Sancus 2.0: A Low-Cost Security Architecture for IoT Devices

JOB NOORMAN, JO VAN BULCK, JAN TOBIAS MÜHLBERG and FRANK PIESSENS, iMinds-DistriNet, KU Leuven
PIETER MAENE, BART PRENEEL and INGRID VERBAUWHEDE, iMinds-COSIC, KU Leuven
JOHANNES GÖTZFRIED, TILO MÜLLER and FELIX FREILING, FAU Erlangen-Nürnberg

The Sancus security architecture for networked embedded devices was proposed in 2013 at the USENIX Security conference. It supports 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 provide a secure communication channel between software modules and software providers. Software modules can securely maintain local state, and can securely interact with other software modules that they choose to trust.

Over the past three years, significant experience has been gained with applications of Sancus, and several extensions of the architecture have been investigated – both by the original designers as well as by independent researchers. Informed by these additional research results, this journal version of the Sancus paper describes an improved design and implementation, supporting additional security guarantees (such as confidential deployment) and a more efficient cryptographic core.

We describe the design of Sancus 2.0 (without relying on any prior knowledge of Sancus), and develop and evaluate a prototype FPGA implementation. The prototype extends an MSP430 processor with hardware support for the memory access control and cryptographic functionality required to run Sancus. We report on our experience with using Sancus in a variety of application scenarios, and discuss some important avenues of ongoing and future work.

Categories and Subject Descriptors: []

ACM Reference Format:
ACM Trans. Priv. Secur. V, N, Article A (January YYYY), 32 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

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. Many consumer products contain embedded technology to have Internet connectivity. This Internet of Things (IoT) is estimated to grow to an astonishing number of 26 billion units by 2020 [Gartner 2013]. Second, more and more devices support extensibility of the software they run – often even by third parties different

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 Research Foundation - Flanders (FWO). This work was partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89). Job Noorman, Jo Van Bulck, and Pieter Maene are supported by a doctoral grant of the Research Foundation - Flanders (FWO).
Author's addresses: {Job.Noorman, Jo.VanBulck, JanTobias.Muehlberg, Frank.Piessens}@cs.kuleuven.be, {Pieter.Maene, Bart.Preneel, Ingrid.Verbauwhede}@esat.kuleuven.be, {Johannes.Goetzfried, Tilo.Mueller, Felix.Freiling}@cs.fau.de

from the device manufacturer or device owner. The IoT becomes *infrastructure* on which many stakeholders can install and run software applications. These two factors are important because they enable a vast array of interesting applications, ranging from over-the-air updates on smart cards, over updatable implanted medical devices, to programmable sensor networks or smart home applications. 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 [Giannetsos et al. 2009; Francillon and Castelluccia 2008]. Viega and Thompson [2012] 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-constrained devices, no effective low-cost solutions are known. Many embedded platforms lack standard security features present in high-end processors, such as privilege levels or advanced memory management units that support virtual memory. 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.

Over the past few years, researchers have been exploring alternative security architectures for low-end networked devices. For instance, Eldefrawy et al. [2012] propose SMART, a simple and efficient hardware-software primitive to establish a dynamic root of trust in an embedded processor, and Strackx et al. [2010] propose a simple program counter-based memory access control system to isolate software components. For a more complete overview of this line of work, we refer to Section 7.

The key contribution of this paper is the design, implementation and evaluation of one such security architecture, the Sancus architecture. Sancus was first proposed in 2013 at the USENIX Security conference [Noorman et al. 2013] as a security architecture that supports secure third-party software extensibility for a network of low-end processors with a hardware-only Trusted Computing Base (TCB). Over the past three years, significant experience has been gained with applications of Sancus, including for instance the development of a trust assessment infrastructure that uses Sancus to protect the trust measurement code [Mühlberg et al. 2015], and the design of a smart meter secured by Sancus [Mühlberg et al. 2016]. Also, researchers have been investigating several extensions of Sancus, for instance to support more flexible resource sharing [Van Bulck et al. 2015] or to support confidential loading of code [Götzfried et al. 2015]. Informed by these additional research results, this journal version of the Sancus paper describes an improved design and implementation, supporting additional security guarantees, such as confidential deployment and a more efficient cryptographic core. While this paper extends and improves the conference paper, we carefully made sure that this paper is self-contained: readers do not have to be familiar with the conference paper.

More specifically, we make the following contributions:

— We propose Sancus[1], a security architecture for resource-constrained, extensible networked embedded systems, that can provide strong isolation guarantees, remote attestation, secure communication, secure linking, and confidential software deployment with a minimal (hardware) TCB.

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

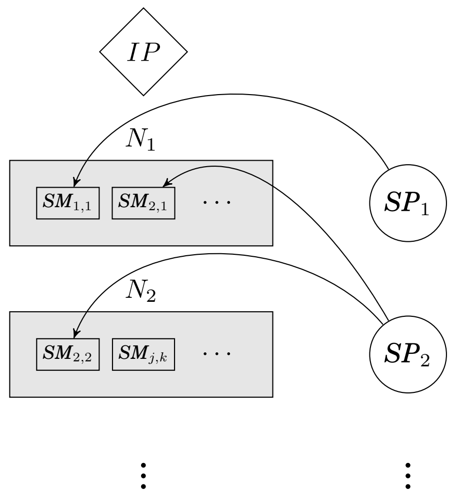

Fig. 1. Overview of our system model. *IP* provides a number of nodes N_i on which software providers SP_j can deploy software modules $SM_{j,k}$.

— 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 using simple annotations with standard C code, showing that the cost in terms of software development is low as well.
— We implement a Contiki-based (untrusted) software stack to automate the deployment process of Sancus modules.
— We report on our experience with implementing a variety of applications on Sancus, and evaluate Sancus in terms of performance, hardware cost, and security.

To guarantee the reproducibility and verifiability of our results, all our research materials, including the hardware design of the processor, the C compiler, and the deployment stack 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 describes applications that have been developed with Sancus, and outlines remaining challenges for future work. Section 6 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.

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 [GlobalPlatform 2015]) 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 that does not support a memory management unit, protection rings, hypervisors, or other security mechanisms typically found on high-end processors.

Any system that supports extensibility (through installation of software modules) by several software providers must implement measures to make sure that the different modules cannot 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 above is still an open problem, and an active area of research [Farooq and Kunz 2011]. One straightforward solution is to transplant the higher-end solutions to these low-end systems: one can extend the processor with virtual memory, or 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 Operating System (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 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 the hardware of the infrastructure and his own modules; he does not need to trust any infrastructural or third-party software on the nodes.

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 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. Note that the security of the communication channel between *IP* and software providers is out of scope.

With respect to the cryptographic capabilities of the attacker, we follow the Dolev-Yao attacker model [Dolev and Yao 1983]: attackers cannot break cryptographic primitives, but they can perform protocol-level attacks.

Finally, attacks against the hardware of individual nodes are out of scope. We assume the attacker does not have physical access to the hardware, cannot place probes on the memory bus, cannot 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 [Kocher 1996; Kocher et al. 1999; Boneh et al. 2001; Anderson and Kuhn 1998]. 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.

Although our attacker model excludes hardware attacks, our security properties do limit the consequences of such an event (Section 2.3, *hardware breach confinement*).

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 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 unmodified on a specific node of *IP*.
— *Secure communication.* A software provider can communicate with a specific software module on a specific node with confidentiality, integrity, authenticity, and freshness guarantees.
— *Secure linking.* A software module on a node can link with 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 with cannot be observed or tampered with by other software on the same node.
— *Confidential deployment.* If so desired, a software provider can deploy encrypted modules. This ensures that no attacker will be able to inspect the module's code at any point in time.
— *Hardware breach confinement.* If an attacker manages to breach the hardware protections on a node, they may be able to manipulate or impersonate modules running on *that* node. However, such a breach should not allow them to do the same with modules running on *other* nodes.

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.

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, in particular authenticated encryption. While public key cryptography would simplify key management, the cost of implementing it in hardware is too high [Lee et al. 2008].

We first present some cryptographic primitives that will be used in the rest of this paper (Section 3.1). Then, we give an overview of our design (Section 3.2) followed by the elaboration of its most interesting aspects (Sections 3.3 to 3.7). We conclude this section with an end-to-end example (Section 3.8).

3.1. Cryptographic Primitives

Throughout the design of Sancus, we assume the existence of three cryptographic primitives. First, a classical cryptographic hash function is used to compute digests of data. Second, a *key derivation function* is used to derive a cryptographic key from a master key and some diversification data:

$$K_{M,D} = \textbf{\textit{kdf}}(K_M, D)$$

Third, an *authenticated encryption with associated data* primitive is used to simultaneously provide confidentiality, integrity, and authenticity guarantees on data. Such a primitive consists of two functions: one for encryption and one for decryption. The encryption function takes as input a key, plaintext and associated data and produces ciphertext and an authentication tag. The ciphertext covers the given plaintext and the tag is a Message Authentication Code (MAC) over both the plaintext and the associated data:

$$C, T = \textit{aead-encrypt}(K, P, A)$$

The decryption function does the opposite operation and fails (i.e., produces no plaintext) if the tag is incorrect for the given ciphertext and associated data:

$$P = \textit{aead-decrypt}(K, C, A, T)$$

Note that this primitive can be used to compute a plain MAC over some data:

$$\textit{mac}(K, D) \equiv \textit{aead-encrypt}(K, \{\}, D)$$

(Here, we discard the ciphertext result of *aead-encrypt*.)

3.2. Overview

3.2.1. Nodes. Nodes are low-cost, low-power microcontrollers (our implementation is based on the TI MSP430). The processor in a node 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.

3.2.2. Software Providers. Software providers are principals that can deploy software to the nodes of *IP*. Each software provider has a unique public ID *SP*.[2] *IP* uses a key derivation function *kdf* to compute a key $K_{N,SP} = \textit{kdf}(K_N, \textit{SP})$, 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.

3.2.3. Software Modules. Software modules are essentially simple binary files containing two mandatory sections: a *text section* containing code and constants and a *data section* containing a module's runtime data. 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 a hash of (1) the content of the text section and (2) the start and end addresses of the text and data sections. We refer to this second part of the identity as the *layout* of the module. It follows that two modules

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

Table I. Overview of the keys used in Sancus, how they are created and who can access them. Note that derived keys are also accessible by any entity that has access to their master keys but this is not explicitly mentioned.

Key	Creation	Accessible by
K_N	Random	*IP*, N
$K_{N,SP}$	$kdf(K_N, SP)$	*SP*
$K_{N,SP,SM}$	$kdf(K_{N,SP}, SM)$	*SM* (indirectly)

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*. A software module is deployed by loading each of the sections of the module in memory at the specified addresses. For each module, the processor maintains the layout information in a *protected storage* area inaccessible from software. It follows that the node can compute the identity of all modules loaded on it: 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 (Section 3.5).

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. Table I gives an overview of the keys used by Sancus.

Note that the provider *SP* can also compute $K_{N,SP,SM}$, 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 secure the communications between *SM* and *SP*.

Figure 2 shows a schematic of a node with a software module loaded. The picture also shows the keys and the layout information that the node has to manage.

3.2.4. 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. Our design relies on *program counter-based memory access control* [Strackx et al. 2010], as this memory access control model has been shown to support strong isolation [Strackx and Piessens 2012], as well as remote attestation [Eldefrawy et al. 2012]. Roughly speaking, isolation is implemented by restricting access to the 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 special instructions to access the cryptographic capabilities. If such an instruction is invoked from within the text section of a specific module *SM*, the processor will use key $K_{N,SP,SM}$. Moreover, these instructions are only available after memory protection has been enabled for module *SM*. It follows that only a well-isolated *SM* installed on behalf of *SP* on N can compute cryptographic primitives with $K_{N,SP,SM}$, and this is the basis for implementing both remote attestation and secure communication.

3.2.5. Remote Attestation and Secure Communication. In order to provide a confidential, integrity-protected, and authenticated communication channel between a software

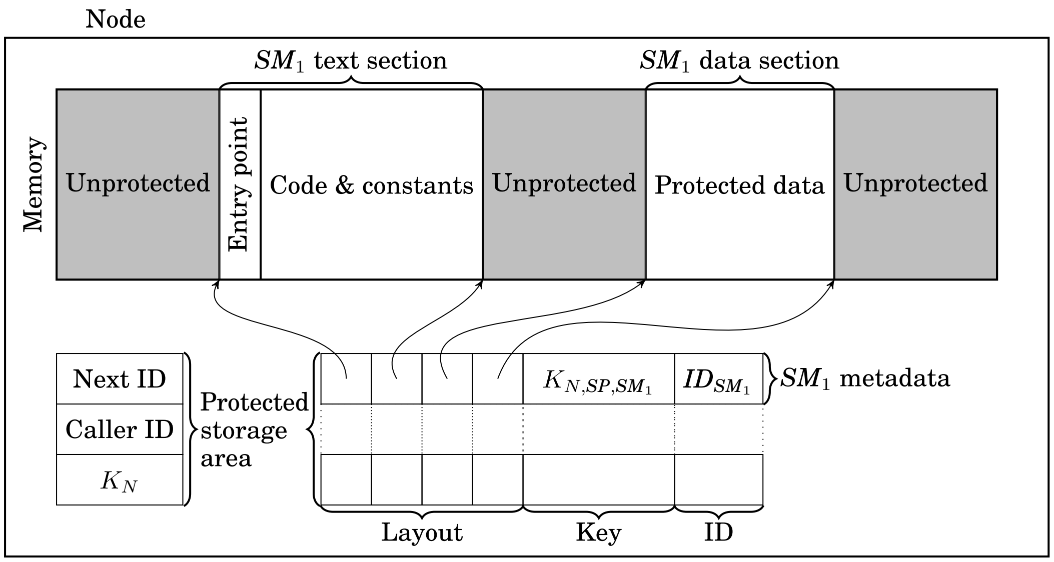

Fig. 2. A node with a software module loaded. The left part of the protected storage area is global while the right part is per module metadata. Sancus ensures the keys can never leave the protected storage area by only making them available to software in indirect ways through new processor instructions.

provider and its modules, Sancus includes an authenticated encryption primitive. New instructions are provided to encrypt and decrypt messages using the key of the calling module. When *SP* receives a message encrypted with $K_{N,SP,SM}$, he will have high assurance that it has been produced by *SM* since, as mentioned above, only *SM* is able to use this key. Note that since *SM* is the *identity* of the module that *SP* is communicating with, this primitive also provides for remote attestation.

3.2.6. 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 the identity (i.e., a hash) of SM_2 in its text section.[3] The processor includes a new 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 identity of that module has the expected value.

A similar mechanism can be used by SM_2 to verify that it is indeed called by SM_1 (*caller authentication*). In its entry point, SM_2 can call a new instruction that verifies the identity of the module that called the entry point. For this to work, the processor keeps track of the *previously* executing software module.

Fortunately, this expensive – a hash needs to be calculated over a potentially large text section – authentication of software modules is needed only once. Section 3.7 discusses a more efficient procedure for subsequent authentications.

[3]Note that if SM_2 also wants to link to SM_1, this method creates a circular dependency between their identities. This can be resolved by not including the other's identity in the text section but having the software provider securely send it after deployment and storing it in the data section.

3.3. Key Management

We handle key management without relying on public-key cryptography [Leighton and Micali 1994]. *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 (Table I):

— 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, hence violating *hardware breach confinement* (Section 2.3). 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*.[4] 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.

3.4. 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 constants, and the 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, runtime metadata such as the module's call stack should typically also be included in the 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 runtime metadata is in its data section, but our implementation comes with a compiler that ensures this automatically (Section 4.2).

The memory access control logic in the processor enforces that (1) the data section of a module is only accessible while code in the text section of that module is being

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

Table II. 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	Data	Unprotected
Entry	`r-x`	`r-x`	`rw-`	`rwx`
Text	`r-x`	`r-x`	`rw-`	`rwx`
Unprotected/ Other SM	`--x`	`---`	`---`	`rwx`

executed, (2) the text section can only be executed by jumping to a well-defined entry point, and (3) the text section cannot be written and can only be read while code in that section is being executed. The second part is important since it prevents attackers from misusing code chunks in the text section to extract data from the data section. For example, without this guarantee, an attacker might be able to launch a Return-Oriented Programming (ROP) attack [Castelluccia et al. 2009] by selectively combining gadgets found in the text section. Of course, if a module contains a bug that allows an attacker to divert its control-flow, he might still be able to launch such an attack; enforcing an entry point prevents these attacks being launched from code outside of the module. Note that, as shown in Figure 2, 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 II gives an overview of the enforced access rights.

Besides memory access control, the processor also ensures that modules cannot be interrupted while being executed to prevent register contents from leaking outside the module. Supporting interruptible modules is orthogonal to our goals and is left as future work.

Memory access control for a module is enabled at the time the module is loaded. First, untrusted code (for instance the node's operating system) will load the module in memory as discussed in Section 3.2. 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 (Section 3.2 and Figure 2);
— 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 OS that loads the module in memory: if the content of the text section or the layout 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` *continuation*

The effect of this instruction is to lift the memory protection of the module *from which the* `unprotect` *instruction is called*. To prevent the leakage of confidential data, this instruction also clears the module's code- and data sections. Since the `unprotect` instruction itself is part of the code section, the programmer has to provide a pointer to the code where the execution is to be continued in the *continuation* argument.

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 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 [Devriese and Piessens 2010]. Invalid writes or jumps could be dropped or modified to actions that are allowed as in edit-automata [Ligatti et al. 2005]. For instance, an invalid memory read might just return zero, and an invalid jump might be redirected to an exception handler.

3.5. Remote Attestation and Secure Communication

We extend the processor with two more instructions that are used for remote attestation and secure communication:

`encrypt` *plaintext, associated data, ciphertext (output), tag (output)[, key]*
`decrypt` *ciphertext, associated data, tag, plaintext (output)[, key]*

These instructions have the same semantics as the *aead-encrypt* and *aead-decrypt* functions, respectively (Section 3.1).

As can be seen from the signatures above, both instructions have the key as an optional argument. If none is given, the module key of the invoking module is implicitly used (or an error code is returned if invoked by unprotected code). This is the *only* way for software to access a module key and the key $K_{N,SP,SM}$ will *only* be used when invoked by the module with identity *SM* deployed by *SP* on node N. Note that, besides being able to access the module key, these instructions are *not* privileged and the same memory access rules are enforced as for any instruction that accesses memory.

These instructions can be used to provide confidentiality, integrity, and authenticity guarantees of data exchanged between modules and their providers. The ciphertext plus the corresponding tag can be sent using the untrusted operating system over an untrusted network. If the tag verifies correctly (using $K_{N,SP,SM}$) upon receipt by the provider *SP*, he can be sure that the decrypted plaintext indeed comes from *SM* running on N on behalf of *SP* as the node's hardware makes sure only this specific module can use the module key $K_{N,SP,SM}$. The reasoning is equivalent for data sent to the module.

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 with the `encrypt` instruction (Section 3.1 explains how this can be done). 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 *aead-encrypt*$(K_{N,SP,SM}, O, No\,||I)$ using the `encrypt` instruction. The resulting ciphertext and tag are then sent to *SP*. By verifying the tag 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.

SP can use secure communication to establish a shared secret between two or more of its modules to allow them to directly communicate with each other. Although this is feasible for modules running on different nodes, the overhead is probably too high for secure communication between modules running on the same node; we discuss a more efficient technique in Section 3.7.

3.6. Confidential Loading

If, besides the integrity guarantees provided through remote attestation, one wants to have *confidentiality guarantees* for a module's text section, more architectural support is necessary. Indeed, although a module's text section is not readable by other modules (Table II), this is only enforced after enabling a module; i.e., up to that point an attacker can easily read the module's text section.

Therefore, we provide a second way to use the `protect` instruction:

`protect` *layout*, *SP*, *MAC*

In this form, the `protect` instruction behaves exactly the same (Section 3.4) except that, before calculating the module key, the module's text section is decrypted in place using $K_{N,SP}$. If the integrity check using the given MAC, i.e., an authenticated encryption tag, fails, the text section is cleared and the protection disabled.

Note that $K_{N,SP}$ is now used to encrypt confidential modules, as well as for key derivation. However, the uniqueness of both operations can be guaranteed by domain separation, i.e., by setting the first bit of the associated data to 0 or 1 for the encryption and key derivation respectively.

It should be mentioned that the integrity check is not strictly necessary for confidential loading, since any subsequent remote attestation will also verify the module's integrity. However, it could be used as a simple form of module authentication: by disabling the non-decrypting form of the `protect` instruction, only entities possessing a valid software provider key can install modules on the system.

3.7. 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.

In our design, if module SM_1 wants to link securely to SM_2, SM_1 should be deployed with the identity of SM_2. The processor provides a special instruction to check the existence and integrity of a module at a specified address:

`attest` *address, expected hash*

This instruction will:

— verify that a module is loaded (with protection enabled) at the provided address;
— compute the identity of that module (i.e., a hash of its text section and layout);
— compare the resulting hash with the *expected hash* parameter of the instruction; and
— if the hashes were equal, return the module's ID (to be explained below), otherwise return zero.

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 hash), our design also includes a more efficient mechanism for repeated authentication. The processor will assign sequential IDs[5] to modules that it loads, and will ensure

[5]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*.

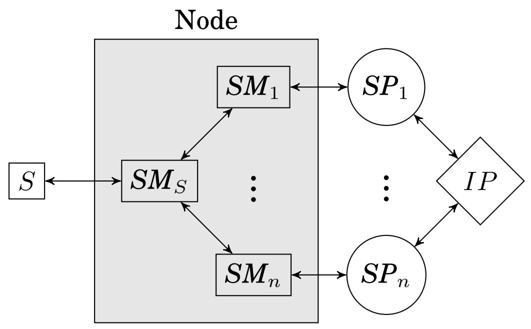

Fig. 3. Setup of the sensor node example discussed in Section 3.8. Sancus ensures only module SM_S is allowed to directly communicate with the sensor S. Other modules securely link to SM_S to receive sensor data in a trusted way.

that – within one boot cycle – it never reuses these IDs. This can be implemented by storing the ID to be used for the next module in a register ("Next ID" in Figure 2), incrementing it after a new module is enabled, and generating a violation when it overflows. A processor instruction:

`get-id` *address*

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.

For caller authentication, the processor keeps track of the previously executing module by recording its ID in a register ("Caller ID" in Figure 2). This register is updated whenever execution enters a new module. Modules can attest their caller through two instruction: `attest-caller` and `get-caller-id`. These instruction behave similar to `attest` and `get-id` respectively but use the previously executing module implicitly.

3.8. 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 3 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, SM_S, that is in charge of communicating with S. By ensuring that the 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 3 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*. This identity is included in the text 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 $K_{N,\mathit{SP},\mathit{SM}}$, 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 `attest` instruction passing the address of the expected identity of SM_S (included in *SM*'s text section) 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 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 calculate $\mathit{aead\text{-}encrypt}(K_{N,\mathit{SP},\mathit{SM}}, O, \mathit{No})$ using the `encrypt` instruction. *SM* then passes the ciphertext C and tag T to the (untrusted) network stack to be sent to *SP*. When *SP* receives the output of *SM*, it can verify its integrity by calculating $\mathit{aead\text{-}decrypt}(K_{N,\mathit{SP},\mathit{SM}}, C, \mathit{No}, T)$.

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 [Girard 2016]. 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 cryptographic operations. Finally, we describe the modifications we made to the openMSP430 core itself.

4.1.1. Isolation. This part of the implementation deals with enforcing the access rights shown in Table II. 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 4 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 five 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[6] – while `mb_en` indicates whether the address bus is enabled for the current instruction and `mb_wr` whether the access is a

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

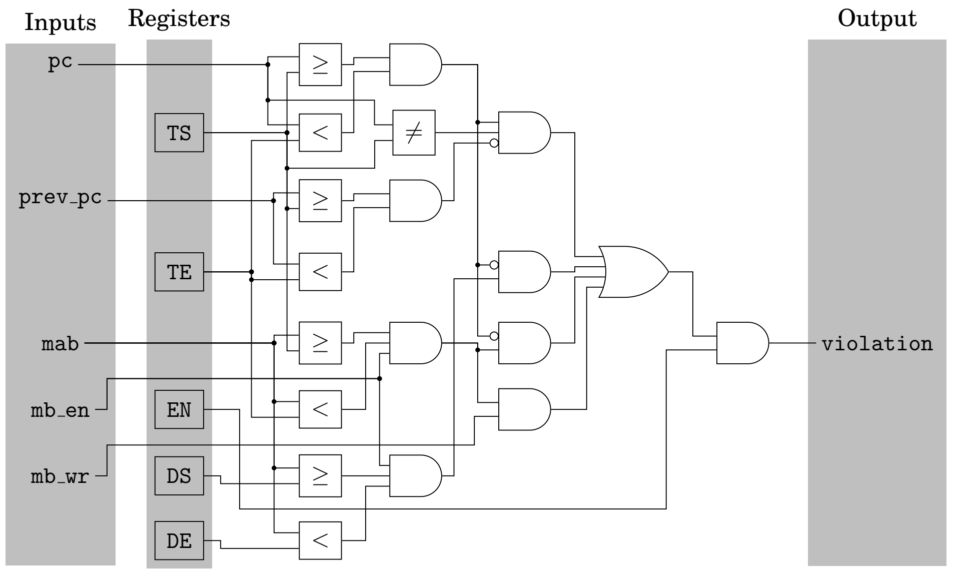

Fig. 4. Schematic of the Memory Access Logic (MAL), the hardware used to enforce the memory access rules for a single protected module.

write. The MAL circuit has one output, `violation`, that is asserted whenever one 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), `DS` (start of data section) and `DE` (end of data 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.

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 the large combinational circuit adds to the length of the critical path of the processor. We will explore the implications our design has on the processor's critical path in Section 6.1.

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 four 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; (3) keep track of the currently and previously executing *SM*; and (4) 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.

4.1.2. Cryptography. As explained in Section 3.1, a hardware implementation of three cryptographic primitives is needed to implement our design: authenticated encryption, key derivation and hashing. 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.

We have chosen to build these cryptographic primitives on the SPONGEWRAP [Bertoni et al. 2011] authenticated encryption construction using SPONGENT [Bogdanov et al. 2012] as the underlying sponge function. Since keyed sponge functions are shown to be pseudorandom functions [Andreeva et al. 2015b], we can reuse SPONGEWRAP to calculate MACs, and consequently for key derivation. Since the security of SPONGEWRAP relies on the soundness of the sponge function it uses, it can also be used as a hashing function by calling *aead-encrypt*$(\{\}, \{\}, M)$.

Besides being able to use it for all necessary primitives, there are several reasons we use SPONGEWRAP with SPONGENT. Since the security of SPONGEWRAP is proportional to the capacity of the underlying sponge function[7], and SPONGENT is defined for a large range of capacities, we can create an implementation with a selectable security parameter. More specifically, our core can be synthesized with a security parameter between 16 and 256 bits, although values less than 80 bits should be avoided. Since the security parameter influences the core's area (Section 6.2), it is a trade-off between cost and security.

As we will see later, all module keys are stored in hardware making the key size an important design parameter regarding area. Another advantage of SPONGEWRAP is that the key size may be as small as the security parameter whereas some other lightweight authenticated encryption primitives require a key that is twice as long, e.g., APE [Andreeva et al. 2015a].

A downside of SPONGEWRAP is that uniqueness of the associated data is required for confidentiality, and no security guarantees can be given when a nonce is reused. More specifically, if two ciphertext messages are captured that are encrypted with the same key and associated data, part of the XOR of the corresponding plaintext message may be leaked (see [Bertoni et al. 2011] for details). Therefore, the user of this primitive should ensure that, for a specific key, the associated data is unique, i.e., that it includes a nonce. Note that this is only necessary when encrypting data and there is no nonce requirement for creating MACs. In contrast, *nonce-misuse resistant* authenticated encryption algorithms (e.g., APE mentioned above) limit information leakage about the message when the nonce is reused, but this comes at an additional implementation cost.

It is a software provider's responsibility that the nonce requirement is fulfilled by the modules it deploys. In our prototypes, this is achieved by having *SP* send an initial counter value as nonce in its first message to a newly deployed module. For subsequent messages, modules can simply increment the counter and use that value as the next nonce. Alternatively, if *SP* never wants to send messages to a module, the initial counter value can be included in the module's text section.

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 SPONGEWRAP 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 performance which we feel is justified because, when using 128 bit keys, SPONGEWRAP needs about 90 cycles per input byte (Section 6.1). Since the module key is needed for every remote attestation and whenever the module's output needs to be encrypted, having to calculate it on the fly would introduce a runtime overhead that we expect to be too high for most applications.

[7]To be exact, for a sponge function with a capacity of c bits, SPONGEWRAP has a security of $c/2$ bits [Bertoni et al. 2011].

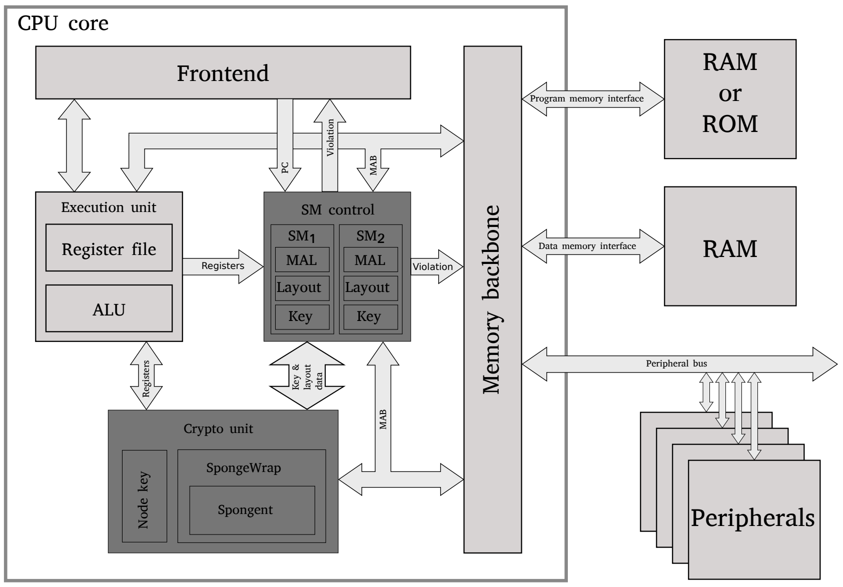

Fig. 5. Overview of the hardware blocks in the Sancus core. Lightly shaded blocks are part of the original openMSP430 design while the darkly shaded ones are added specifically for Sancus. Remember that, while we only draw two *SM* blocks for clarity, this number (N_{SM}) can be chosen when synthesizing the core. Notice how the *SM* control unit takes the program counter (PC) and the memory address bus (MAB) as input to produce the violation signal using the memory access logic (MAL) circuits.

Because of the associated data uniqueness requirement explained above, our implementation of confidential loading is slightly different from its design (Section 3.6). Since modules deployed on N by *SP* are always encrypted using $K_{N,SP}$, the `protect` instruction takes an extra argument, *nonce*, to be able to fulfill the nonce requirement. This argument is used as the associated data input for the decryption routine.

4.1.3. 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.

Since our experience has shown that developing applications on a system that resets on violations is rather tedious, we added a synthesis option to generate a non-maskable interrupt instead. If this option is enabled, the memory backbone will disable all memory accesses when a violation is generated and the frontend will initiate the IRQ sequence. Although this may superficially seem secure, it brings with it a number of problems (e.g., if a module generates a violation, its register contents will be leaked) we have not dealt with yet. Therefore, it currently is not enabled by default and should not be used in production environments.

Figure 5 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 4.

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 [LLVM Developer Group 2016b] 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 calculating 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.

4.2.1. 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 a logical entry point, its ID 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 an external function, 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 enabled.

4.2.2. Stack Switching. As discussed in Section 3.4, it is preferable to place the runtime stack of software modules inside the 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.

4.2.3. 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.

4.2.4. Secure Linking. Calls to protected modules are automatically instrumented to verify the called module. This includes automatically calculating any necessary module keys and identities (Section 3.7). 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 module 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 identity hashes 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.

[8] We use Clang [LLVM Developer Group 2016a] as our compiler frontend. This means any C-dialect accepted by Clang is supported.

Given these difficulties, we felt the need to develop a proof-of-concept software stack providing a 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 which can then be used to calculate $K_{N,SP,SM}$.

Note that an alternative linking strategy is for *SP* to first request the node's symbol table, link the module locally and then send it to the node to be loaded. This would simplify the node since the custom ELF loader is not needed in this scheme. However, since our toolchain does not support position-independent code, this would mean that the memory locations where the module is going to be loaded need to be reserved while *SP* links the module. We feel that this two-phase deployment scheme adds more overall complexity than a simple ELF loader.

Since the dynamic loader needs to inspect and update parts of the text section of modules, this process does not work when confidential loading is used. Although our tools currently do not support the fully automatic loading of encrypted modules, it can be implemented as follows. First, *SP* sends a request to N indicating the sizes of the sections of the module it wants to load. Then, the host software allocates memory for those sections and replies with a handle identifying the allocated memory and a symbol table. Using this symbol table, *SP* links *SM* locally and sends the resulting image, together with the memory handle, back to N. The host software on N then loads it in the pre-allocated memory sections and enables its protection.

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 activated.

This interface is used to upload the identity hashes to *SM* of the modules it securely links to. To this end, *SP* either calculates these hashes after it received the symbol table or, if it concerns modules belonging to a different software provider that use confidential loading, receives them from their respective providers. Then, *SP* encrypts those hashes using $K_{N,SP,SM}$ and sends them to *SM* using the interface described above.

5. APPLICATION SCENARIOS

We have investigated application scenarios for Sancus in the context of trust assessment for IoT devices and sensor nodes [Mühlberg et al. 2015], and for building a secure smart electricity meter for the emerging smart electricity grid [Mühlberg et al. 2016]. Both applications yield promising results, in particular with respect to system performance and the size of the software TCB. In this section we briefly present our findings from these two scenarios, and outline challenges and future applications in domains such as embedded real-time control systems.

5.1. Trust Assessment Modules for the Internet of Things

Devices in the IoT or in Wireless Sensor Networks (WSNs) are typically equipped with inexpensive low-performance microcontrollers. Yet, these devices are interconnected and thereby exposed to physical as well as virtual attacks [Roman et al. 2011]. Even when

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

not considering malicious interference, the device's autonomous mode of operation, exposure to harsh environmental conditions and resource scarceness, make these systems prone to malfunction and the effects of software aging.

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 WSNs [Fernandez-Gago et al. 2007; Granjal et al. 2015; Lopez et al. 2010]. Most techniques proposed focus on observing the communication behavior and on validating the plausibility of sensor readings obtained from nodes, so as to assess the trustworthiness of these nodes. This approach is certainly suitable to detect the systematic failure or misbehavior of single nodes. However, detection is not immediate and a malfunctioning node may output corrupted data that is not labelled as such, before the network will begin to distrust the node: the quality of readings from a sensor may degrade gradually, software failures may lead to non-deterministic behavior or a node may be captured by an attacker, exposing benign and malicious behavior alternately.

In [Mühlberg et al. 2015] we show that the above shortcoming can be mitigated by employing Sancus in an 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. Sancus allows us to integrate trust assessment modules into a largely unmodified and untrusted embedded OS without using techniques such as virtualization and hypervisors, which would incur unacceptable performance overheads for many embedded applications. With Sancus' remote attestation functionality, 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.

We describe and evaluate a prototype implementation of our approach that integrates Sancus-protected trust assessment modules with the Contiki OS [Dunkels et al. 2004], measuring properties such as code integrity, the content of OS- and application data structures, the availability of system resources or the occurrence of system events. As an example, we have implemented a trust assessment module that monitors the process list maintained by Contiki's scheduler, and checks code integrity of the processes present in the above list. To test the effectiveness of our trust assessment module, our scenario integrates a number of trivial application processes and an attacker process that aims to perform alterations to OS data and application code. Expectedly, all changes performed by the attacker are detected and reported with the subsequent invocation of the trust assessment module. 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 [Erlingsson et al. 2010] to be adapted for IoT nodes and in the domain of WSNs.

5.2. High Assurance Smart Metering

With the rise of the smart electricity grid and the extended use of renewable energy resources, there is need for appliances and metering equipment to become smart. Smart appliances can use electricity efficiently when it is inexpensive due to, e.g., local production or the behavior of other users of the grid. Smart electricity meters must therefore timely communicate measurements of local energy consumption and production to grid operators. Being part of the critical infrastructure of our society, security of the smart grid must be guaranteed, not only at the level of the grid operator but also at the level of individual premises.

In [Mühlberg et al. 2016] we implement a simplified scenario for smart meter deployment and evaluate security aspects of this scenario. Our implementation is loosely based on the British "Smart Metering Implementation Programme" [Department of Energy and Climate Change 2014] but simplifies communication protocols and adopts architectural changes suggested in [Cleemput et al. 2016], relying on Sancus to implement security features. Importantly, the goal of this case study is *not* to accurately implement [Department of Energy and Climate Change 2014] but to provide a security-focused reference implementation that illustrates the use of Sancus to achieve a notion of *high assurance smart metering* by means of logical component isolation, mutual authentication, and by minimizing the software TCB.

Our scenario contains software components that implement a smart electricity meter to be installed at a client's premises, and a Load Switch that can enable or disable power supply to the premises. We further implement components to represent the grid operator's Central System and an In-Home Display. The smart meter and the Load Switch communicate with the Central System via a Wide Area Network (WAN) Interface. In our case, the WAN Interface supports periodic access to the smart meter's operational data, as well as control of the Load Switch. The smart meter and the In-Home Display communicate via the Home Area Network (HAN) Interface. Only consumption data is periodically sent from the smart meter to the In-Home Display via this interface. All components are meant to be deployed as protected software modules on a Sancus-like infrastructure that facilitates software component isolation and authenticated and secure communication between these modules. Relying on these security primitives, our approach and prototype guarantee that all outputs of the software system can be explained by the system's source code and the actual physical input events. We further guarantee integrity and confidentiality of messages while relying on a very small software TCB at runtime – less than 300 LOC, excluding drivers. For scenarios that involve critical infrastructure, such as the smart grid, we believe that our approach has the strategic advantage of enabling formal verification and security certification of small, isolated software components while maintaining the strong security guarantees of the distributed system that is formed by the interaction of these components.

5.3. Challenges and Future Directions

In the past few years we have seen a number of attack scenarios in which safety-critical infrastructure was successfully compromised and abused at the site of the end user. Amongst the most prominent cases are certainly remote attacks against insulin pumps [Radcliffe 2011], general hospital equipment [Erven and Merdinger 2014], and against automotive vehicles [Miller and Valasek 2015]. In all these cases the initial subject of an attack was software. In particular, software that would interact with critical components of the appliance but that was not considered safety-critical itself. We believe that these attack scenarios would become infeasible or at least very hard to implement if critical software components would be integrity protected, resistant against certain code misuse attacks, not vulnerable to low-level attacks that exploit implementation details, and use authenticated encryption to securely communicate with each other. Sancus provides a trusted computing infrastructure to securely implement the above scenario, excluding many attack vectors by design, in particular when combined with fully abstract compilation [Patrignani et al. 2015].

In future application scenarios for Sancus we will investigate authentic execution for distributed event-driven applications that execute on a heterogeneous shared infrastructure and require a small TCB. Similar to the above example of a high assurance smart meter, these applications are characterized by consisting of multiple components that execute on different computing nodes and for which program flow is determined by events such as the occurrence of sensor readings.

As an example, consider an automotive anti-lock braking system (ABS) with its sensors (pedal and rotation) and actuators (brake hydraulics). For these systems, we strive for a notion of authentic execution that entails, roughly speaking, the following: if the application produces a physical output event (e.g., engaging the brakes), then there must have happened a sequence of physical input events (sensor readings) such that this sequence, when processed by the application (as specified by the application's source code), produces that output event. This provide strong integrity guarantees, ruling out both spoofed events as well as tampering with the execution of the program. The problem of trustworthiness of automotive control systems has been addressed in recent standards and research (e.g., [AUTOSAR 2015; Nürnberger and Rossow 2016]), albeit without consideration of software security and software integrity that our approach can provide. Ongoing research further investigates the reliability of automotive sensor readings, which can be subject to active spoofing attacks (cf. [Shoukry et al. 2013; Shin et al. 2016]). Generally, these vulnerabilities have to be addressed at the physical layer and are out of scope for Sancus.

Handling input and output events requires the ability to implement secure I/O drivers. Section 3.8 already described a simple way to implement such a driver: by mapping its data section over the memory-mapped I/O region of a device, the driver gains exclusive access over this device. However, future work will have to address open issues such as sharing devices between multiple modules and securely delivering interrupts to drivers.

Yet, implementing distributed control systems based on Sancus or similar technology is difficult beyond achieving security guarantees. In ongoing work we address challenges with respect to efficiently implementing resource sharing between Sancus modules, and the problem of providing availability and real-time guarantees.

5.3.1. Implementing Resource Sharing and Access Control. Traditionally, software for small microcontrollers relies heavily on implementing communication between software components via shared memory, and sharing I/O devices between different components. Of course, this has security implications such that software components must mutually trust each other. Isolating different software components as protected modules, however, limits their ability to securely share system resources, asking for new development paradigms and potentially imposing performance overheads. In [Van Bulck et al. 2015], we describe and evaluate an approach to implement and securely enforce application-grained access control policies for IoT nodes. Our access control mechanism can manage access to various system resources such as file systems, I/O devices, or specific devices attached to an external communication bus. While incurring low overheads, our mechanism guarantees at runtime that only authenticated software modules gain access to resources as specified in the policy; the internal state of the access control implementation is protected and cannot be tampered with. We evaluate a prototypic implementation of our access control mechanism in two application scenarios that facilitate secure data sharing between software modules through (1) a shared memory implementation and (2) peripheral flash memory and the Coffee [Tsiftes et al. 2009] file system. Our evaluation shows that module isolation and access control impose relatively low overheads that should be acceptable in deployment scenarios with stringent safety and security requirements.

5.3.2. Availability and Real-Time Guarantees for Sancus. The security guarantees offered by current protected module architectures are limited to confidentiality and integrity guarantees – they do not extend to *availability*. A buggy or malicious application can still harm the availability of the platform by overwriting crucial OS data structures, or by monopolizing a shared system resource such as CPU time. In [Van Bulck et al. 2016], we report on our work-in-progress towards lifting this limitation. We argue that, in addition to isolating software, hardware-level protection mechanisms can be extended to also preserve availability (possibly even real-time) guarantees on a partially compromised embedded system. The objective of this line of work is to address some of the availability challenges induced by hardware-level protected module architectures, while keeping the TCB small. We outline a hardware mechanism that makes Sancus' protected modules fully interruptible and reentrant. We show how our mechanism preserves deadlines for external events, and facilitates reasoning about real-time guarantees by ensuring a deterministic interrupt latency at all times. We sketch a multitasking model that introduces protection domains *within* a conventional control flow thread and show how logical threads can be managed by an unprivileged scheduler.

6. EVALUATION

In this section we evaluate Sancus in terms of runtime performance, impact on chip size, and provided security. All experiments were performed using a Xilinx XC6SLX25 Spartan-6 FPGA with a speed grade of -2, synthesized using Xilinx ISE Design Suite.

6.1. Performance

There are two important performance aspects to consider with our design. First, since we made changes to the CPU core, we evaluate the impact on its critical path, i.e., the maximum frequency it can run at. Second, we measure the runtime overhead of the added instructions, as well as the code transformations performed by the compiler.

6.1.1. Critical Path. Since the Xilinx tools offer no direct way to find the critical path of a design, we measured it indirectly. Using timing constraints, one can specify what clock rate certain signals should be able to sustain; the tools will then err when the constraint cannot be met. By varying the constraint on the input clock signal, we can get a measure on how fast the design will be able to run and, thus, on the length of the critical path. We found that the unmodified openMSP430 core can run at 51 MHz with our setup. For our modifications, there are two parameters that may influence the critical path: the security level (i.e., the size of the keys) and the number of supported modules N_{SM}. The reason these parameters may influence the critical path is the same for both. The keys are stored in the MAL circuits and routed to the crypto unit through a multiplexer. Both the key size and N_{SM} will increase the size of this multiplexer, and hence increase the length of the critical path.

Figure 6 shows the maximum obtainable frequency in function of N_{SM} for a number of different security levels. Note that although our implementation allows for security levels up to 256 bits, 128 bits are ample for our target platforms, and we therefore do not evaluate higher security levels.

The influence of the security level and N_{SM} should be clear from the figure. However, it should also be clear that the maximum frequency for any security level is not influenced much by values of N_{SM} up to 8 where it is always around 39 MHz. The reason for the maximum frequency for low numbers of N_{SM} being smaller than the unmodified core (51 MHz) is because of the MAL circuits (Figure 4), which add a number of comparators to the path of the memory address bus.

6.1.2. Microbenchmarks. To quantify the impact on performance of our extensions, we first performed microbenchmarks to measure the cost of each new instruction. To this end, we added a custom timestamp counter peripheral to the CPU core that allowed us to conveniently measure the amount of cycles passed since power up. It should also be noted that all measurements are completely noiseless and thus accurate. Consequently, it is not necessary to calculate an average value over multiple measurements.

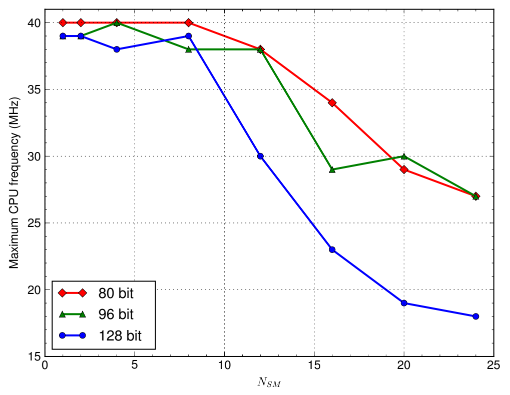

Fig. 6. The maximum frequency for which the core can be synthesized in function of the number of modules (N_{SM}) for a number of security levels. The maximum frequency decreases with N_{SM} due to the large multiplexer needed to get the module key out of the MAL circuits. This also explains why the maximum frequency decreases much faster when the key size is larger. Note that the unmodified openMSP430 core can be synthesized at 51 MHz.

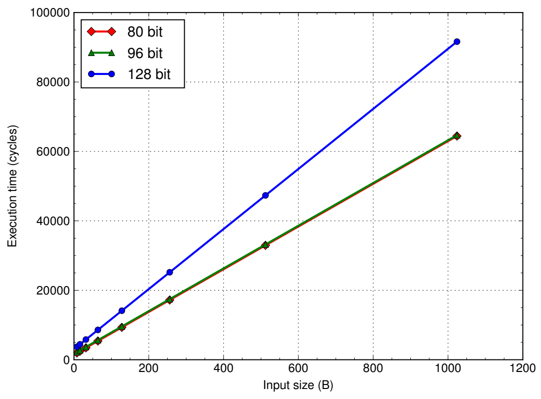

Fig. 7. Execution time of the `encrypt` instruction for a number of security levels. The cost of the other cryptographic instructions is similar with the exception of `enable` for confidential loading, which is twice as big. Notice how the performance for the 80- and 96 bit security levels is almost equal because they use the same SPONGENT variant.

The `get-id` and `unprotect` instructions are very fast: they both take one clock cycle. The other instructions perform cryptographic operations on their input, and hence their runtime cost depends linearly on the size of the input they handle. Remember that all cryptographic operations are implemented using the same underlying primitive (Section 4.1.2), which means their runtime cost is almost exactly the same. The only exception is the `enable` instruction when confidential loading is used. In this case the underlying primitive is called twice: once for decryption and once for key derivation. Its cost is therefore twice as big as the other instructions. Figure 7 shows the measurements for the `encrypt` instruction for a number of different security levels.

Note that the performance of the 80- and 96 bit implementations is almost the same. The reason for this is that the performance of our crypto unit is determined by the underlying SPONGENT variant. Recall that SPONGENT is defined in different variants and we select the correct variant for the required security level (Section 4.1.2). The 80- and 96 bit security levels require the same SPONGENT variant.

6.1.3. Macrobenchmark. To give an indication of the impact on performance in real-world scenarios, we performed the following macro benchmark. We synthesized our processor with 128-bit keys and configured it as in the example shown in Figure 3. 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:

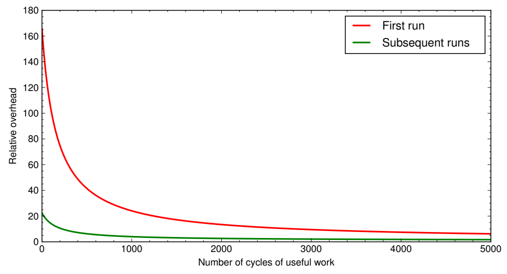

Fig. 8. Although the absolute overhead Sancus imposes on our example setup (Figure 3) is quite large, the relative overhead quickly drops when the modules perform some useful work. Notice how the subsequent runs are much faster than the first due to the optimization discussed in Section 3.7.

(1) the 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 encrypts its output together with the nonce. The overhead introduced by Sancus is due to a call to `attest` in step (2) and a call to `encrypt` 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 230 bytes, and that nonces and output data encrypted by M_i both have a size of 16 bits.

We measured the fixed overhead to be $26,834$ cycles for the first time data is requested from the module. Since the call to `attest` in step (2) is not needed after the initial verification (Section 3.7), we also measured the overhead of any subsequent requests, which is $3,481$ 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 8.

6.2. Area

We evaluated the area of our design using Synopsys Design Compiler v2013.12 with the UMC 130nm and NanGate 15nm standard-cell libraries. The default ASIC synthesis settings for the openMSP were used, except for disabling clock gating and DFT insertion. The unmodified openMSP430 core measures 11kGE and 15kGE, respectively, using these libraries. The area of Sancus in function of the number of modules N_{SM} for a number of security levels is shown in Figure 9. If computational overhead is of lesser concern, the area can be reduced by computing the module key on the fly instead of storing it in registers. Exploring other improvements is left as future work.

6.3. 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. [Agten et al. 2012; Patrignani et al. 2015] 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 [Erlingsson et al. 2010] to get unintended access to 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 [Younan et al. 2012].

The security of *remote attestation* and *secure communication* follows from the following key observation: the computation of MACs with the module key is only possible by

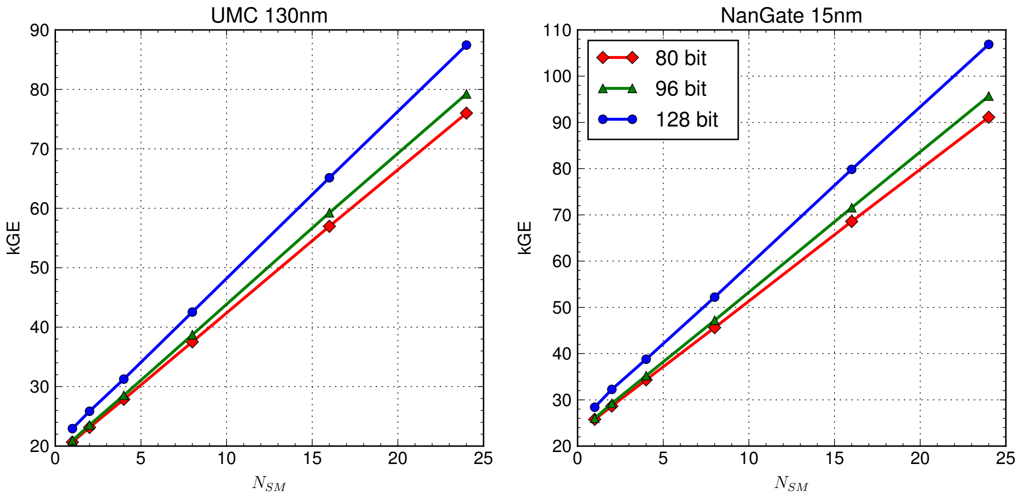

Fig. 9. The area of the whole openMSP430 core with Sancus extensions when synthesizing for different security levels using the UMC 130nm and NanGate 15nm standard-cell libraries. Synthesis was done for a target clock frequency of 25MHz.

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.

If a module has access to the correct identity of another module it wants to call, the security of *secure linking* follows from the definition of the `attest` instruction (Section 3.7). Indeed, this instruction will only succeed if a module with the given identity is enabled at the given location. This means that an attacker can only force the instruction to succeed by either (1) loading the correct module; or (2) constructing a different module with the same identity. The latter amounts to finding a hash collision, which our attacker model precludes.

The identity used for secure linking must not be stored in unprotected memory where an attacker can easily manipulate it. There are two options to provide the identity securely to a module. First, it can be stored in a module's text section. Although, if confidential loading is not used for this module, an attacker can manipulate the text section before protection is enabled,this manipulation will be detected when its provider performs remote attestation. Second, the identity can be sent using secure communication after deployment and stored in the module's data section. This is the technique that our implementation uses (Section 4.3).

The security of *confidential loading* follows from two observations. First, before the `enable` instruction is called, the module's text section is encrypted using the vendor key, which the attacker does not have access to. Second, after the instruction is finished, Sancus' access rules (Table II) will deny any access to the text section from outside the module. Therefore, only API-level attacks would enable an attacker to read (parts of) the text section of modules that use confidential loading.

Finally, *hardware breach confinement* follows from the fact that we use independent master keys on all nodes (Section 3.3).

7. 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. Below we discuss research that is more directly related to Sancus. For an extended comparison of hardware-based trusted computing architectures we refer the reader to [Maene et al. 2017].

7.1. Isolation in High-End Devices

The Multics [Corbato and Vyssotsky 1965] 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 [McCune et al. 2008], a system that relies on a TPM chip and trusted computing functionality of modern CPUs, to provide strong isolation of modules with a TCB of only 250 LOCs. Subsequent research [McCune et al. 2010; Azab et al. 2011; Sahita R 2009; Strackx and Piessens 2012] 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.

ARM TrustZone [Alves and Felton 2004] implements hardware based access control to use a physical core as two virtual processors so as to execute security critical applications in their own “world”, in isolation from the normal world. The secure world runs its own OS, libraries and applications, which mutually trust each other. TrustZone for the v8-M architecture [ARM 2016] employs a “secure gateway” instruction to enter the secure world at specific entry point addresses, providing similar but more coarsely-grained isolation properties than Sancus.

More recently, Intel started shipping x86 processors equipped with Software Guard Extensions (SGX) [McKeen et al. 2013] that allows the execution of security-critical code via hardware-enforced individually isolated *enclaves* in a shared address space, managed by an untrusted OS. SGX also provides functionality for local and remote attestation and for data sealing [Anati et al. 2013].

While the aforementioned architectures focus on isolating relatively small, security-sensitive application components, an alternative line of work seeks to protect largely unmodified legacy applications from an untrusted OS. Overshadow [Chen et al. 2008] and InkTag [Hofmann et al. 2013] employ a trusted hypervisor, whereas Haven [Baumann et al. 2014] leverages Intel SGX processor extensions to isolate application binaries. The untrusted OS continues to provide resource management and application services. However, due to the complex nature of legacy operating system interfaces and hardware, these designs are exposed to a new class of powerful side-channel attacks [Xu et al. 2015]. Compared to higher-end MMU-based systems, Sancus can be considered less susceptible to such threats considering the elementary design of its security extensions, as well as the underlying processor.

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 [Kostiainen et al. 2009]. 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.

7.2. 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 [Halperin et al. 2008] 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 or security considerations, as in early versions of TinyOS [Levis 2012]. 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 [Han et al. 2005] or Contiki [Dunkels et al. 2006]. 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 [Gu and Stankovic 2006] 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 [Kumar et al. 2007] through a combination of backend compile time rewriting and run time checking on the sensor. Safe TinyOS [Cooprider et al. 2007] 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 [Simon et al. 2006]. SenShare [Leontiadis et al. 2012] provides a virtual 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. [Strackx et al. 2010] introduced the idea of a program counter-based access control model, but without providing any implementation. Agten et al. [Agten et al. 2012] 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. El Defrawy et al. [Eldefrawy et al. 2012] implemented hardware support for allowing 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. TrustLite [Koeberl et al. 2014], on the other hand, features an Execution-Aware Memory Protection Unit (EA-MPU) that records program counter-based memory access rules in a configurable hardware table. Compared to Sancus, this allows for more complex policies, such as multiple private data sections per module, or protected data sharing between two or more modules. TrustLite, however, relies on a trusted Secure Loader software entity to initialize the EA-MPU table at boot time, and does not allow modules to be unloaded at run time. More recently, the TyTAN [Brasser et al. 2015] architecture extends TrustLite with dynamic loading, and local and remote attestation guarantees for isolated tasks from mutually distrusting stakeholders. Their approach to attestation resembles Sancus' in that they derive keys from task identities and a hardware-level platform key. In contrast to Sancus however, TyTAN relies on a trusted software runtime to measure task identities, and to guard inter-module authenticated communication.

Java Card [Oracle 2015; GlobalPlatform 2015] by Oracle is a smart card technology with applications in, e.g., mobile communications and electronic citizen IDs. Java Card provides an isolated environment, the Java Card VM and the Applet Firewall, to execute Java Card Applets in isolation from the operating system and other such applets. The technology features a range of symmetric and public key cryptographic algorithms but does not provide lightweight embedded ciphers such as Sancus' SPONGEWRAP [Bertoni et al. 2011]. Extensions for remote attestation are under development. Overall, Java Card can be used to implement many features of Sancus. Yet, Java Card has a larger hardware and software TCB and exhibits different performance characteristics than our approach, impeding deployment in some of the scenarios outlined in Section 5.

8. 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.

9. 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, binary packages and documentation can be found at https://distrinet.cs.kuleuven.be/software/sancus/.

References

Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. 2012. Secure Compilation to Modern Processors, In 2012 IEEE 25th Computer Security Foundations Symposium (CSF 2012). *Computer Security Foundations Symposium, IEEE* (2012), 171–185.

Tiago Alves and Don Felton. 2004. TrustZone: Integrated hardware and software security. *ARM white paper* 3, 4 (2004), 18–24.

Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. (2013).

Ross J. Anderson and Markus G. Kuhn. 1998. Low Cost Attacks on Tamper Resistant Devices. In *Proceedings of the 5th International Workshop on Security Protocols*. Springer-Verlag, London, UK, UK, 125–136.

Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. 2015a. *APE: Authenticated Permutation-Based Encryption for Lightweight Cryptography*. Springer Berlin Heidelberg, Berlin, Heidelberg, 168–186.

Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. 2015b. *Security of Keyed Sponge Constructions Using a Modular Proof Approach*. Springer Berlin Heidelberg, Berlin, Heidelberg, 364–384.

ARM. 2016. TrustZone technology for ARMv8-M Architecture Version 1.1. https://static.docs.arm.com/100690_0101/00/armv8_m_architecture_trustzone_technology_100690_0101_00_en.pdf. (2016).

AUTOSAR. 2015. AUTOSAR Specification 4.2. (2015). http://www.autosar.org/specifications/release-42/.

A.M. Azab, P. Ning, and X. Zhang. 2011. 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*. ACM, 375–388.

Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 267–283.

Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2011. Duplexing the Sponge: Single-pass Authenticated Encryption and Other Applications. In *Selected Areas in Cryptography*. Springer, 320–337.

Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. 2012. SPONGENT: The Design Space of Lightweight Cryptographic Hashing. *IEEE Trans. Comput.* 99, PrePrints (2012), 1.

Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 2001. On the Importance of Eliminating Errors in Cryptographic Computations. *J. Cryptology* 14 (2001), 101–119.

Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: Tiny Trust Anchor for Tiny Devices. In *Proceedings of the 52Nd Annual Design Automation Conference (DAC '15)*. ACM, New York, NY, USA, Article 34, 6 pages.

Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. 2009. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS '09)*. ACM, New York, NY, USA, 400–409.

Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 2–13.

Sara Cleemput, Mustafa A. Mustafa, and Bart Preneel. 2016. High Assurance Smart Metering. In *17th Int. Symposium on High Assurance Systems Engineering (HASE'16)*. IEEE, 294–297.

Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. 2007. Efficient memory safety for TinyOS. In *Proceedings of the 5th international conference on Embedded networked sensor systems (SenSys '07)*. ACM, New York, NY, USA, 205–218.

FJ Corbato and VA Vyssotsky. 1965. Introduction and overview of the Multics system. In *Proceedings of the November 30–December 1, 1965, Fall joint computer conference, part I*. ACM, 185–196.

Department of Energy and Climate Change. 2014. Smart Metering Implementation Programme – Smart Metering Equipment Technical Specifications; Version 1.58. (2014). https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/381535/SMIP_E2E_SMETS2.pdf.

Dominique Devriese and Frank Piessens. 2010. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy*. 109–124.

Danny Dolev and Andrew C. Yao. 1983. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–208.

Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06)*. ACM, New York, NY, USA, 15–28.

A. Dunkels, B. Gronvall, and T. Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. 455–462. http://www.contiki-os.org/.

Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. 2012. 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.

Úlfar Erlingsson, Yves Younan, and Frank Piessens. 2010. Low-Level Software Security by Example. In *Handbook of Information and Communication Security*. Springer.

Scott Erven and Shawn Merdinger. 2014. Just What the Doctor Ordered? *DEV CON 22* (2014).

Muhammad Omer Farooq and Thomas Kunz. 2011. Operating Systems for Wireless Sensor Networks: A Survey. *Sensors* 11, 6 (2011), 5900–5930.

M.C. Fernandez-Gago, R. Roman, and J. Lopez. 2007. A Survey on the Applicability of Trust Management Systems for Wireless Sensor Networks. In *Security, Privacy and Trust in Pervasive and Ubiquitous Computing, 2007. SECPerU 2007. Third International Workshop on*. 25–30.

Aurélien Francillon and Claude Castelluccia. 2008. Code injection attacks on Harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS '08)*. ACM, New York, NY, USA, 15–26.

Gartner. 2013. Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. http://www.gartner.com/newsroom/id/2636073. (2013).

Thanassis Giannetsos, Tassos Dimitriou, and Neeli R. Prasad. 2009. Self-propagating worms in wireless sensor networks. In *Proceedings of the 5th international student workshop on Emerging networking experiments and technologies (Co-Next Student Workshop '09)*. ACM, New York, NY, USA, 31–32.

Olivier Girard. 2016. openMSP430. http://opencores.org/project,openmsp430. (2016).

GlobalPlatform. 2015. GlobalPlatform Card Specification v2.3. http://www.globalplatform.org/. (2015).

Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix Freiling, and Ingrid Verbauwhede. 2015. Soteria: Offline Software Protection within Low-cost Embedded Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*.

Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. 2015. Security in the integration of low-power Wireless Sensor Networks with the Internet: A survey. *Ad Hoc Networks* 24, Part A, 0 (2015), 264–287.

Lin Gu and John A. Stankovic. 2006. t-kernel: providing reliable OS support to wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, Boulder, Colorado, USA, 1–14.

D. Halperin, T.S. Heydt-Benjamin, B. Ransford, S.S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W.H. Maisel. 2008. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. Ieee, 129–142.

Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. 2005. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services (MobiSys '05)*. ACM, New York, NY, USA, 163–176.

Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 265–278.

Paul C. Kocher. 1996. 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 (CRYPTO '96)*. Springer-Verlag, London, UK, UK, 104–113.

Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '99)*. Springer-Verlag, London, UK, UK, 388–397.

Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 10, 14 pages.

Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. 2009. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS '09)*. ACM, New York, NY, USA, 104–115.

Ram Kumar, Eddie Kohler, and Mani Srivastava. 2007. Harbor: software-based memory protection for sensor nodes. In *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN '07)*. ACM, New York, NY, USA, 340–349.

Yong Ki Lee, K. Sakiyama, L. Batina, and I. Verbauwhede. 2008. Elliptic-Curve-Based Security Processor for RFID. *Computers, IEEE Transactions on* 57, 11 (nov. 2008), 1514 –1527.

Frank Thomson Leighton and Silvio Micali. 1994. Secret-Key Agreement without Public-Key Cryptography. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '93)*. Springer-Verlag, London, UK, UK, 456–479.

Ilias Leontiadis, Christos Efstratiou, Cecilia Mascolo, and Jon Crowcroft. 2012. SenShare: transforming sensor networks into multi-application sensing infrastructures. In *Proceedings of the 9th European conference on Wireless Sensor Networks (EWSN'12)*. Springer-Verlag, Berlin, Heidelberg, 65–81.

Philip Levis. 2012. Experiences from a decade of TinyOS development. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 207–220.

Jay Ligatti, Lujo Bauer, and David Walker. 2005. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security* 4, 1-2 (2005), 2–16.

LLVM Developer Group. 2016a. Clang. http://clang.llvm.org/. (2016).

LLVM Developer Group. 2016b. LLVM. http://llvm.org/. (2016).

Javier Lopez, Rodrigo Roman, Isaac Agudo, and Carmen Fernandez-Gago. 2010. Trust Management Systems for Wireless Sensor Networks: Best Practices. *Comput. Commun.* 33, 9 (2010), 1086–1093.

P. Maene, J. Gotzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. 2017. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Trans. Comput.* PP, 99 (2017).

Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*. ACM, 315–328.

Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. 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 '13)*. ACM, New York, NY, USA, Article 10, 1 pages.

Charlie Miller and Chris Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* (2015).

Jan Tobias Mühlberg, Sara Cleemput, Mustafa A. Mustafa, Jo Van Bulck, Bart Preneel, and Frank Piessens. 2016. An Implementation of a High Assurance Smart Meter using Protected Module Architectures. In *WISTP '16 (LNCS)*. Springer, Heidelberg. To appear.

Jan Tobias Mühlberg, Job Noorman, and Frank Piessens. 2015. Lightweight and Flexible Trust Assessment Modules for the Internet of Things. In *ESORICS '15 (LNCS)*, Vol. 9326. Springer, Heidelberg, 503–520.

Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security symposium*. USENIX Association, 479–494.

Stefan Nürnberger and Christian Rossow. 2016. *– vatiCAN – Vetted, Authenticated CAN Bus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 106–124.

Oracle. 2015. Java Card Technology 3.0.5. http://www.oracle.com/technetwork/java/javacard/overview/. (2015).

Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, 2, Article 6 (April 2015), 50 pages.

Jerome Radcliffe. 2011. Hackng Medical Devices for Fun and Insulin: Breaking the Human SCADA System. *Black Hat USA* (2011).

Rodrigo Roman, Pablo Najera, and Javier Lopez. 2011. Securing the Internet of Things. *Computer* 44, 9 (2011), 51–58.

Dewan P. Sahita R, Warrier U. 2009. Protecting Critical Applications on Mobile Platforms. *Intel Technology Journal* 13 (2009), 16–35. Issue 2.

Hocheol Shin, Yunmok Son, Youngseok Park, Yujin Kwon, and Yongdae Kim. 2016. Sampling Race: Bypassing Timing-based Analog Active Sensor Spoofing Detection on Analog-digital Systems. In *WOOT 2016*. USENIX Association, Berkeley, CA, USA, 200–210.

Yasser Shoukry, Paul Martin, Paulo Tabuada, and Mani Srivastava. 2013. Non-invasive Spoofing Attacks for Anti-lock Braking Systems. In *CHES 2013 (LNCS)*. Springer, Berlin, Germany, 55–72.

Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. 2006. Java™ on the bare metal of wireless sensor devices: the squawk Java virtual machine.. In *VEE* (2006-12-15), Hans-Juergen Boehm and David Grove (Eds.). ACM, 78–88.

Raoul Strackx and Frank Piessens. 2012. 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),*. ACM Press, 2–13.

Raoul Strackx, Frank Piessens, and Bart Preneel. 2010. 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*, Vol. 50. Springer, 1–18.

Nicolas Tsiftes, Adam Dunkels, Zhitao He, and Thiemo Voigt. 2009. Enabling large-scale storage in sensor networks with the coffee file system. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*. IEEE Computer Society, 349–360.

Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2015. Secure Resource Sharing for Embedded Protected Module Architectures. In *WISTP '15 (LNCS)*, Vol. 9311. Springer, Heidelberg, 71–87.

Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2016. Towards Availability and Real-Time Guarantees for Protected Module Architectures. In *MASS '16, MODULARITY Companion Proceedings '16*. ACM, New York, 146–151.

J. Viega and H. Thompson. 2012. The State of Embedded-Device Security (Spoiler Alert: It's Bad). *Security Privacy, IEEE* 10, 5 (Sept.-Oct. 2012), 68 –70.

Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.

Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Comput. Surv.* 44, 3, Article 17 (June 2012), 28 pages.