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

1.1 Introduction

1.1.1 Contributions

- We propose Sancus, 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.
- We implement the hardware required for Sancus as an extension of a mainstream micro-processor, 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.

1.2 Background

1.2.1 System Model

A single infrastructure provider (IP), owns and administers a set of microprocessor-based systems that we refer to as nodes (N). A variety of third-party software providers (SP) are interested in using the infrastructure provided by IP. They do so by deploying software modules (SM) on the nodes administered by IP.

1.2.2 System Requirements

Any system that supports extensibility by several software providers must implement measures to make sure that the different modules cannot interfere with each other in undesired ways. For high- to mid-end systems, this problem is relatively well-understood, and good solutions exist. Two important classes of solutions are the use of virtual memory, and the use of a memory-safe virtual machine.

1.2.3 System goals

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.

1.2.4 Attacker Model

We consider attackers with two powerful capabilities. First, we assume attackers can manipulate all the software on the nodes. Second, we assume attackers can control the communication network that is used by software providers and nodes to communicate with each other. 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, attackers cannot break cryptographic primitives, but they can perform protocol-level attacks. Finally, attacks against the hardware of individual nodes are out of scope.

1.2.5 Security Properties

- Software module isolation
- Remote attestation
- Secure communication
- Secure linking
- Confidential deployment
- Hardware breach confinement

1.3 Design

1.3.1 Main Challenge

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.

1.3.2 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. 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.

1.3.3 Key Management

We handle key management without relying on public-key cryptography. 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:

- 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.

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.

1.3.4 Memory Access Control

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 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. 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.

1.3.5 Remote Attestation & Safe Communication

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. This gives the SP assurance that the correct module is running on that node at this point in time.

1.3.6 Processor Isolation Implementation

For isolation, 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. 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.

1.3.7 Cryptographic Implementation

We have chosen to build these cryptographic primitives on the SpongeWrap authenticated encryption construction using spongent as the underlying sponge function. Since keyed sponge functions are shown to be pseudorandom functions, 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.

1.3.8 Compiler Entrypoint Implementation

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.