have *any* influence on a module's initial state. Next to the two protected sections discussed above, a module can opt to load a number of *unprotected sections*. This is useful to, for example, limit the amount of code that can access protected data. Indeed, allowing code that does not need it access to protected data increases the possibility of bugs that could leak data outside of the module. In other words, this gives developers the opportunity to keep the trusted code of their own modules *as small as possible*. Each section has a header that specifies the start and end address of the section.

The *identity* of a software module consists of (1) the content of the text section and (2) the start and end addresses of the text and protected data sections. We refer to this second part of the identity as the *layout* of the module. It follows that two modules with the exact same code and data can coexist on the same node and will have different identities as their layout will be different. We will use notations such as *SM* or SM_1 to denote the identity of a specific software module.

Software modules are always loaded on a node on behalf of a specific software provider *SP*. The loading proceeds as expected, by loading each of the sections of the module in memory at the specified addresses. For each module loaded, the processor maintains the layout information in a *protected storage* area inaccessible from

[2] That is, the continuous monitoring of a temperature-controlled supply chain.

[3] Throughout this text, we will often refer to a software provider using its ID *SP*.

$$K_N = \text{Known by } IP$$
$$K_{N,SP} = kdf(K_N, SP)$$
$$K_{N,SP,SM} = kdf(K_{N,SP}, SM)$$

Figure 2: Overview of the keys used in Sancus. The node key K_N is only known by *IP* and the hardware. When *SP* is registered, it receives its key $K_{N,SP}$ from *IP* which can then be used to create module specific keys $K_{N,SP,SM}$.

software. It follows that the node can compute the identity of all modules loaded on the node: the layout information is present in protected storage and the content of the text section is in memory.

An important sidenote here is that the loading process is *not* trusted. It is possible for an attacker to intervene and modify the module during loading. However, this will be detected as soon as the module communicates with its provider or with other modules (see Section 3.3).

Finally, the node computes a symmetric key $K_{N,SP,SM}$ that is specific to the module *SM* loaded on node *N* by provider *SP*. It does so by first computing $K_{N,SP} = kdf(K_N, SP)$ as discussed above, and then computing $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$. All these keys are kept in the protected storage and will only be available to software indirectly by means of new processor instructions we discuss later. Figure 2 gives an overview of the keys used by Sancus.

Note that the provider *SP* can also compute the same key, since he received $K_{N,SP}$ from *IP* and since he knows the identity *SM* of the module he is loading on *N*. This key will be used to attest the presence of *SM* on *N* to *SP* and to protect the integrity of data sent from *SM* on *N* to *SP*.

Figure 3 shows a schematic picture of a node with a software module loaded. The picture also shows the keys and the layout information that the node has to manage.

Memory protection on the nodes. The various modules on a node must be protected from interfering with each other in undesired ways by means of some form of memory protection. We base our design on the recently proposed *program-counter based memory access control* [43], as this memory access control model has been shown to support strong isolation [42] as well as remote attestation [14]. Roughly speaking, isolation is implemented by restricting access to the protected data section of a module such that it is only accessible while the program counter is in the corresponding text section of the same module. Moreover, the processor instructions that use the keys $K_{N,SP,SM}$ will be program counter dependent. Essentially the processor offers a special instruction to compute a Message Authentication Code (MAC). If the instruction is invoked from within the text section of a specific module *SM*, the processor will use key $K_{N,SP,SM}$ to compute the MAC. Moreover, the instruction is only available after memory protection for module *SM* has been enabled. It follows that only a well-isolated *SM* installed on behalf of *SP* on *N* can compute MACs with $K_{N,SP,SM}$, and this is the basis for implementing both remote attestation and secure (integrity-protected) communication to *SP*.

Secure linking. A final aspect of our design is how we deal with secure linking. When a software provider sends a module SM_1 to a node, this module can specify that it wants to link to another module SM_2 on the same node, so that SM_1 can call services of SM_2 locally. SM_1 specifies this by including a MAC of (the identity of) SM_2 computed using the key K_{N,SP,SM_1} in an unprotected section.[4] The processor includes a new special instruction that SM_1 can call to check that (1) there is a module loaded (with memory protection enabled) at the address of SM_2 and (2) the MAC of the identity of that module has the expected value.

This initial authentication of SM_2 is needed only once. In Section 3.5, we will discuss a more efficient procedure for subsequent authentications.

We currently do not incorporate *caller authentication* in our design. That is, SM_2 can not easily verify that it has been called by SM_1. Note that this can in principle be implemented in software: SM_1 can call SM_2 providing a secret nonce as parameter. SM_2 can then call-back SM_1, passing the same nonce, asking for acknowledgement that it had indeed been called by SM_1. Future work will include caller authentication in the core of Sancus' design to make it more efficient and transparent.

Separating the various uses of MACs. Sancus uses MACs for a variety of integrity checks as well as for key derivation. Our design includes a countermeasure to avoid attacks where an attacker replays a MAC computed for one purpose in another context. In order to achieve separation between the different applications of MAC functions, we make sure the first byte of the input to the MAC function is different for each use case: `01` for the derivation of $K_{N,SP}$, `02` for the derivation of $K_{N,SP,SM}$, `03` for attestation and `04` for MAC computations on data.

Confidentiality. As mentioned in Section 2.3, we decided to not include confidentiality of communication in our design. However, since we provide attestation of modules and authentication of messages, confidentiality can

[4] Note that since this MAC depends on the load addresses of SM_1 and SM_2, it may not be known until SM_1 has been deployed. If this is the case, *SP* can simply send the MAC *after* SM_1 is deployed and the load addresses are known.

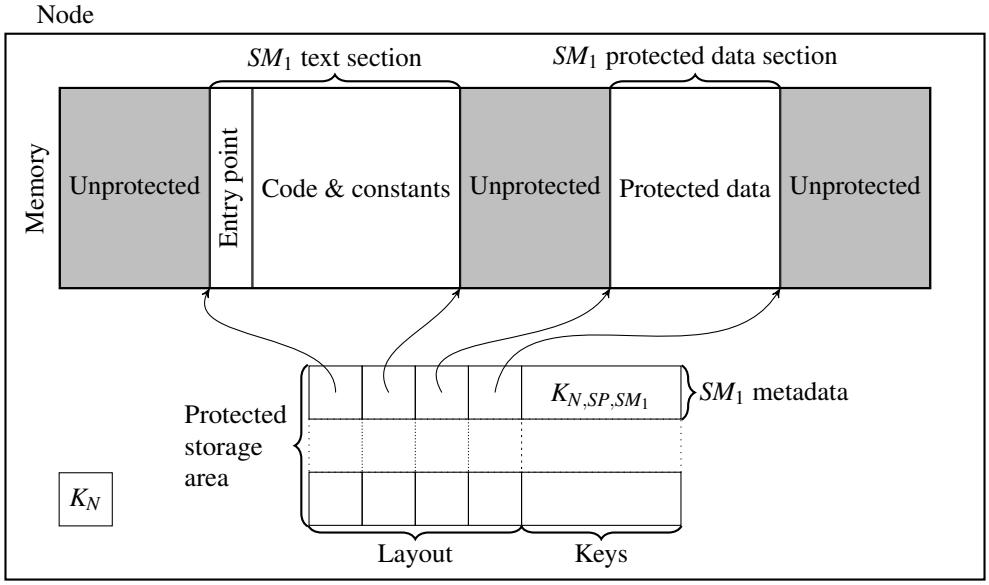

Figure 3: A node with a software module loaded. Sancus ensures the keys can never leave the protected storage area by only making them available to software in indirect ways through new processor instructions.

be implemented in software if necessary. One possibility is deploying a module with the public key of *SP* and a software implementation of the necessary cryptographic primitives. Another possibility is establishing a shared secret after deployment using a method such as Diffie-Hellman key exchange with authenticated messages. Note that implementing this last method is non-trivial due to the lack of a secure source of randomness. However, in the context of wireless sensor networks, methods have been devised to create cryptographically secure random number generators using only commonly available hardware [17].

Since the methods outlined above are expensive in terms of computation time and increase the TCB of modules, we are currently considering adding confidentiality to the core of Sancus' design. Exploring this is left as future work.

This completes the overview of our design. We now zoom in on the details of key management, memory access control, secure communication, remote attestation and secure linking.

3.2 Key management

We handle key management without relying on public-key cryptography [32]. *IP* is a trusted authority for key management. All keys are generated and/or known by *IP*. There are three types of keys in our design (Figure 2):

- Node master keys K_N shared between node N and *IP*.
- Software provider keys $K_{N,SP}$ shared between a provider *SP* and a node N.
- Software module keys $K_{N,SP,SM}$ shared between a node N and a provider *SP*, and the hardware of N makes sure that only *SM* can use this key.

We have considered various ways to manage these keys. A first design choice is how to generate the node master keys. We considered three options: (1) using the same node master key for every node, (2) randomly generating a separate key for every node using a secure random number generator and keeping a database of these keys at *IP*, and (3) deriving the master node keys from an *IP* master key using a key derivation function and the node identity N.

We discarded option (1) because for this choice the compromise of a single node master key breaks the security of the entire system. Options (2) and (3) are both reasonable designs that trade off the amount of secure storage and the amount of computation at *IP*'s site. Our prototype uses option (2).

The software provider keys $K_{N,SP}$ and software module keys $K_{N,SP,SM}$ are derived using a key derivation function as discussed in the overview section.

Finally, an important question is how compromised keys can be handled in our scheme. Since any secure key derivation function has the property that deriving the master key from the derived key is computationally infeasible, the compromise of neither a module key $K_{N,SP,SM}$ nor a provider key $K_{N,SP}$ needs to lead to the revocation of K_N. If $K_{N,SP}$ is compromised, provider *SP* should receive a

new name SP' since an attacker can easily derive $K_{N,SP,SM}$ for any SM given $K_{N,SP}$. If $K_{N,SP,SM}$ is compromised, the provider can still safely deploy other modules. SM can also still be deployed if the provider makes a change to the text section of SM.[5] If K_N is compromised, it needs to be revoked. Since K_N is different for every node, this means that only one node needs to be either replaced or have its key updated.

3.3 Memory access control

Memory can be divided into (1) memory belonging to modules, and (2) the rest, which we refer to as unprotected memory. Memory allocated to modules is divided into two sections, the *text* section, containing code and public constants, and the *protected data* section containing all the data that should remain confidential and should be integrity protected. Modules can also have an *unprotected data* section that is considered to be part of unprotected memory from the point of view of the memory access control system.

Apart from application-specific data, run-time metadata such as the module's call stack should typically also be included in the protected data section. Indeed, if a module's stack were to be shared with untrusted code, confidential data may leak through stack variables or control-data might be corrupted by an attacker. It is the module's responsibility to make sure that its call stack and other run-time metadata is in its protected data section, but our implementation comes with a compiler that ensures this automatically (see Section 4.2).

The memory access control logic in the processor enforces that (1) data in the protected data section of a module is only accessible while code in the text section of that module is being executed, and (2) the code in the text section can only be executed by jumping to a well-defined entry point. The second part is important since it prevents attackers from misusing code chunks in the text section to extract data from the protected data section. For example, without this guarantee, an attacker might be able to launch a Return-Oriented Programming (ROP) attack [7] by selectively combining gadgets found in the text section. Note that, as shown in Figure 3, our design allows modules to have a single entry point only. This may seem like a restriction but, as we will show in Section 4.2, it is not since multiple logical entry points can easily be dispatched through a single physical entry point. Table 1 gives an overview of the enforced access rights.

Memory access control for a module is enabled at the time the module is loaded. First, untrusted code (for instance the node operating system) will load the module in memory as discussed in Section 3.1. Then, a special instruction is issued:

`protect` *layout,SP*

This processor instruction has the following effects:

- the layout is checked not to overlap with existing modules, and a new module is registered by storing the layout information in the protected storage of the processor (see Section 3.1 and Figure 3);
- memory access control is enabled as discussed above; and
- the module key $K_{N,SP,SM}$ is created – using the text section and layout of the actually loaded module – and stored in the protected storage.

This explains why we do not need to trust the operating system that loads the module in memory: if the content of the text section, or the layout information would be modified before execution of the `protect` instruction, then the key generated for the module would be different and subsequent attestations or authentications performed by the module would fail. Once the `protect` instruction has succeeded, the hardware-implemented memory access control scheme ensures that software on the node can no longer tamper with SM.

The only way to lift the memory access control is by calling the processor instruction:

`unprotect`

The effect of this instruction is to lift the memory protection of the module *from which the* `unprotect` *instruction is called*. A module should only call `unprotect` after it has cleared the protected data section.

Finally, it remains to decide how to handle memory access violations. We opt for the simple design of resetting the processor and clearing memory on every reset. This has the advantage of clearly being secure for the security properties we aim for. However an important disadvantage is that it may have a negative impact on availability of the node: a bug in the software may cause

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.

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

[5] For example, a random byte could be appended to the text section without changing the semantics of the module.

the node to reset and clear its memory. An interesting avenue for future work is to come up with strategies to handle memory access violations in less severe ways. Invalid reads could return some default value as in secure multi-execution [10]. Invalid writes or jumps could be dropped or modified to actions that are allowed as in edit-automata [35]. For instance, an invalid memory read might just return zero, and an invalid jump might be redirected to an exception handler.

3.4 Remote attestation and secure communication

The module key $K_{N,SP,SM}$ is managed by the hardware of the node, and it can only be used by software in two ways. The first way is by means of the following processor instruction (we discuss the second way in Section 3.5):

`MAC-seal` *start address, length, result address*

This instruction can only be called from within the text section of a protected module, and the effect is that the processor will compute the MAC of the data in memory starting at *start address* and up to *start address* + *length* using the module key of the module performing the instruction. The resulting MAC value is written to *result address*.

Modules can use this processor instruction to protect the integrity of data they send to their provider. The data plus the corresponding MAC can be sent using the untrusted operating system over an untrusted network. If the MAC verifies correctly (using $K_{N,SP,SM}$) upon receipt by the provider *SP*, he can be sure that this data indeed comes from *SM* running on *N* on behalf of *SP* as the node's hardware makes sure only this specific module can compute MACs with the module key $K_{N,SP,SM}$.

To implement remote attestation, we only need to add a freshness guarantee (i.e. protect against replay attacks). Provider *SP* sends a fresh nonce *No* to the node *N*, and the module *SM* returns the MAC of this nonce using the key $K_{N,SP,SM}$, computed using the `MAC-seal` instruction. This gives the *SP* assurance that the correct module is running on that node at this point in time.

Building on this scheme, we can also implement secure communication. Whenever *SP* wants to receive data from *SM* on *N*, it sends a request to the node containing a nonce *No* and possibly some input data *I* that is to be provided to *SM*. This request is received by untrusted code on the node which passes *No* and *I* as arguments to the function of *SM* to be called. When *SM* has calculated the output *O*, it asks the processor to calculate a MAC of $No||I||O$ using the `MAC-seal` instruction. This MAC is then sent along with *O* to *SP*. By verifying the MAC with its own copy of the module key, the provider has strong assurance that *O* has been produced by *SM* on node *N* given input *I*.

3.5 Secure linking and local communication

In this section, we discuss how we assure the secure linking property mentioned in Section 2.3. More specifically, we consider the situation where a module SM_1 wants to call another module SM_2 and wants to be ensured that (1) the integrity of SM_2 has not been compromised, and (2) SM_2 is correctly protected by the processor. The second point is important, and is the reason why SM_1 can not just verify the integrity of the text section of SM_2 by itself. SM_1 will need help from the processor to give assurance that SM_2 is loaded with the expected layout and that protection for SM_2 is enabled.

In our design, if module SM_1 wants to link securely to SM_2, SM_1 should be deployed with a MAC of SM_2 created with the module key K_{N,SP,SM_1}. The processor provides a special instruction to check the existence and integrity of a module at a specified address:

`MAC-verify` *address*, *expected MAC*.

This instruction will:

- verify that a module is loaded (with protection enabled) at the provided address;
- compute the MAC of the identity of that module using the module key of the module calling this instruction;
- compare the resulting MAC with the *expected MAC* parameter of the instruction; and
- if the MACs were equal, return the module's ID (to be explained below), otherwise return zero.

This is the second (and final) way in which a module can use its module key (next to the `MAC-seal` instruction discussed in Section 3.4).

Using this processor instruction, a module can securely check for the presence of another expected module, and can then call that other module.

Since this authentication process is relatively expensive (it requires the computation of a MAC), our design also includes a more efficient mechanism for repeated authentication. The processor will assign sequential IDs[6] to modules that it loads, and will ensure that – within one boot cycle – it never reuses these IDs. A processor instruction:

`get-id` *address*

[6] To avoid confusion between the two different identity concepts used in this text, we will refer to the hardware-assigned number as *ID* while the text section and layout of a module is referred to as *identity*.

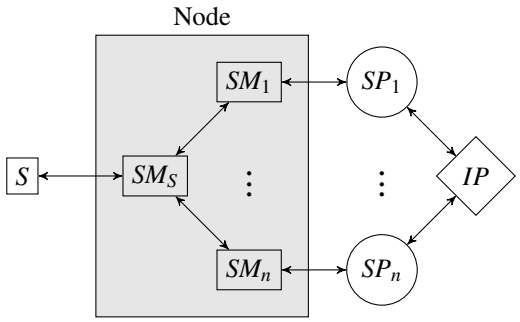

Figure 4: Setup of the sensor node example discussed in Section 3.6. Sancus ensures only module SM_S is allowed to directly communicate with the sensor S. Other modules securely link to SM_S to receive sensor data in a trusted way.

checks that a protected module is present at *address* and returns the ID of the module. Once a module has checked using the initial authentication method that the module at a given address is the expected module, it can remember the ID of that module, and then for subsequent authentications it suffices to check that the same module is still loaded at that address using the `get-id` instruction.

3.6 An end-to-end example

To make the discussion in the previous sections more concrete, this section gives a small example of how our design may be applied in the area of sensor networks. Figure 4 shows our example setup. It contains a single node to which a sensor S is attached; communication with S is done through memory-mapped I/O. The owner of the sensor network, *IP*, has deployed a special module on the processor, SM_S, that is in charge of communicating with S. By ensuring that the protected data section of SM_S contains the memory-mapped I/O region of S, *IP* ensures that no software outside of SM_S is allowed to configure or communicate directly with S; all requests to S need to go through SM_S.

Figure 4 also shows a number of software providers $(SP_1, \ldots, SP_n)$ who have each deployed a module $(SM_1, \ldots, SM_n)$. In the remainder of this section, we walk the reader through the life cycle of a module in this example setup.

The first step for a provider *SP* is to contact *IP* and request permission to run a module on the sensor node. If *IP* accepts the request, it provides *SP* with its provider key for the node, $K_{N,SP}$.

Next, *SP* creates the module *SM*, that he wants to run on the processor and calculates the associated module key, $K_{N,SP,SM}$. Since *SM* will communicate with SM_S, *SP* requests the identity of SM_S from *IP*. A MAC of this identity, created with $K_{N,SP,SM}$, is included in an unprotected section of *SM* so that *SM* can use it to authenticate SM_S. Then *SM* is sent to the node for deployment.

Once *SM* is received on the node, it is loaded, by untrusted software like the operating system, into memory and the processor is requested to protect *SM*, using the `protect` processor instruction. As discussed, the processor enables memory protection, computes the key $K_{N,SP,SM}$ and stores it in hardware.

Now that *SM* has been deployed, *SP* can start requesting data from it. We will assume that *SM*'s function is to request data from S through SM_S, perform some transformation, filtering or aggregation on it and return the result to *SP*. The first step is for *SP* to send a request containing a nonce *No* to the node. Once the request is received (by untrusted code) on the node, *SM* is called passing *No* as an argument.

Before *SM* calls SM_S, it needs to verify the integrity of module SM_S. It does this by executing the `MAC-verify` instruction passing the address of the known MAC of SM_S and the address of the entry point it is about to call. The ID of SM_S is then returned to *SM* and, if it is non-zero, *SM* calls SM_S to receive the sensor data from S. *SM* will usually also store the returned ID of SM_S in its protected data section so that future authentications of SM_S can be done with the `get-id` instruction.

Once the received sensor data has been processed into the output data O, *SM* will request the processor to seal $No||O$ using the `MAC-seal` instruction. *SM* then passes this MAC together with O to the (untrusted) network stack to be sent to *SP*. When *SP* receives the output of *SM*, it can verify its integrity by recalculating the MAC.

4 Implementation

This section discusses the implementation of Sancus. We have implemented hardware support for all security features discussed in Section 3 as well as a compiler that can create software modules suitable for deployment on the hardware.

4.1 The processor

Our hardware implementation is based on an open source implementation of the TI MSP430 architecture: the openMSP430 from the OpenCores project [20]. We have chosen this architecture because both GCC and LLVM support it and there exists a lot of software running natively on the MSP430, for example the Contiki operating system.

The discussion is organized as follows. First, we explain the features added to the openMSP430 in order to implement the isolation of software modules. Then, we discuss how we added support for the attestation related

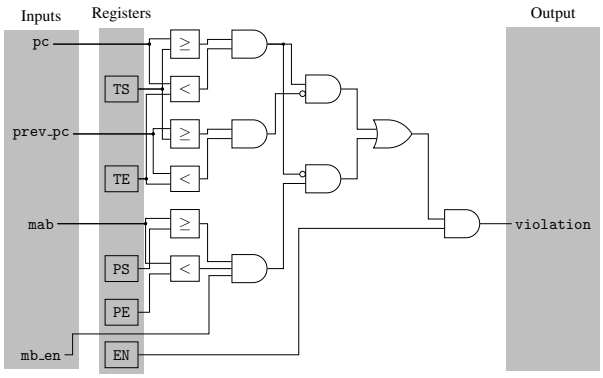

Figure 5: Schematic of the Memory Access Logic (MAL), the hardware used to enforce the memory access rules for a single protected module.

operations. Finally, we describe the modifications we made to the openMSP430 core itself.

Isolation. This part of the implementation deals with enforcing the access rights shown in Table 1. For this purpose, the processor needs access to the layout of every software module that is currently protected. Since the access rights need to be checked on every instruction, accessing these values should be as fast as possible. For this reason, we have decided to store the layout information in special registers inside the processor. Note that this means the total number of software modules that can be protected at any particular time has a fixed upper bound. This upper bound, N_{SM}, can be configured when synthesizing the processor.

Figure 5 gives an overview of the Memory Access Logic (MAL) circuit used to enforce the access rights of a single software module. This MAL circuit is instantiated N_{SM} times in the processor. It has four inputs: `pc` and `prev_pc` are the current and previous values of the program counter, respectively. The input `mab` is the memory address bus – the address currently used for load or store operations[7] – while `mb_en` indicates whether the address bus is enabled for the current instruction. The MAL circuit has one output, `violation`, that is asserted whenever one of of the access rules is violated.

Apart from the input and output signals, the MAL circuit also keeps state in registers. The layout of the protected software module is captured in the `TS` (start of text section), `TE` (end of text section), `PS` (start of protected section) and `PE` (end of protected section) registers. The `EN` register is set to 1 if there is currently a module being protected by this MAL circuit instantiation. The layout is saved in the registers when the `protect` instruction is called at which time `EN` is also set. When the `unprotect` instruction is called, we just unset `EN` which disables all checks.

In our prototype we load new modules through a debug interface on the node and only the debug unit is allowed to write to the memory region where text sections are loaded. Therefore, the read-only nature of text sections is already enforced and the MAL does not need to check this. In a production implementation this check should be added and would cost two additional comparators in the MAL circuit.

Since the circuit is purely combinational, no extra cycles are needed for the enforcement of access rights. As explained above, this is exactly what we want since these rights need to be checked for every instruction. The only downside this approach might have is that this large combinational circuit may add to the length of the critical path of the processor. We will show in Section 5 that this is not the case. Note that since the MAL circuits are instantiated in parallel, N_{SM} does not influence the length of the critical path.

Apart from hardware circuit blocks that enforce the access rights, we also added a single hardware circuit to control the MAL circuit instantiations. It implements three tasks: (1) combine the `violation` signals from every MAL instantiation into a single signal; (2) keep track of the value of the current and previous program counter; and (3) when the `protect` instruction is called, select a free MAL instantiation to store the layout of the new software module and assign it a unique ID.

Attestation. As explained in Section 3, two cryptographic features are needed to implement our design: the ability to create MACs and a key derivation function. Since our implementation is based on a small microprocessor, one of our main goals here is to make the implementation of these features as small as possible.

The MAC algorithm we have chosen is HMAC, the hash-based message authentication code. One of the reasons we have chosen HMAC is its simplicity: only two calls of a hash function are needed to calculate a MAC. Another reason is that it serves as the basic building block for HKDF [28], a key derivation function. This means a lot of hardware can be shared between the implementations of the MAC and the key derivation function. For the hash function, we have chosen to use SPONGENT because it is one of the hash functions with the smallest hardware footprint published to date [5]. More specifically, we use the variant SPONGENT-128/128/8 implemented using a bit-parallel, word-serial architecture, which has a small footprint while maintaining acceptable throughput. Since SPONGENT-128/128/8 requires 8 bit inputs and the openMSP430 architecture is 16 bit, an 8 bit buffer and a tiny finite state machine are required to make the hash

[7] Of course, this includes implicit memory accesses like a `call` instruction.

implementation and the processor work together.

All the keys used by the processor are 128 bits long. The node key K_N is fixed when the hardware is synthesized and should be created using a secure random number generator. When a module *SM* is loaded, the processor will first derive $K_{N,SP}$ using the HKDF implementation which is then used to derive $K_{N,SP,SM}$. The latter key will then be stored in the hardware MAL instantiation for the loaded module. Note that we have chosen to cache the module keys instead of calculating them on the fly whenever they are needed. This is a trade-off between size and speed which we feel is justified because SPONGENT-128/128/8 needs about 8.75 cycles per input bit. Since the module key is needed for every remote attestation and whenever the module's output needs to be signed, having to calculate it on the fly would introduce a runtime overhead that we expect to be too high for most applications.

Under assumptions on the underlying hash function, HMAC is known to be a pseudo-random function [4]. It is shown [28, Section 3] that this is sufficient for a key derivation function, provided that the key to the pseudo-random function (in our notation the first input to $kdf(.,.)$) is uniformly random or pseudo-random. This is the case in our application, hence there is no need to use the more elaborate "extract-and-expand" construction [28].

Core modifications. The largest modification that had to be made to the core is the decoding of the new instructions. We have identified a range of opcodes, starting at 0x1380, that is unused in the MSP430 instruction set and mapped the new instructions in that range.

Further modifications include routing the needed signals, like the memory address bus, into the access rights modules as well as connecting the violation signal to the internal reset. Note that the violation signal is stored into a register before connecting it to the reset line to avoid the asynchronous reset being triggered by combinational glitches from the MAL circuit.

Figure 6 gives an overview of the added hardware blocks when synthesized with support for two protected modules. In order to keep the figure readable, we did not add the input and output signals of the MAL blocks shown in Figure 5.

4.2 The compiler

Although the hardware modifications enable software developers to create protected modules, doing this correctly is tedious, as the module can have only one entry point, and as modules may need to implement their own call-stack to avoid leaking the content of stack allocated variables to unprotected code or to other modules. Hence, we have implemented a compiler extension based on LLVM [37] that deals with these low-level details. We have also implemented a support library that offers an API to perform some commonly used functions like creating a MAC of data.

Our compiler compiles standard C files.[8] To benefit from Sancus, a developer only needs to indicate which functions should be part of the protected module being created, which functions should be entry points and what data should be inside the protected section. For this purpose, we offer three attributes – `SM_FUNC`, `SM_ENTRY` and `SM_DATA` – that can be used to annotate functions and global variables.

Entry points. Since the hardware supports a single entry point per module only, the compiler implements multiple logical entry points on top of the single physical entry point by means of a jump table. The compiler assigns every logical entry point a unique ID. When calling one of the logical entry points, the ID of that entry point is placed in a register before jumping to the physical entry point of the module. The code at the physical entry point then jumps to the correct function based on the ID passed in the register.

When a module calls a function outside its text section, the same entry point is also used when this function returns. This is implemented by using a special ID for the "return entry point". If this ID is provided when entering the module, the address to return to is loaded from the module's stack. Of course, this is only safe if stack switching is also used.

Stack switching. As discussed in Section 3.3, it is preferable to place the runtime stack of software modules inside the protected data section. Our compiler automatically handles everything needed to support multiple stacks. For every module, space is reserved at a fixed location in its protected section for the stack. The first time a module is entered, the stack pointer is loaded with the address of this start location of the stack. When the module is exited, the current value of the stack pointer is stored in the protected section so that it can be restored when the module is reentered.

Exiting modules. Our compiler ensures that no data is leaked through registers when exiting from a module. When a module exits, either by calling an external function or by returning, any register that is not part of the calling convention is cleared. That is, only registers that hold a parameter or a return value retain their value.

Secure linking. Calls to protected modules are automatically instrumented to verify the called module. This

[8] We use Clang [36] as our compiler frontend. This means any C-dialect accepted by Clang is supported.

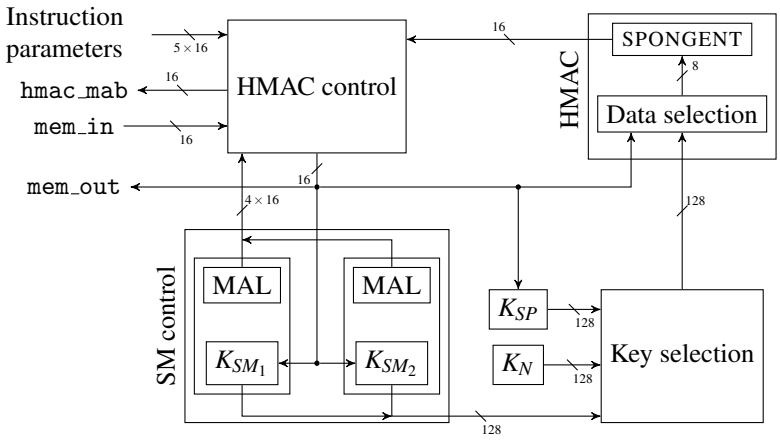

Figure 6: Overview of the hardware blocks added to the openMSP430 core.

includes automatically calculating any necessary module keys and MACs. Of course, a software provider needs to provide its key to the compiler for this function to work.

4.3 Deployment

Since the identity of a *SM* is dependent on its load addresses on node *N*, *SP* must be aware of these addresses in order to be able to calculate $K_{N,SP,SM}$. Moreover, any MACs needed for secure linking will also be dependent on the load addresses of other modules. Enforcing static load addresses is obviously not a scalable solution given that we target systems supporting dynamic loading of software modules by third-party software providers.

Given these difficulties, we felt the need to develop a proof-of-concept software stack providing a scalable deployment solution. Our stack consists of two parts: a set of tools used by *SP* to deploy *SM* on *N* and host software running on *N*. Note that this host software is *not* part of any protected module and, hence, does not increase the size of the TCB.

We will now describe the deployment process implemented by our software stack. First, *SP* creates a relocatable Executable and Linkable Format (ELF) file of *SM* and sends it to *N*. The host software on *N* receives this file, finds a free memory area to load *SM* and relocates it using a custom made dynamic ELF loader. Then, hardware protection is enabled for *SM* and a symbol table is sent back to *SP*. This symbol table contains the addresses of any global functions[9] as well as the load addresses of all protected modules on *N*. Using this symbol table, *SP* is able to reconstruct the exact same image of *SM* as the one loaded on *N*. This image can then be used to calculate $K_{N,SP,SM}$ and any needed MACs. These MACs can then be sent to *N* to be loaded in memory. Note that this deployment process has been *fully* automated.

After *SM* has been deployed, the host software on *N* provides an interface to be able to call its entry points. This can be used by *SP* to attest that *SM* has not been compromised during deployment and that the hardware protection has been correctly activated.

5 Evaluation

In this section we evaluate Sancus in terms of runtime performance, power consumption, impact on chip size and provided security. All experiments were performed using a Xilinx XC6SLX9 Spartan-6 FPGA running at 20MHz.

Performance A first important observation from the point of view of performance is that our hardware modifications do not impact the processor's critical path. Hence, the processor can keep operating at the same frequency, and any code that does not use our new instructions runs at the same speed. This is true independent of the number of software modules N_{SM} supported in the processor.[10] The performance results below are also independent of N_{SM}.

To quantify the impact on performance of our extensions, we first performed microbenchmarks to measure the cost of each of the new instructions. The `get-id` and `unprotect` instructions are very fast: they both take one clock cycle. The other three instructions compute hashes or key derivations, and hence their run time cost depends linearly on the size of the input they handle. We summarize their cost in Table 2. Note that since `MAC-seal` and `MAC-verify` both compute the HMAC of the input data, one might expect that they would need the same number

[9] For example, `libc` functions and I/O routines.

[10] We verified this experimentally for values of N_{SM} up to 8.

Table 2: The number of cycles needed by the new instructions for various input sizes. The input for the instructions is as follows: `protect`: the text section of the software module being protected; `MAC-seal`: the data to be signed; and `MAC-verify`: the text section of the software module to be verified.

Instruction	256B	512B	1024B
`protect`	30,344	48,904	86,016
`MAC-seal`	24,284	42,848	79,968
`MAC-verify`	24,852	43,416	80,536

of cycles. However, since `MAC-verify` includes the layout of the module to be verified in the input to HMAC, it has a fixed overhead of 568 cycles.

To give an indication of the impact on performance in real-world scenarios, we performed the following macro benchmark. We configured our processor as in the example shown in Figure 4. We measured the time it takes from the moment a request arrives at the node until the response is ready to be sent back. More specifically, the following operations are timed: (1) The original request is passed, together with the nonce, to SM_i; (2) SM_i requests SM_S for sensor data; (3) SM_i performs some transformation on the received data; and (4) SM_i signs its output together with the nonce. The overhead introduced by Sancus is due to a call to `MAC-verify` in step (2) and a call to `MAC-seal` in step (4) as well as the entry and exit code introduced by the compiler. Since this overhead is fixed, the amount of computation performed in step (3) will influence the *relative overhead* of Sancus. Note that the size of the text section of M_S is 218 bytes and that nonces and output data signed by M_i both have a size of 16 bits.

By using the `Timer_A` module of the MSP430, we measured the fixed overhead to be 28,420 cycles for the first time data is requested from the module. Since the call to `MAC-verify` in step (2) is not needed after the initial verification, we also measured the overhead of any subsequent requests, which is 6,341 cycles. Given these values, the relative overhead can be calculated in function of the number of cycles used during the computation in step (3). The result is shown in Figure 7.

We believe that these numbers are clear evidence of the practicality of our approach.

Area The unmodified Spartan-6 FPGA implementation of the openMSP430 uses 998 slice registers and 2,322 slice LUTs. The fixed overhead[11] of our modification is 586 registers and 1,138 LUTs. For each protected module, there is an additional overhead of 213 registers and 307 LUTs.

[11] That is, the overhead when $N_{SM} = 0$.

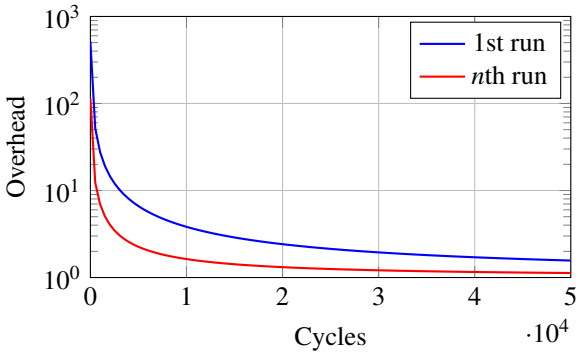

Figure 7: Relative overhead, in function of the number of cycles used for calculations, of Sancus on the macro benchmark. The nth run is significantly faster due to the secure linking optimization discussed in Section 3.5.

There are two easy ways to improve these numbers. First, if computational overhead is of lesser concern, the module key may be calculated on the fly instead of storing it in registers. Second, in applications with lower security requirements, smaller keys may be used reducing the number registers used for storage as well as the internal state of the SPONGENT implementation. Exploring other improvements is left as future work.

Power Our static power analysis tool[12] predicts an increase of power consumption for the processor of around 6% for the processor running at 20MHz. We measured power consumption experimentally, but could not detect a significant difference between an unmodified processor and our Sancus prototype. Of course, since Sancus introduces a runtime overhead, the total overhead in *energy* consumption will be accordingly.

Security We provide an informal security argument for each of the security properties Sancus aims for (see Section 2.3).

First, *software module isolation* is enforced by the memory access control logic in the processor. Both the access control model as well as its implementation are sufficiently simple to have a high assurance in the correctness of the implementation. Moreover, Agten et al. [1] have shown that higher-level isolation properties (similar to isolation between Java components) can be achieved by compiling to a processor with program-counter dependent memory access control. Sancus does *not* protect against vulnerabilities in the implementation of a module. If a module contains buffer-overflows or other memory safety related vulnerabilities, attackers can exploit them using well-known techniques [15] to get unintended access to

[12] We used Xilinx XPower Analyzer.

data or functionality in the module. Dealing with such vulnerabilities is an orthogonal problem, and a wide range of countermeasures for addressing them has been developed over the past decades [47].

The security of *remote attestation and secure communication* both follow from the following key observation: the computation of MACs with the module key is only possible by a module with the correct identity running on top of a processor configured with the correct node key (and of course by the software provider of the module). As a consequence, if an attacker succeeds in completing a successful attestation or communication with the software provider, he must have done it with the help of the actual module. In other words, within our attacker model, only API-level attacks against the module are possible, and it is indeed possible to develop modules that are vulnerable to such attacks, for instance if a module offers a function to compute MACs with its module key on arbitrary input data. But if the module developer avoids such API-level attacks, the security of Sancus against attackers conforming to our attacker model follows.

The security of *secure linking* is the most intricate security property of Sancus. It follows again from the fact that computation of MACs with the module key is only possible by a module with the correct identity running on top of a processor configured with the correct node key, or by the software provider of the module. Hence, an attacker can not forge MACs of other modules that a module wants to link to and call. Because of our technique for separation of uses of MACs (Section 3.1), he can also not do this by means of an API level attack against the module. As a consequence, if a module implements a MAC-verify check for any module it calls[13], this verification can only be successful for modules for which the software provider has deployed the MAC. Hence the module will only call modules that its provider has authorized it to call.

6 Related Work

Ensuring strong isolation of code and data is a challenging problem. Many solutions have been proposed, ranging from hardware-only to software-only mechanisms, both for high-end and low-end devices.

Isolation in high-end devices. The Multics [9] operating system marked the start of the use of protection rings to isolate less trusted software. Despite decades of research, high-end devices equipped with this feature are still being attacked successfully. More recently, research has switched to focus on the isolation of software modules with a minimal TCB by relying on recently added hardware support. McCune et al. propose Flicker [39], a system that relies on a TPM chip and trusted computing functionality of modern CPUs, to provide strong isolation of modules with only a TCB of 250 LOCs. Subsequent research [3, 38, 40, 42] focuses on various techniques to reduce the number of TPM accesses and significantly increase performance, for example by taking advantage of hardware support for virtual machines.

The idea of deriving module specific keys from a master key using (a digest of) the module's code is also used by the On-board Credentials project [27]. They use existing hardware features to enforce the isolated execution of *credential programs* and securely store secret keys. Only one credential program can effectively be loaded at any single moment but the concept of *families* is introduced to be able to share secrets between different programs. Although secure communication is implemented using symmetric cryptography, they rely on public key cryptography during the deployment process.

Isolation in low-end devices. While recent research results on commodity computing platforms are promising, the hardware components they rely on require energy levels that significantly exceed what is available to many embedded devices such as pacemakers [22] and sensor nodes. A lack of strong security measures for such devices significantly limits how they can be applied and vendors may be required to develop closed systems or leave their system vulnerable to attack.

Sensor operating systems and applications, for example, were initially compiled into a monolithic and static image without safety considerations, as in early versions of TinyOS [34]. The reality that sensor deployments are long-lived, and that the full set of modules and their detailed functionality is often unknown at development time, resulted in dynamic modular operating systems such as SOS [23] or Contiki [12]. As stated in the introduction of this paper, the availability of networked modular update capability creates new threats, particularly if the software modules originate from different stakeholders and can no longer be fully trusted. Many ideas have been put forward to address the safety concerns of these shared environments, and solutions to provide memory protection, isolation and (fair) multithreading have appeared. t-kernel [21] rewrites code on the sensor at load time. Coarse-grained memory protection (basically MMU emulation) is available for the SOS operating system by sandboxing in the Harbor system [30] through a combination of backend compile time rewriting and run time checking on the sensor. Safe TinyOS [8] equally uses a combination of backend compile time analysis and minimal run time error handlers to provide type and memory safety. Java's language features and the Isolate mechanism are used on the Sun SPOT embedded platform using the Squawk VM [41]. SenShare [33] provides a virtual

[13] Note that our compiler automatically adds these checks.

machine for TinyOS applications. While these proposed solutions do not require any hardware modifications, they all incur a software-induced overhead. Moreover, third-party software providers must rely on the infrastructure provider to correctly rewrite modules running on the same device.

To increase security of embedded devices, Strackx et al. [43] introduced the idea of a program-counter based access control model, but without providing any implementation. Agten et al. [1] prove that isolation of code and data within such a model only relies on the vendor of the module and cannot be influenced by other modules on the same system. More recently El Defrawy et al. [14] implemented hardware support that allows attestation that a module executed correctly without any interference, based on a similar access control model. While this is a significant step forward, it does not provide isolation as sensitive data cannot be kept secret from other modules between invocations.

7 Conclusion

The increased connectivity and extensibility of networked embedded devices as illustrated for instance by the trend towards decoupling applications and platform in sensor networks leads to exciting new applications, but also to significant new security threats. This paper proposed a novel security architecture called Sancus, that is low-cost yet provides strong security guarantees with a very small, hardware-only, TCB.

8 Availability

To ensure reproducibility and verifiability of our results, we make the hardware design and the software of our prototype publicly available. All source files, as well as binary packages and documentation can be found at `https://distrinet.cs.kuleuven.be/software/sancus/`.

9 Acknowledgments

This work has been supported in part by the Intel Lab's University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

References

[1] AGTEN, P., STRACKX, R., JACOBS, B., AND PIESSENS, F. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium (CSF 2012)* (Los Alamitos, CA, USA, 2012), IEEE Computer Society, pp. 171–185.

[2] ANDERSON, R. J., AND KUHN, M. G. Low cost attacks on tamper resistant devices. In *Proceedings of the 5th International Workshop on Security Protocols* (London, UK, UK, 1998), Springer-Verlag, pp. 125–136.

[3] AZAB, A., NING, P., AND ZHANG, X. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 375–388.

[4] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 1996), CRYPTO '96, Springer-Verlag, pp. 1–15.

[5] BOGDANOV, A., KNEZEVIC, M., LEANDER, G., TOZ, D., VARICI, K., AND VERBAUWHEDE, I. Spongent: The design space of lightweight cryptographic hashing. vol. 99, IEEE Computer Society, p. 1.

[6] BONEH, D., DEMILLO, R. A., AND LIPTON, R. J. On the importance of eliminating errors in cryptographic computations. *J. Cryptology 14* (2001), 101–119.

[7] CASTELLUCCIA, C., FRANCILLON, A., PERITO, D., AND SORIENTE, C. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 400–409.

[8] COOPRIDER, N., ARCHER, W., EIDE, E., GAY, D., AND REGEHR, J. Efficient memory safety for TinyOS. In *Proceedings of the 5th international conference on Embedded networked sensor systems* (New York, NY, USA, 2007), SenSys '07, ACM, pp. 205–218.

[9] CORBATO, F., AND VYSSOTSKY, V. Introduction and overview of the Multics system. In *Proceedings of the November 30–December 1, 1965, Fall joint computer conference, part I* (1965), ACM, pp. 185–196.

[10] DEVRIESE, D., AND PIESSENS, F. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010), pp. 109–124.

[11] DOLEV, D., AND YAO, A. C. On the security of public key protocols. *IEEE Transactions on Information Theory 29*, 2 (1983), 198–208.

[12] DUNKELS, A., FINNE, N., ERIKSSON, J., AND VOIGT, T. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems* (New York, NY, USA, 2006), SenSys '06, ACM, pp. 15–28.

[13] DYO, V., ELLWOOD, S. A., MACDONALD, D. W., MARKHAM, A., MASCOLO, C., PÁSZTOR, B., TRIGONI, N., AND WOHLERS, R. Wildlife and environmental monitoring using RFID and WSN technology. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2009), SenSys '09, ACM, pp. 371–372.

[14] ELDEFRAWY, K., FRANCILLON, A., PERITO, D., AND TSUDIK, G. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA* (San Diego, UNITED STATES, 02 2012).

[15] ERLINGSSON, U., YOUNAN, Y., AND PIESSENS, F. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, 2010.

[16] FAROOQ, M. O., AND KUNZ, T. Operating systems for wireless sensor networks: A survey. *Sensors 11*, 6 (2011), 5900–5930.

[17] FRANCILLON, A., AND CASTELLUCCIA, C. TinyRNG: A cryptographic random number generator for wireless sensors network nodes. In *In Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops, 2007. WiOpt 2007. 5th International Symposium on* (2007), pp. 1–7.

[18] FRANCILLON, A., AND CASTELLUCCIA, C. Code injection attacks on Harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 15–26.

[19] GIANNETSOS, T., DIMITRIOU, T., AND PRASAD, N. R. Self-propagating worms in wireless sensor networks. In *Proceedings of the 5th international student workshop on Emerging networking experiments and technologies* (New York, NY, USA, 2009), CoNext Student Workshop '09, ACM, pp. 31–32.

[20] GIRARD, O. openMSP430. `http://opencores.org/project,openmsp430`.

[21] GU, L., AND STANKOVIC, J. A. t-kernel: providing reliable OS support to wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems* (Boulder, Colorado, USA, 2006), ACM, pp. 1–14.

[22] HALPERIN, D., HEYDT-BENJAMIN, T., RANSFORD, B., CLARK, S., DEFEND, B., MORGAN, W., FU, K., KOHNO, T., AND MAISEL, W. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), Ieee, pp. 129–142.

[23] HAN, C.-C., KUMAR, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2005), MobiSys '05, ACM, pp. 163–176.

[24] HEINZELMAN, W. B., MURPHY, A. L., CARVALHO, H. S., AND PERILLO, M. A. Middleware to support sensor network applications. *IEEE Network 18*, 1 (2004), 6–14.

[25] KOCHER, P. C. 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* (London, UK, UK, 1996), CRYPTO '96, Springer-Verlag, pp. 104–113.

[26] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 1999), CRYPTO '99, Springer-Verlag, pp. 388–397.

[27] KOSTIAINEN, K., EKBERG, J.-E., ASOKAN, N., AND RANTALA, A. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (New York, NY, USA, 2009), ASIACCS '09, ACM, pp. 104–115.

[28] KRAWCZYK, H., AND ERONEN, P. HMAC-based extract-and-expand key derivation function (HKDF). `http://tools.ietf.org/html/rfc5869`.

[29] KUMAR, P., AND LEE, H.-J. Security issues in healthcare applications using wireless medical sensor networks: A survey. *Sensors 12*, 1 (2011), 55–91.

[30] KUMAR, R., KOHLER, E., AND SRIVASTAVA, M. Harbor: software-based memory protection for sensor nodes. In *Proceedings of the 6th international conference on Information processing in sensor networks* (New York, NY, USA, 2007), IPSN '07, ACM, pp. 340–349.

[31] LEE, Y. K., SAKIYAMA, K., BATINA, L., AND VERBAUWHEDE, I. Elliptic-curve-based security processor for RFID. *Computers, IEEE Transactions on 57*, 11 (nov. 2008), 1514 –1527.

[32] LEIGHTON, F. T., AND MICALI, S. Secret-key agreement without public-key cryptography. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, UK, 1994), CRYPTO '93, Springer-Verlag, pp. 456–479.

[33] LEONTIADIS, I., EFSTRATIOU, C., MASCOLO, C., AND CROWCROFT, J. Senshare: transforming sensor networks into multi-application sensing infrastructures. In *Proceedings of the 9th European conference on Wireless Sensor Networks* (Berlin, Heidelberg, 2012), EWSN'12, Springer-Verlag, pp. 65–81.

[34] LEVIS, P. Experiences from a decade of tinyos development. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 207–220.

[35] LIGATTI, J., BAUER, L., AND WALKER, D. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security 4*, 1-2 (2005), 2–16.

[36] LLVM DEVELOPER GROUP. Clang. `http://clang.llvm.org/`.

[37] LLVM DEVELOPER GROUP. LLVM. `http://llvm.org/`.

[38] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2010).

[39] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)* (Apr. 2008), ACM, pp. 315–328.

[40] SAHITA R, WARRIER U., D. P. Protecting Critical Applications on Mobile Platforms. *Intel Technology Journal 13* (2009), 16–35.

[41] SIMON, D., CIFUENTES, C., CLEAL, D., DANIELS, J., AND WHITE, D. Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE* (2006), H.-J. Boehm and D. Grove, Eds., ACM, pp. 78–88.

[42] STRACKX, R., AND PIESSENS, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012),* (Oct. 2012), ACM Press, pp. 2–13.

[43] STRACKX, R., PIESSENS, F., AND PRENEEL, B. Efficient isolation of trusted subsystems in embedded systems. In *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering: Security and Privacy in Communication Networks* (September 2010), vol. 50, Springer, pp. 1–18.

[44] U.S. CUSTOMS AND BORDER PROTECTION. C-TPAT. `http://www.cbp.gov/ctpat`.

[45] VIEGA, J., AND THOMPSON, H. The state of embedded-device security (spoiler alert: It's bad). *Security Privacy, IEEE 10*, 5 (Sept.-Oct. 2012), 68 –70.

[46] WERNER-ALLEN, G., LORINCZ, K., JOHNSON, J., LEES, J., AND WELSH, M. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 381–396.

[47] YOUNAN, Y., JOOSEN, W., AND PIESSENS, F. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Comput. Surv. 44*, 3 (June 2012), 17:1–17:28.

[48] ZÖLLER, S., REINHARDT, A., MEYER, M., AND STEINMETZ, R. Deployment of wireless sensor networks in logistics potential, requirements, and a testbed. In *Proceedings of the 9th GI/ITG KuVS Fachgespräch Drahtlose Sensornetze* (Sep 2010), R. Kolla, Ed., Julius-Maximilians-Universität Würzburg, pp. 67–70.