POLITECNICO DI TORINO

Department of Control and Computer Engineering

Securing Smart Environments with Authentic Execution

Candidate:
Gianluca Scopelliti

Thesis advisor:
Prof. Cataldo Basile

Supervisors:
Prof. Frank Piessens
Dr. Jan Tobias Mühlberg
Ir. Fritz Alder

Master's Degree in Computer Engineering

Academic Year 2019-2020

Acknowledgements

Firstly, I would like to thank Prof. Piessens, who was the first person I came into contact with at KU Leuven. Without his help, I would not have been able to pursue this interesting topic. Secondly, I would like to express my gratitude to my daily advisors JT and Fritz, who have been my guidance throughout the entire year. Despite the pandemic, they always have been available via email and online meetings, allowing me to reach my goal. Thanks also to Prof. Basile, my advisor at Politecnico, for his availability and willingness to help and give suggestions, even though the distance.

I would also like to thank my family and friends for all their support, and to all the people who have always believed in me and my abilities, sometimes even more than myself. Special mention to Cif, my pandemic companion. Finally, I would like to thank my old self, for giving me the opportunity to enjoy this Erasmus experience and opening me the gates to the future.

Abstract

Securing Smart Environment applications is a huge concern nowadays. The use of small, embedded devices to connect the physical to the digital world is a big challenge in terms of security, due to the limited hardware resources available and the real-time constraints the system must ensure. As an example, a paper released in 2018 by the US Department of Homeland Security highlights numerous, potential issues and threats regarding the use of new Internet of Things (IoT) technologies in Precision Agriculture and Smart Farming.

To address this problem, the concept of *Authentic Execution* was proposed in 2017 by Noorman et al., which described a secure framework for a distributed, event-driven application. This solution relied on the use of Trusted Computing (TC) and Trusted Execution Environments (TEE) to achieve strong security properties such as confidentiality, integrity and authenticity of software and data. An implementation of this approach was provided for Sancus, an embedded TEE that extends the TI MSP430 CPU.

However, this solution is not sufficient in a real scenario: in fact, most IoT systems need a cloud backend, to gather data and perform expensive computation. Hence, an *heterogeneous* system is needed in such applications, composed by both embedded devices and remote servers.

Therefore, this Master's Thesis describes an implementation of the *Authentic Execution* approach for Software Guard eXtensions (SGX), a TEE included in recent Intel processors. We developed a framework that allows a developer to write only the logic of his own application, as well as providing a high-level description of the system (e.g., to specify how the modules are connected to each other). The framework implicitly handles the execution of a module inside an isolated, trusted environment (called *enclave*) and the communication between different modules. Moreover, the framework is entirely written in Rust, a modern, fast programming language that provides by design numerous features to enhance security, such as protection against many memory-management vulnerabilities (e.g., buffer overflows), as well as a safe use of threads and concurrency. Along with the SGX implementation, we provide tools to easily deploy a heterogeneous, distributed application on a shared infrastructure.

In addition to the *Trusted Execution* of their modules, Sancus and SGX can bring further advantages if used together: while the former is able to perform *Secure I/O*, the latter provides a feature called *data sealing*, to securely store data on disk. Particularly, Secure I/O is used to establish *trusted paths* between high-end computation nodes and I/O devices.

Finally, we prove the effectiveness of *Authentic Execution* by implementing a prototype for a smart irrigation system. A security evaluation shows that this approach ensures strong confidentiality and integrity guarantees. From a performance point of view, instead, tests reveal that our solution is widely acceptable for an irrigation system, whereas it might not be feasible for applications with stricter real-time constraints.

Keywords: IoT, Authentic Execution, Trusted Computing, TEE, Sancus, Intel SGX, Rust, Smart Environments, Precision Agriculture, Smart Farming

Contents

List of Tables

List of Figures

List of Source Codes

List of Acronyms

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

CAN Controlled Area Network

DCAP Data Center Attestation Primitives

ECDSA Elliptic Curve Digital Signature Algorithm

EDP Enclave Development Platform

EM Event Manager

EPC Enclave Page Cache

EPCM Enclave Page Cache Map

EPID Enhanced Privacy ID

GCM Galois/Counter Mode

I/O input/output

IAS Intel SGX Attestation Service

IoT Internet of Things

IP Infrastructure Provider

ISA Instruction Set Architecture

MAC Message Authentication Code

MCU Microcontroller Unit

MEE Memory Encryption Engine

MMIO Memory-Mapped I/O

OS Operating System

PA Precision Agriculture

PC Program Counter

PRM Processor Reserved Memory

QE Quoting Enclave

RA Remote Attestation

RAII Resource Acquisition Is Initialization

ROP Return-Oriented Programming

RTT Round-Trip Time

SFT Smart Farming Technology

SGX Software Guard eXtensions

SLOC Source Lines of Code

SM Software Module

SP Software Provider

TC Trusted Computing

TCB Trusted Computing Base

TCG Trusted Computing Group

TEE Trusted Execution Environment

TPM Trusted Platform Module

UART Universal Asynchronous Receiver-Transmitter

Chapter 1

Introduction

In recent years, more and more applications have been developed to connect the physical to the digital world, with the purpose to improve human's everyday activities. These applications rely on input/output (I/O) devices to interact with the real world in both directions: input peripherals (sensors) are used to get information from the environment (e.g., light, temperature, etc.), whereas output devices (actuators) perform actions on it (e.g., moving a robot). Systems that use this kind of technologies are called *Smart Environments* [1]. A Smart Environment application consists of several components ranging from low-end I/O devices to high-end computation systems. All of them work together, exchanging information to each other. Hence, they must be part of the same local network, which can be either wireless or wired. The running application is then *distributed*, as it is scattered among multiple components.

These applications may bring huge benefits for their stakeholders, e.g., an improved quality of life or financial profits. However, from a security point of view there are some challenges to be addressed. Since the application is distributed, each component needs to be reachable through well-defined interfaces, which might be exploited by an adversary to take control of the system. In addition, some systems require an Internet connection to work, making these interfaces accessible even from a remote location, increasing the risk of illegal accesses. Moreover, attackers might tamper with the communication channel between two components, causing malfunctions or improper behavior. Providing security guarantees in a Smart Environment application is essential, as a successful attack might cause huge damages to the physical world. For instance, an improper use of a robot might harm not only the environment, but human lives as well.

To solve these issues, several security strategies can be applied. The ultimate goal is to provide confidentiality, integrity and authenticity guarantees of (1) the execution of every component of an application, (2) the communication between two different components and (3) the interaction between a component and an I/O device. Regarding (1), a software on a platform must be run in an isolated context, which should protect code and data from being accessed or modified by other entities. Besides, the software needs to be authenticated at runtime by a remote party (e.g., the developer or another software), to prove its trustworthiness. Finally, an effort must be made to minimize the Trusted Computing Base (TCB), which is the set of all the hardware and software that is considered critical to an application from a security perspective. Usually, an application needs to trust its own code, the underlying Operating System (OS) and the hardware. In this situation, a vulnerability found anywhere in the platform might affect the execution of the application, if exploited by an attacker. Hence, it is essential to reduce the attack surface as much as possible; a smaller TCB makes also easier to check for vulnerabilities. Concerning (2), instead, cryptographic operations can be used to secure the communication between two entities

on a network. Similarly, sensitive data must be protected in (3) as well. Moreover, an I/O device should be accessed and/or controlled only by authorized software on the same platform.

Basing on these security requirements, a framework called *Authentic Execution* was proposed in 2017 by Noorman et al. [2], to provide strong assurance of the secure execution of a distributed, event-driven application. The framework implements the principles of *Trusted Computing (TC)* [3]: each component of the distributed application runs inside an isolated portion of memory called *enclave*, which resides in a secure area of the processor called Trusted Execution Environment (TEE) [4]. Thanks to the hardware support, code and data inside the enclave are not accessible from the outside and the execution of a software cannot be modified at runtime. Furthermore, the owner of the application can obtain a proof of authenticity of each component by performing the Remote Attestation (RA) process [5]. If RA succeeds, the owner has the guarantee that the application is running untampered and the isolation mechanisms of the TEE are in place. The Authentic Execution framework also protects the communication channel with encryption techniques and supports *Secure I/O*.

The Authentic Execution concepts can be applied to a generic TEE; however, the implementation provided by Noorman et al. only includes support for Sancus, an embedded TEE that extends the TI MSP430 CPU [6], limiting the applicability of the framework in a real scenario. In fact, a Smart Environment is often made of heterogeneous components, as discussed above. High-end nodes are essential in most applications, as they provide the hardware and software resources needed to perform highly expensive tasks, such as running a machine learning algorithm or storing huge quantities of sensor data. An application composed of only embedded devices might work on small systems, but is likely to be infeasible on more sophisticated scenarios.

Therefore, this Master's Thesis aims to remove this limitation by providing the Authentic Execution framework support for SGX, a TEE included in recent Intel processors [7]. By combining Sancus and SGX together, the framework becomes able to support a much higher number of real-world applications: in particular, this Thesis focuses on Smart Farming [8], developing a prototype for a secure smart irrigation system. Smart Farming applications are critical from a security point of view; a study conducted by the US Department of Homeland Security [9] shows that many threats are related to the adoption of new digital technologies in Precision Agriculture (PA). Hence, a secure infrastructure is essential to deploy an application that is robust and resilient to attacks.

1.1 Contributions

The main contributions we provide in this Master's Thesis work are:

1. An implementation of the Authentic Execution framework for Intel SGX, along with modifications of the existing Sancus code to provide compatibility between the two TEEs.
2. The development of a prototype for a secure smart irrigation system, implemented using the upgraded Authentic Execution framework.
3. An evaluation of performance, size and security aspects of our prototype implementation.

All source code and documentation are available at `https://github.com/gianlu33/authentic-execution`.

1.2 Outline

The remainder of this Master's Thesis is structured as follows:

Chapter 2: Background. This chapter gives preliminary information about the key topics of this work. Firstly, it discusses technical details of Trusted Computing and TEEs. Secondly, it introduces Smart Farming.

Chapter 3: Problem Statement. This chapter defines the system model, the attacker model and the security properties we aim for. In addition, we provide an extensive description of the Authentic Execution framework.

Chapter 4: Implementation. This chapter focuses on the technical aspects of our work, describing how we implemented the Authentic Execution framework in Intel SGX and what changes we made on the existing Sancus implementation and the tools to deploy and manage a system. Furthermore, it shows a design of the application-layer protocol that the components of our system use to communicate to each other.

Chapter 5: Prototype. This chapter presents the prototype we developed, which is based on the architecture illustrated in the previous chapters. It also includes an evaluation of both size and performance of the prototype based on experimental results, along with a discussion about its security properties and feasibility in a real-world scenario.

Chapter 6: Conclusion. This chapter summarizes the Thesis and introduces similar work in the research world. Moreover, it points out potential future improvements of our solution.

Chapter 2

Background and related work

This chapter aims to provide background information about the most important topics of this Master's Thesis and introduces similar work in the research world. It is structured as follows: Section 2.1 introduces the concepts of Trusted Computing (TC). Section 2.2 describes what is a Trusted Execution Environment (TEE) and how it works, with a special focus on Intel Software Guard eXtensions (SGX) and Sancus. Section 2.3, instead, discusses about Smart Farming applications and their security threats. Finally, section 2.4 concerns related work.

2.1 Trusted Computing

Trusted Computing (TC) [3, 10, 11] is a technology developed by the Trusted Computing Group (TCG) [12]. Its goal is to provide strong guarantees that a software always behaves in the expected way, as specified in the source code and regardless of other *untrusted* software in the system (e.g., the OS and other processes). Hardware extensions are introduced to enforce this behavior and to provide cryptographic capabilities, protecting the software against unexpected changes and attacks.

The key concepts of TC are the following:

- *Endorsement key.* The endorsement key is an unique, private key that is stored on a dedicated chip called Trusted Platform Module (TPM) [13] and never leaves it. Neither the OS nor any other software can obtain the key from the TPM. It is created randomly at manufacture time and cannot be changed. The corresponding public key is used for either attestation (i.e., to verify the integrity and authenticity of a software) or encryption of sensitive data sent to the chip.

- *Memory curtaining.* Memory curtaining ensures strong isolation of sensitive areas of memory, e.g., locations of cryptographic keys. This isolation is enforced by the hardware. Unauthorized accesses of this memory are not possible.

- *Secure I/O.* Secure input/output (I/O) ensures the confidentiality and integrity of I/O peripherals connected to the hardware. This feature rules out a wide variety of threats, such as keyloggers (i.e., hardware or software that records what the user types on the keyboard), introduction of fake data, manipulation of input data, record or alteration of the content displayed on the screen, etc.

- *Sealed storage.* The sealing feature allows a device or software to store securely private information on disk. The stored data is encrypted with a *sealing key*, which is included in

the TPM along with the endorsement key. The data is bound to the sealer (e.g., the device or a software), which is the only entity able to access it.

- *Remote attestation.* Remote Attestation (RA) [5] is the process in which an entity is authenticated to a remote host. For instance, a vendor might want to attest his software running on a remote machine. During the process, the software sends to the vendor a certificate, which is generated by the hardware and provides information about the software's current status. The certificate is digitally signed with the endorsement key, so that the vendor has a guarantee of its authenticity and can verify that the software is running untampered on the machine. During RA, a symmetric key might also be established between the vendor and the software, to protect future communications between the two.

- *Trusted third party.* RA does not provide anonymity, i.e., the attested entity needs to be identified in order for the attestant to verify its authenticity. This is not an issue in the case explained above, where a vendor wants to attest his own software. However, for a communication between two remote users this might be a problem, since one of them (or both) might want to remain anonymous. In these situations, a trusted third party can be used to guarantee anonymity on the network. In this scenario, the users are attested by the trusted third party itself: for each user, it performs the RA process and generates a signed certificate saying that the user is running untampered, without revealing his real identity. Then, this certificate is used by a user to prove his authenticity to the others.

From this model, different architectures have been designed in recent years, which implement a subset of these concepts. In the next section, we focus on a particular technology: Trusted Execution Environments.

2.2 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a secure area of the processor. An application that runs inside a TEE is protected with respect to confidentiality and integrity. Code and data of an application are loaded inside an *enclave*, which is a container that isolates the application from the rest of the world. As such, an external entity cannot access nor modify the memory inside the enclave. This behavior is enforced by the hardware (and, in some cases, the software as well): each TEE uses different mechanisms to achieve this goal.

One of the main benefits of a TEE is the reduction of the TCB, which is the set of all the hardware and software that is considered critical to an application from a security perspective. Normally, an application needs to trust all the infrastructure in which is executed: hardware, OS and other software such as system utilities. In this scenario the attack surface is huge, because a vulnerability found anywhere in the infrastructure might potentially affect the application, if exploited by an attacker. Thanks to the enclaved execution inside a TEE, instead, the application need not trust any other component except for its code and the hardware that enforces the isolation mechanisms. This results in a reduction of the TCB, which might be more or less significant depending on the infrastructure and the size of the application. For instance, the Linux kernel is estimated to have around 27.8 million lines of code[1]: in this case, the TCB reduction would be enormous. Although the TEE protects an enclave from the rest of the world, it cannot protect from vulnerabilities found in the enclave itself. Thus, it is critical from a

[1] https://www.theregister.com/2020/01/06/linux_2020_kernel_systemd_code

security point of view to write code that is bug-free, using safe programming languages and/or formal verification tools to ensure its correctness.

Depending on the TEE, additional features from the Trusted Computing model may be provided, such as secure I/O and data sealing. A feature common to all TEEs, instead, is RA, which is a key concept of this Master's Thesis work. In the next sub-sections we introduce Sancus and SGX, describing which features they provide and how RA is achieved in each of them.

2.2.1 Sancus

Sancus [6] is a TEE designed for low-cost and low-power embedded systems. Its implementation is based on the TI MSP430 processor, which uses a Von Neumann architecture with a single address space for instructions and data. Sancus offers a hardware-only TCB, which means that a protected software needs only to trust the hardware and its own code.

Sancus uses symmetric key cryptography to provide its security properties. Three main primitives are implemented: firstly, a hash function to calculate digests of data. Secondly, a key derivation function used to obtain a derived key $K_{M,D}$ from a master key K_M and diversification data D. Thirdly, an Authenticated Encryption with Associated Data (AEAD) primitive to provide confidentiality, integrity and authenticity guarantees on the data. This primitive consists of two functions, one for encryption and the other for decryption. The former takes as input a key K, a plaintext P and associated data A and returns the ciphertext C and a tag T, which is a Message Authentication Code (MAC) over the plaintext and associated data. The latter takes as input K, C, A and T and, if the MAC is correct, returns the plaintext P. If the MAC is not correct, instead, it produces a decryption error. The cryptographic functions are based on the SpongeWrap authenticated encryption scheme [14], using *SPONGENT* as the underlying sponge function [15].

Infrastructure and key derivation

The Sancus infrastructure consists of three categories of entities:

- *Nodes.* Nodes are the Sancus microcontrollers that are owned by an Infrastructure Provider (IP). Each node shares with the IP an unique master key K_N, which is stored in hardware inside a protected storage and is not directly accessible to software.

- *Software Providers.* A Software Provider (SP) is an entity that can deploy software on the IP nodes and is identified by a public ID. The IP uses the key derivation function to compute $K_{N,SP}$ from a node's key K_N, and it is used to derive future symmetric keys shared between the SP and his software deployed on N, as described below.

- *Software Modules.* A Software Module (SM) is the terminology Sancus uses to describe an enclave. It is a software developed by a SP and executed on a node N. Its binary file contains at least a text section and a data section, which is zero-initialized before execution. Each SM has an identity called *layout*: the layout of a SM is the hash of the text section and the start/end addresses of the text and data sections. That is, two SMs compiled from the same code would have different identities. The layout of a SM is used as diversification data to derive the module key $K_{N,SP,SM}$ from the SP key $K_{N,SP}$. This key is computed by the node and stored in the protected storage. Additionally, the SP can also derive this key on his own, since he knows the identity of the SM as well. Hence, the SP can use this key to attest a SM and to secure the communications with it.

Isolation and memory access control

Sancus uses a *program counter-based memory access control* [16] to protect a module inside a node. In short, this design makes sure that the data of a module is accessible only if the Program Counter (PC) is on the text section of the same module. That is, other modules cannot access that data because the access control system would block every illegal behavior. Furthermore, the cryptographic instructions that rely on $K_{N,SP,SM}$ depend on the value of the PC to retrieve the module key from the protected storage.

The text section of a module is read-only and can be accessed only through a well-defined entry point. This design choice was implemented to prevent Return-Oriented Programming (ROP) attacks [17]. However, multiple logical entry points can be defined in a module's code: the compiler implements them on top of the single physical entry point, by means of a jump table. To enable the memory access control on a module, the instruction `protect layout, SP` needs to be called. This instruction checks the layout of a module to be sure that it does not overlap with other modules, then enables the memory access control and finally creates the $K_{N,SP,SM}$ key, storing it in the protected storage together with the layout of the module and its ID. There is an alternative version of this instruction, which accepts an encrypted layout (encrypted with $K_{N,SP}$) along with a MAC to provide integrity. This procedure is also called *Confidential Loading*, as the text section of a module is not exposed during the deployment. To disable the memory access control, the instruction `unprotect continuation` is issued, which also clears the code and data sections of the module to avoid leaks.

Remote Attestation

RA in Sancus is a simple process that does not require particular effort from the involved parties (the SP and his module). Since the $K_{N,SP,SM}$ key is only known by the module and the SP, the latter can be sure that the former is running untampered on node N if a message sent from the module and encrypted with its master key can be successfully decrypted by the SP. Of course, the SP needs to trust the Sancus architecture, which must make sure that the key $K_{N,SP,SM}$ cannot be used by any other module. To prevent replay attacks, the RA process should use a nonce to provide freshness. The procedure is similar to a challenge-response protocol: the SP sends a fresh nonce to the module, which calls the encrypting function to compute a MAC of the nonce. Then, the MAC is sent back to the SP, which performs the integrity check that attests the module.

Secure I/O

An important feature provided by Sancus is *Secure I/O*. Sancus uses Memory-Mapped I/O (MMIO) to communicate with its devices, which means that the same address space is used for both memory and I/O devices. Hence, to gain exclusive access to a device, it is sufficient to map the data section of a module over the MMIO region of the device. As specified above, the memory access control makes sure that the data section of the module can only be accessed if the PC is on its text section. Thus, nobody except for the module itself would be able to access the I/O device.

However, this implementation contains a drawback: since the data section needs to be contiguous and the MMIO region is fixed, a module can use its private data section either for MMIO or data but not for both. As a consequence, if the data section is mapped on the MMIO region, the module cannot use any memory, including the stack. [2] proposes a solution of this issue, by using two separate modules: a *MMIO* module that is connected to the device (using

the method discussed above) and a *driver* module that implements the logic to use the device. When a module's entry point is called, the hardware provides information about the ID of the caller module; hence, the MMIO module is able to accept only requests coming from the driver module, discarding all the others.

2.2.2 Intel SGX

Software Guard eXtensions (SGX) [7, 18] is a TEE introduced in 2015 with the sixth generation of the Intel Core microprocessors based on the Skylake architecture. It consists of a set of of instructions added in the Intel Instruction Set Architecture (ISA), along with some hardware modifications. The key concept of SGX is the *enclave*, a protected area in the application's address space which provides confidentiality and integrity guarantees. The new instructions manage enclaves, including procedures to create an enclave, to load the enclave inside protected memory, to enter/exit the enclave, to perform security operations such as the generation of symmetric keys and to do other activities such as debugging or paging.

Isolation

A portion of the main memory called Processor Reserved Memory (PRM) is reserved for the enclaves and protected from external hardware or software accesses. This memory includes the Enclave Page Cache (EPC), which is the region where enclaves are loaded and is divided in chunks of four kilobytes (pages). An enclave might reserve one or more pages in the EPC. Additionally, the Enclave Page Cache Map (EPCM) is a data structure inside the PRM that contains metadata attached to each EPC page. This information is needed by the hardware to protect enclave memory accesses: among the other fields, it includes the identity of the enclave that owns the page and a value that shows whether the page is valid or not. Since the EPC has limited size (typically between 64 and 128 MB), there are mechanisms to evict and reload pages to/from memory outside the PRM. Naturally, the pages stored outside the PRM are encrypted to to preserve confidentiality and integrity of enclaves' code and data. Furthermore, the traffic between EPC and processor needs to be protected as well: this operation is performed by another hardware unit called Memory Encryption Engine (MEE).

Enclave identities

The SGX architecture provides each enclave two measurement registers, called *MRENCLAVE* and *MRSIGNER*. *MRENCLAVE* is a SHA-256 digest computed over the code and data of the enclave, the relative position of the pages in the enclave and the security properties of each page. *MRSIGNER* is a hash of the public key of the *Sealing Authority*, which is the entity that signs the enclave before distribution (normally, the enclave builder). If multiple enclaves are signed by the same Sealing Authority, they will have the same *MRSIGNER* value. This is particularly important for data sealing (as explained below), allowing multiple enclaves to share sealed data.

Local Attestation

Local Attestation is the process in which two enclaves on the same platform can attest each other. It requires a particular structure called *REPORT*, which can be issued by the hardware using the `EREPORT` instruction. A *REPORT* contains the two identities of the enclaves and their attributes (which are properties established at loading time), additional data to be sent to the target enclave, a proof of trustworthiness of the hardware and a MAC over the whole structure.

The MAC is computed using the *Report Key*, which is a key known only by the target enclave and the `EREPORT` instruction. The target enclave can retrieve this key by calling `EGETKEY`, the instruction that provides an enclave symmetric keys for encryption and authentication.

Attestation between two enclaves A and B involves three steps: the following procedure shows a scenario in which B sends the first message, however the process is symmetric. In the first step, B sends its *MRENCLAVE* value to A in clear. The second step requires A to invoke `EREPORT`, passing B's *MRENCLAVE* as parameter. The *REPORT* generated is then sent to B. In the third step, B verifies the *REPORT* by computing a MAC over the structure (using the Report Key obtained with `EGETKEY`) and comparing it with the provided MAC inside the *REPORT*. If the MACs are the same, B obtains a proof that A is an enclave running untampered on the same platform. Then, B requests a *REPORT* destined to A, which attests B using the same procedure.

Remote Attestation

RA allows a remote provider to acquire the proof that a software is running untampered inside an enclave, on an updated SGX system. After the enclave is attested, an encrypted communication channel can be established between it and the provider, to protect future transmissions of sensitive data. There are currently two types of RA supported by Intel: a service that uses Intel SGX Enhanced Privacy ID (EPID) and an Elliptic Curve Digital Signature Algorithm (ECDSA)-based RA that uses third-party attestation with Intel SGX Data Center Attestation Primitives (DCAP). In our work we used the former, hence only this technology is explained in this section.

Intel EPID is a group signature scheme that allows a platform to cryptographically sign objects without losing its privacy. With Intel EPID, each platform (also called signer) belongs to a group; the signer uses its own private key for signing, but a remote verifier uses the group's public key to verify a signature. Therefore, a verifier cannot associate a signature to a specific member of the group, which as a consequence remains anonymous.

A fundamental component of the RA process is the Quoting Enclave (QE), which is a trusted enclave provided by Intel that resides on the same platform as the target enclave. The QE verifies *REPORTs* created with its *MRENCLAVE* value, attesting (locally) the enclaves who generated them. Additionally, it converts and signs each *REPORT* with the asymmetric EPID key, creating a *QUOTE*. Only the QE has access to the EPID key, hence the *QUOTE* can be seen to be issued by the processor itself.

In short, the RA process works in the following way: the service provider (i.e., the attestant) sends a challenge to the enclave, containing a nonce for liveness purposes. Then, the enclave calls the `EREPORT` instruction to generate a *REPORT* destined to the QE, which includes the response to the challenge and optionally an ephemerally generated public key which might be used later by the provider to transmit secrets to the enclave. The QE attests the enclave, generating a *QUOTE* as described above, which is sent back to the provider. Finally, the provider validates the signature of the *QUOTE*, either directly with the EPID public key or using an external EPID verification service to perform the validation, such as the Intel SGX Attestation Service (IAS) [19]. RA succeeds if the *QUOTE* validation passes and the response to the challenge is correct.

Data Sealing

Sealing [20] is a particular SGX feature that allows data to be stored on disk in a secure way. In essence, an enclave can use the `EGETKEY` instruction to obtain a *seal key*, which is used to encrypt and authenticate the data before storing it. There are two different policies that can be used:

Sealing to the Enclave Identity and Sealing to the Sealing Identity. In the former case, `EGETKEY` returns a key based on the enclave's *MRENCLAVE* value. This results in a different key for each enclave. Additionally, different versions of the same enclave will have different seal keys. This policy might be used to store enclave's sensitive data such as symmetric keys. In the latter case, `EGETKEY` returns a key based on the enclave's *MRSIGNER* value. Since the *MRSIGNER* value depends on the Sealing Authority, it is possible for multiple enclaves (or different versions of the same enclave) to retrieve the same seal key. This policy might be used to share data between enclaves or to allow offline migration of sealed data between enclave versions.

2.3 Smart Farming

According to [8], Smart Farming Technologies (SFTs) are technologies used in Precision Agriculture (PA) [21] for data acquisition, data analysis and evaluation, and precision application. Benefits of SFTs are: a more efficient application of inputs (seeds, fertilizers, water, etc.), an increased work speed and a reduced human intervention. This leads to an improved quality and quantity of the production (which means more profits), while at the same time enhancing the farmer's comfort and lifestyle.

SFTs are divided into three main categories:

- *Data acquisition technologies.* This category includes all surveying, mapping, navigation and sensing technologies. Their purpose is to gain knowledge about the field, crop or livestock. Applications might include soil mapping, RGB or thermal cameras, soil moisture sensors, etc.
- *Data analysis and evaluation technologies.* These SFTs are used to analyze the data obtained from the data acquisition SFTs, to compute statistics and make decisions. These technologies range from simple decision support systems to complex farm management and information systems.
- *Precision application technologies.* This category contains the SFTs used to apply the decisions taken using the other two categories. Applications might include automatic fertilization of fields, smart planting/seeding, precision irrigation systems, etc.

Essentially, a Smart Farming application consists of many different SFTs connected to the same (mainly wireless) network, each of which responsible to perform a specific task. The infrastructure is heterogeneous, as sensors/actuators are deployed on low-power embedded systems, whereas data analysis and evaluation SFTs run on high-end platforms. The system needs to be robust, as a Smart Farming application is meant to be used for many years under harsh outside conditions. There are also real-time constraints, which are more or less strict according to the task that needs to be done. For instance, piloting a fly surveillance drone has much stricter real-time requirements than fertilizing a field.

2.3.1 Security threats

In 2018, the US Department of Homeland Security released a paper called *Threats to Precision Agriculture* [9]. This paper contains a study about potential issues and threats regarding the use of new IoT technologies in crop and livestock production. As a matter of fact, these SFTs need to be connected together on the same local network (which, in some cases, is also connected to the Internet), which inevitably increases the attack surface for threat actors. The paper distinguish between three categories of threats:

- *Threats to Confidentiality.* This category covers data privacy and includes attacks aimed to access and steal sensitive data from data acquisition SFTs or data analysis and evaluation SFTs. For instance, an attacker might intercept sensitive data sent from a sensor to another SFT.
- *Threats to Integrity.* Integrity threats concerns the authenticity of software and data. Potential attacks include falsification of sensor data, introduction of rogue data on the network and manipulation of decision support systems. All of these attacks result in wrong decisions taken by the data analysis and evaluation SFTs which, as a consequence, cause an improper use of precision application technologies. Attacks that directly tamper with the application SFTs are possible as well.
- *Threats to Availability.* This category includes attacks aimed to affect the availability of the SFTs and/or the network. For instance, if an adversary wants to disrupt the water supply of the crop, he might perform different kinds of attacks, ranging from disabling the irrigation system to intercepting and dropping all messages exchanged on the network.

These threats can cause a dramatic impact on the farmer, not only financial but also reputational and emotional. Additionally, a Smart Farming application is also critical from a national security perspective. [9] highlights the fact that foreign governments could take an advantage over a country by exploiting these vulnerabilities, which might cause disruption of food/water supplies, identification of critical infrastructures, etc. Hence, countermeasures need to be adopted in order to drastically reduce the attack surface. The paper also describes in detail eight threat scenarios that might affect a Smart Farming system: these scenarios are analyzed and discussed in Section 5.6, during the security evaluation of the prototype we developed in this Master's Thesis work.

2.4 Related work

Several solutions exist in the research world, aimed to establish a trusted path between high-end nodes and I/O devices. BitE [22] provides confidentiality and integrity of the communication between the host platform and I/O devices, with mobile clients placed in between that perform cryptographic operations and attestation. However, data inside the host platform is not isolated, hence the OS kernel must be part of the TCB. Bumpy [23], a subsequent work from the same research group, removes this limitation by relying on the Flicker TEE [24] and using encryption-capable I/O devices to establish secure paths with an application. More recently, research has been made to provide support for Secure I/O in SGX. SGXIO [25] offers a solution of this problem by using a trusted hypervisor and secure I/O drivers. A similar approach was adopted in [26], with the implementation of a security microkernel called seL4. SGX-USB [27], instead, places a proxy device between the host and an I/O peripheral connected through USB, to establish a secure path to a SGX enclave. Unlike the two other approaches, SGX-USB does not rely on software modifications in the host platform.

In the next sub-section we focus on Fidelius, the most recent approach that is comparable to our work.

2.4.1 Fidelius

In 2019, Eskandarian et al. presented Fidelius [28], a solution to protect user secrets during web browsing sessions, using Trusted Computing. Fidelius aims to build a *trusted path* between a user and a remote server, while minimizing the TCB. Its ultimate goal is to provide confidentiality,

integrity and authenticity guarantees over some input fields of a web page, in which sensitive data needs to be inserted by the user, such as credit card information.

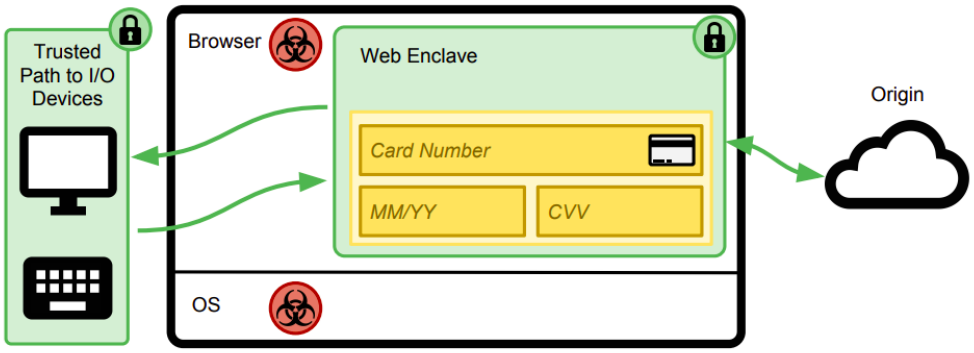


Figure 2.1: Overview of Fidelius. The web enclave, embedded in a malicious browser and OS, communicates with the user through a trusted I/O path and securely sends data to a remote server.

The solution adopted by Fidelius consists of hardware and software components implemented in the client. Firstly, a *web enclave*, which is an enclave built on top of the SGX architecture, running a minimalistic browser engine. Secondly, hardware dongles (e.g., microcontrollers) mounted between the I/O devices and the PC to protect sensitive input data (e.g., text inserted on the keyboard) and to send trusted output to the user (e.g., content displayed on screen). On the client's side, only these components are considered trusted, whereas the OS and browser might be maliciously controlled by an attacker. Figure 2.1 shows the Fidelius architecture. Input data such as a keystroke is captured by a dongle, which encrypts and sends it to the web enclave. Then, the web enclave manages the encrypted data, eventually forwarding it to the remote server. Output data, instead, goes in the opposite direction. The trusted path between a dongle and the web enclave is established through pre-shared symmetric keys. Conversely, web enclave and remote server perform RA to attest each other and protect the communication channel.

Fidelius can be enabled on specific HTML fields, which are `<script>`, `<form>` and `<input>`. A developer can protect these fields by specifying the `secure` attribute, along with a signature made with the server's private key, for integrity checks. When the focus goes to a `secure` HTML field, the mechanisms of Fidelius are put in place, transferring the control to the web enclave and enabling the secure paths to the I/O devices. A particular effort is made to inform the user about whether Fidelius is active at a particular moment or not. As a matter of fact, users must be assured that the trusted paths have been correctly established before inserting any sensitive data. To address this problem, two mechanisms are used: firstly, an LED is mounted on each dongle, which is turned on if the communication with the web enclave is secure. Secondly, a trusted region of the screen is used to inform the user about the current status of Fidelius, the identity of the remote server and the HTML field that is currently focused.

Although it was designed for a completely different use case, the work done by Eskandarian et al. has a lot of similarities with the Authentic Execution framework. In both cases, a trusted path needs to be established between high-end systems and I/O peripherals. The main difference resides in the attacker model: Fidelius considers trusted the dongles placed between the PC and the I/O devices, hence software and data in a dongle need not be protected (e.g., symmetric

keys can be stored in clear). Our framework, instead, ensures the isolation mechanisms and the attestation of every single component of the system.

Chapter 3

Problem Statement

This chapter looks into the general architecture of a distributed, event-driven application, outlining the risks and issues that might arise in such context and introducing an approach to achieve strong security guarantees in presence of attackers.

An illustration of the system is provided in section 3.1, which describes in detail all the involved parties and how they interact to each other. Section 3.2 presents the *attacker model*, defining what a malicious actor can and cannot do on the system. Finally, section 3.3 highlights the concepts of *Authentic Execution*, the approach used in this Master's Thesis to deploy a secure infrastructure.

3.1 System model

The System Model describes a *distributed*, *event-driven* application. Distributed means that the application is not deployed as a standalone unit, but rather is scattered among different components: in this Thesis, they are called Software Modules (SMs). Each SM is responsible to implement a specific functionality. Besides, modules are *connected*, meaning that they are able not only to exchange information with each other, but also to communicate with external entities. A SM resides inside a *node*, which is a component that provides to the SMs hardware and software resources. Usually, a node is identified as a "physical" component such as a high-end server or an embedded device; however, in this description a node is more a "logical" component. This means that, for instance, two nodes might be part of the same machine or, in more convoluted implementations, the same node might also be deployed on multiple machines.

An event-driven application, instead, is an application that reacts to external inputs, called *events* [29]. An event can be triggered by a user (e.g., a click of the mouse) or the system (e.g., an interrupt). Normally, an event is a message that contains some information, such as the identity of the source and custom data (e.g., which button of the keyboard was pressed). A key component of the application is the Event Manager (EM), which is responsible for receiving and processing external events, as well as executing the logic associated to them. Usually, an event-driven application does not run any main logic by itself (except for some background tasks), but rather continuously wait for external events. For instance, this is the case of many GUI applications, when the user directly interacts with them by sending events through input sources (e.g., mouse, keyboard). After an event is received, a function (called *handler*) is executed; handlers can be associated to a specific event both at compile time and at runtime. Then, after the code is executed, the application returns back to a waiting state, until a new event arrives.

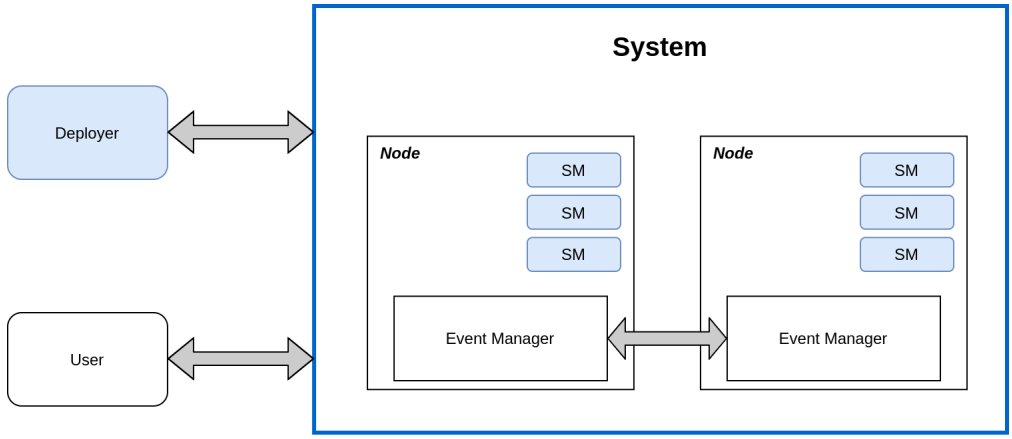


Figure 3.1: Overview of the system. Blue rectangles are considered *trusted* components.

Figure 3.1 shows an overview of the System Model. As described above, the system is composed by nodes, each of which includes an EM and zero or more SMs. A SM is not directly accessible from outside the node, as it is only connected to its local EM, which manages events to and from the SM. It follows that two SM never communicate directly, even in the case that they are both in the same node. The EM in this case acts as a "bridge", forwarding the event from the source to the destination SM. Instead, if the two SMs are on different nodes, the EM in the source node sends the event to the EM in the destination node, which then delivers the event to the destination SM.

In addition, external entities might interact to the system: the figure distinguishes between the *Deployer* and the *User*. The former is the entity responsible of managing the system. This involves tasks such as sending a SM to a node, establishing connections between SMs, setting encryption keys and so on. The latter is a more generic user, which can theoretically be anyone that is able to connect to the system (e.g., by being on the same network). Both entities should only be able to directly communicate with the EMs since, as explained above, the SMs are normally inaccessible from outside their own node. However, events can be delivered to them to trigger specific *entry points* (Section 3.3.3). Certainly, a SM needs to distinguish between an event sent by either another SM, the Deployer or the User, especially considering the fact that the latter might be a malicious entity. Hence, mechanisms that enforce integrity and authenticity of the events need to be implemented, as described in the next paragraphs.

The figure shows in blue the components that are considered *trusted*: first of all, the Deployer is expected to be trusted, since he is the entity responsible for setting up the distributed application. Considering that he is external to the system, it is also assumed that his own software and hardware are not compromised. In addition, all the SMs are trusted, since they are directly developed by the Deployer and run inside a protected environment called *enclave* (Section 2.2).

The main limitation of this model is that there are no availability guarantees: for instance, if the EMs are controlled by the attacker, they might drop all the events they receive, meaning that the SMs would be isolated from the rest of the world, even if they are correctly loaded and attested. There are many other ways to take out the system (e.g., by controlling the network, the OS of the machines where the nodes reside, etc.), hence availability is explicitly left out of scope of this model.

Finally, two important aspects need to be emphasised: firstly, that the system is *heterogeneous*, meaning that the SMs might be developed for different architectures, according to their

functionalities. For instance, a SM that needs to perform highly-expensive computation is likely to be deployed on a high-end platform, whereas a SM responsible to read sensor data would be positioned on a low-end node (e.g., a microcontroller). The second significant aspect is that the Deployer is not necessarily the owner of the infrastructure. The application, in fact, might be deployed on someone else's hardware. Similarly, many applications from different Deployers might potentially run on the same infrastructure.

3.2 Attacker model

Attackers have multiple capabilities. First of all, attackers can manipulate all the software of the nodes. This includes the EMs and any other software running on the nodes, including the OS. They can also deploy their own applications on the infrastructure. Secondly, attackers can control the communication network used by the nodes to communicate to each other. This includes sniffing the network, modifying traffic, and mounting man-on-the-middle attacks. Finally, with respect to the cryptographic capabilities of the attacker, the Dolev-Yao model is followed [30].

Attacks against the hardware are considered to be out of scope. It is assumed that an attacker is not able to have physical access to any component of the system. In addition, side-channel attacks are not contemplated. Although protection to these kind of attacks is essential, it is considered to be orthogonal and complementary to the security properties offered by the approach described in this work.

3.3 Authentic Execution

In 2017, Noorman et al. introduced *Authentic Execution* [2], a framework for distributed, event-driven applications with strong emphasis on integrity. In this Master's Thesis work, the same notions have been applied to provide security on the System Model. This section summarizes the content of the original paper, highlighting its key concepts.

3.3.1 The concept of *Authentic Execution*

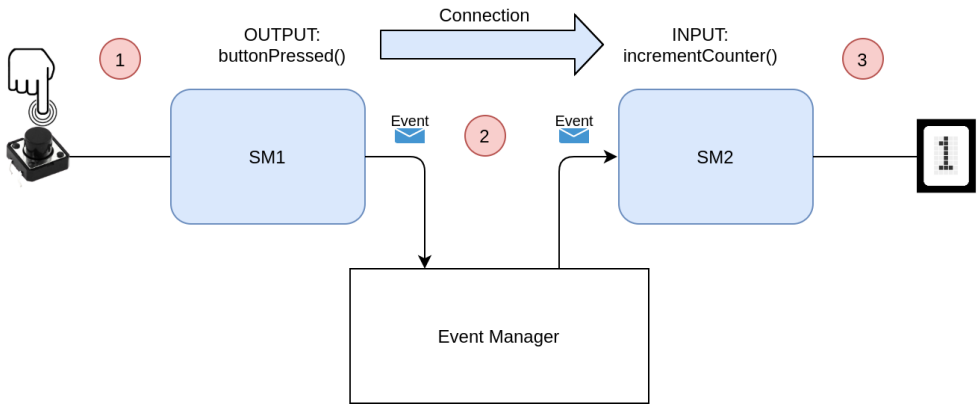


Figure 3.2: Example of Authentic Execution between two SMs. *Only* an authentic button press can trigger the increment of the counter shown in the LCD display.

According to the paper, the notion of Authentic Execution is described as follows:

> "(...) if the application produces a physical output event (e.g., turns on an LED), then there must have happened a sequence of physical input events such that that sequence, when processed by the application (as specified in the high-level source code), produces that output event."

Figure 3.2 shows a more practical example of this concept. Two SMs are part of a system: SM1 and SM2. For the sake of simplicity, it is assumed that they reside on the same node, hence events exchanged between the two are handled only by the local EM. Both the SMs are connected to I/O devices: SM1 is connected to a button (*input*), SM2 to a LCD display (*output*). In addition, these devices cannot be tampered with, which means that it is not possible to generate fake button events, nor to control the LCD display outside of SM2's code. This means that their drivers are protected using *Secure I/O*. Therefore, if SM1 is notified that the button has been pressed, it is certain that someone has *physically* triggered the button. Similarly, if the LCD display changes its content, it has to be due to some logic implemented in SM2.

The behavior of this small system is straightforward. The LCD display shows a number (initially zero), which is increased every time the button is pressed. At deployment time, the Deployer establishes a *connection* between SM1 and SM2: the output function `buttonPressed` of SM1 is connected to the input handler `incrementCounter` of SM2. Hence, after `buttonPressed` is called due to an external physical input (1), an *event* is generated and sent from SM1 to SM2 through the EM (2). SM2, once the event is received, calls `incrementCounter`, which updates the value of the number shown in the LCD display (3).

Authentic Execution means that, if the value shown in the LCD display has changed, the execution can be traced back to the physical event that triggered `buttonPressed`. In other words, if the counter is incremented by one, an external observer can deduce with certainty that the button was physically pressed before, because it is impossible to update the value of the counter otherwise. It is noteworthy that the opposite, instead, is not true: even if the button is pressed, the counter might not be incremented, because there is no guarantee that the event will be delivered to SM2, as all the components of the system except for the two SMs are *not trusted*. For instance, the event might be dropped, lost, or intercepted by an attacker.

3.3.2 Security properties

As the example in the previous section shows, Authentic Execution places particular emphasis on *integrity* and *authenticity* of data. The trace that brings the LCD display to update its content must be *authentic*. This notion precludes a wide variety of threats, including attacks where an adversary tampers with the transmission of events or injects fake ones on the network, as well as other attacks aimed to manipulate the execution of SMs.

Although this framework is not primarily focused on offering *confidentiality* of data, it still provides protection of the application's state, as well as the content of events. However, an attacker is still able to observe the system and deduce information such as the identities of sender and recipient of an event.

Finally, and as already mentioned before, this implementation does **not** provide any *availability* guarantees: for instance, an attacker might suppress the network communication or manipulate the EM to drop all received events.

3.3.3 Technical implementation details

Software Modules

A SM runs in an isolated, protected domain called Trusted Execution Environment (TEE) (Section 2.2). Each SM runs inside an enclave, which protects its code and data from the outside world. In addition, a SM can be *attested*, meaning that the Deployer is able to verify its authenticity at runtime, using Remote Attestation (RA). Furthermore, each SM comes with a symmetric *Master Key* (which might be obtained during the RA process), known only by the module itself and the Deployer, that can be used to establish a secure communication channel to exchange sensitive data.

For each SM, the Deployer specifies a set of *entry points*, which are essentially functions that can be called from outside the SM. There are no restrictions about the identity of the caller, it follows that for some critical entry points mechanisms to preserve confidentiality, integrity and authenticity of data need to be implemented, e.g., by using the Master Key. Moreover, in the context of Authentic Execution a SM can also have *inputs* and *outputs*, special private functions that are used to create *connections* between two or more SMs.

Connections

A *connection* links the output of a SM with the input of another. Each connection is identified by an ID and uses a *symmetric key* to protect the event data. A nonce is also used to provide freshness and avoid replay attacks. Connections are established by the Deployer, calling a special *entry point* of the two involved SMs: `setKey`. For each of them, the Deployer encrypts and authenticates all the sensitive information needed to establish the connection (connection key, connection ID, input/output ID of the SM), then calls the `setKey` entry point passing such data as argument. Then, the SM verifies the correctness of the content and, if the verification succeeds, sets up the new connection. At the same time, the Deployer notifies the source EM that a new connection has been established, providing all the information needed to correctly deliver events from source to destination.

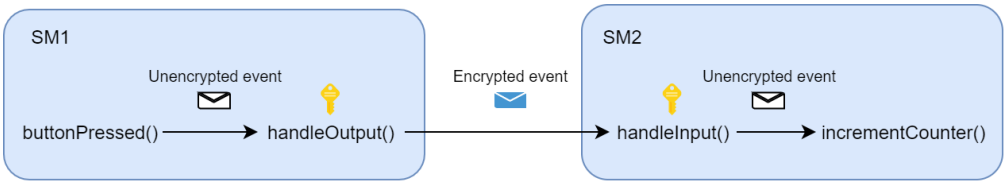


Figure 3.3: Example of a connection between two SMs. The event is encrypted and authenticated during the transmission.

Figure 3.3 illustrates how events are passed from one SM to another. When the output in the source SM is generated, the event is created and sent to a private function called `handleOutput`. There, the event is encrypted and authenticated using the connection's key and nonce, then sent to the EM (not displayed in the figure). To deliver the event, a special entry point of the destination SM is called: `handleInput`. The data passed as argument is decrypted and, if the decryption succeeds, the SM calls the corresponding input function.

Deployment

The Deployer uses his own (trusted) infrastructure to compile the SMs, whose binaries are then sent to the corresponding nodes. The deployment of a SM is an *untrusted* operation, as the binary might be tampered with either during the transmission or the loading process. In this situation, RA becomes essential to ensure that the execution of a SM is authentic.

To set up the system, the Deployer uses a special file called *deployment descriptor*: here, he specifies the configuration of the system, declaring nodes, modules and connections. Nodes are identified by a name, and include information such as type (i.e., the underlying hardware architecture) and address. Modules are the SMs, which are identified by a name and assigned to a specific node. Additional fields might be included in a module's entry, depending on its type. Finally, connections are declared, each of which specifies: the source module, the source output, the destination module and the destination input. Listing 3.1 shows an example of a deployment descriptor, which describes the infrastructure presented in Figures 3.2 and 3.3. For simplicity, some fields have been omitted, such as the type of the modules and nodes.

To deploy the system, the deployment descriptor is parsed by a special tool, which builds the modules, loads them on the nodes, and finally establishes connections between them.

```
{
    "nodes": [
        {
            "name": "node1"
        }
    ],
    "modules" : [
        {
            "name": "sm1",
            "files": ["sm1.c"],
            "node": "node1"
        },
        {
            "name": "sm2",
            "files": ["sm2.c"],
            "node": "node1"
        }
    ],
    "connections": [
        {
            "from_module": "sm1",
            "from_output": "buttonPressed",
            "to_module": "sm2",
            "to_input": "incrementCounter"
        }
    ]
}
```

Listing 3.1: Example of a deployment descriptor.

Chapter 4

Design and Implementation

This chapter shows the implementation details of this Master's Thesis work. It is structured as follows: Section 4.1 gives an overview on the main contributions of this Thesis, with respect to previous work on Authentic Execution. Section 4.2 describes in detail the application-level protocol we designed for the communication between the components of a system, illustrating the structure of each type of message. Section 4.3 introduces the details of the Authentic Execution framework we developed for Intel SGX, whereas Section 4.4 reveals the modifications we made on the already existing Sancus implementation. Finally, section 4.5 describes the tool that we implemented to automatically deploy an Authentic Execution system, given a deployment descriptor as input.

4.1 Extending the Authentic Execution implementation

The main goal of this Master's Thesis was to continue the work carried out by Noorman et al. on the Authentic Execution framework [2], by providing extensions and new features to support the deployment of a distributed application on a real use case. This section presents the most important ones.

4.1.1 Support for Intel SGX

The major limitation of the previous work concerns the number of supported TEEs. In fact, even though the principles of Authentic Execution can be applied to any TEE, only support for Sancus was actually implemented. However, a system of only embedded devices is not sufficient in many applications: high-end nodes are often needed to perform tasks that are unsuitable for low-end embedded systems, because of the limited resources at their disposal. These tasks include both computation (e.g., to execute the logic to move a robot, or to compute statistics using Machine Learning) and storage (e.g., to store and aggregate sensor data). Thus, we addressed this issue by providing Authentic Execution support for Intel SGX. Full details of its implementation are presented in Section 4.3.

4.1.2 Many-to-many relationships

Regarding Authentic Execution, a *connection* establishes a secure channel between the output of a SM and the input of another (Section 3.3). The implementation described in the paper allows an output to be connected only to a single input, and vice versa. In other words, inputs and

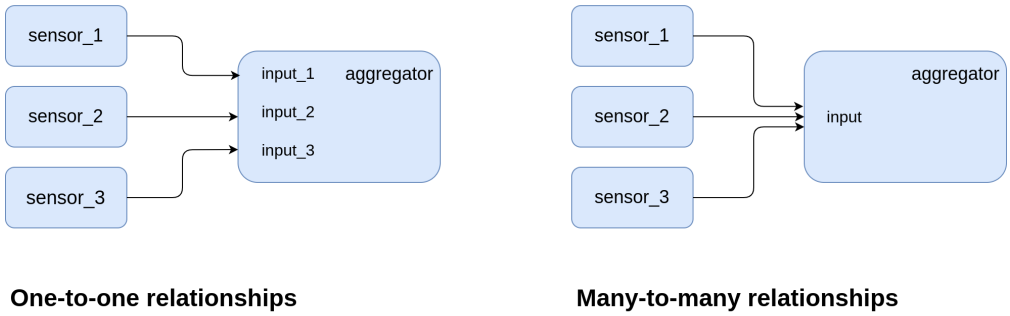
outputs have an one-to-one relationship. Even though this results to a simpler implementation, it is a limitation: in some cases it is useful to link an output to multiple inputs (e.g., to deliver the same sensor data to different SMs in order to perform different tasks, such as computation and storage), or different outputs to the same input (e.g., to send data from different sensors to the same "aggregator" SM).

We solved this problem in this Thesis work by introducing slight modifications in the framework, to support many-to-many relationships between inputs and outputs. In the Authentic Execution paper, a *connection identifier* is a number that identifies the input/output of a SM. It is local to a SM, meaning that different inputs/outputs in different SMs might have the same identifier. When the `handleInput` function of a SM is called, it is immediate to retrieve the connection data (encryption key, nonce), since only one connection can be associated with the requested input/output ID. This does not work with many-to-many relationships: given an input/output ID, there might be many connections associated to it. To solve this issue, it is sufficient to consider the connection identifier as an *unique, global* identifier of a specific connection, rather than an input/output ID. As a consequence, the two SMs of a particular connection will share the same identifier, which will be used to retrieve the connection data. Besides, on each SM the connection identifier needs to be associated with the corresponding input/output, e.g., by using a data structure such as a map. When an output is generated, the `handleOutput` function retrieves all the connection IDs associated to that output (by accessing the data structure), and sends a separate event for each connection. In contrast, `handleInput` has to perform the opposite procedure, to retrieve the input associated to the connection identifier passed as argument.

The introduction of this feature certainly brings some advantages. In general, the total number of inputs and outputs is reduced, as shown in Figure 4.1. Example (a) illustrates a scenario where three sensors send their data to an aggregator. If, with one-to-one relationships, different sensor outputs need to be connected to different inputs of the aggregator SM, with many-to-many relationships it is sufficient to have a single input that receives all the data. The main benefit obtained by the latter implementation is that the overall size of the aggregator module is reduced, by avoiding code duplication. Similarly, in Example (b) a sensor needs to send its data to both a computation and a storage SMs: with one-to-one relationships the sensor needs two distinct outputs, whereas the same output can be used for both the recipients with many-to-many relationships. Again, a reduction of the code size can be observed. Furthermore, it is evident that this solution is also simpler for the Deployer to implement, which only needs to write one single instruction to trigger both the events, resulting in prettier and more manageable code.

However, there are some drawbacks in the implementation of many-to-many relationships that must be mentioned. In order to associate a connection identifier to an input/output, more information needs to be stored on each SM: the size overhead depends on the size of the identifiers and on the data structures used in a particular implementation (e.g., Authentic Execution framework for Sancus uses simple arrays, whereas the SGX version uses hash maps). Other inconveniences concern performance: with one-to-one relationships, both `handleInput` and `handleOutput` have a O(1) complexity, because it is immediate to deduce the input given a connection identifier, and the connection identifier given an output. With many-to-many relationships, instead, the situation is different: the complexity of each of those handlers varies from O(1) to O(n), depending on the data structures used. In `handleOutput`, for instance, all the connections associated to a specific output need to be retrieved: this operation might be performed with a simple iteration over all the connections (complexity O(n)), or with a direct access to a hash map that, given the output ID as a key, returns the list of all the connections associated to that output (complexity O(1)). Each TEE implementation might use a different strategy, to find the best trade-off

(a) Sensors send their data to an aggregator



(b) A sensor send its data to both a computation and a storage module

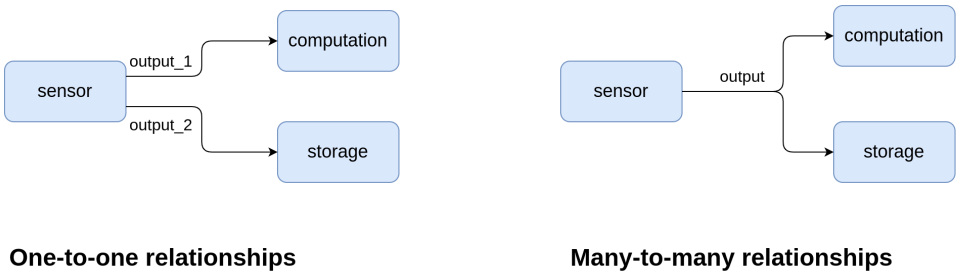


Figure 4.1: Difference between one-to-one relationships and many-to-many relationships between inputs/outputs of a connection.

between performance and memory used.

4.1.3 Periodic events

This sub-section describes a convenience feature that we implemented in order to automatize a periodic task. As described below, it is an *untrusted* feature implemented into an EM. There are no security guarantees in this implementation: on the contrary, an attacker might easily exploit it to affect the availability of a node. Furthermore, developers don't have any guarantees that a task is actually executed with the specified frequency. Availability in TEEs is an orthogonal work and there is active research in this direction [31].

For certain applications, it is necessary to perform some tasks periodically (e.g., reading data from a sensor every second). In the context of Authentic Execution, a task can be triggered either by an external input (e.g., an I/O device such as a button) or an external entity (e.g., a user that wants to compute the average temperature over the last hour). Particularly, the latter case can be generated by calling a specific *entry point* of a SM, as described in Section 3.3. Hence, to set up a periodic task, it is sufficient to call an entry point periodically, from outside the SM. Naturally, this is impractical to be performed manually; nonetheless, a user might easily create a

script that does the job automatically. The drawback of this solution is that the script needs to run inside the user's PC and, in order for the script to work, the PC needs to be always turned on and connected to the same network as the system. To solve this issue, the user might want to run the script on a node (e.g., an high-end SGX server). However, it is not always possible to do so, firstly because the user might not be able to control the node, and secondly because the script might not be compatible with its architecture (e.g., in the case of an embedded Sancus node). Thus, we addressed this problem by implementing a new feature on the EM to support the automatic establishment of periodic events.

```
{
    "periodic-events" : [
        {
            "module" : "sm1",
            "entry" : "read_from_sensor",
            "frequency" : 500
        }
    ]
}
```

Listing 4.1: Deployment descriptor extension for periodic events.

The implementation of periodic events works as follows. The Deployer configures the periodic events in the deployment descriptor (as illustrated in Listing 4.1) by specifying, for each event: the SM name, the entry point to call and the frequency (in milliseconds). Then, for each of the entries in the descriptor, a special *RegisterEntrypoint* message (Section 4.2.8) is sent to the EM where the SM resides, which updates the list of active periodic events. From that moment, and with the specified frequency, an event to call the specified entry point will be generated automatically. This work is done by a secondary thread of the EM, which uses timers to ensure that the entry points are called at the right time. The events, once generated, are sent to the main thread, which processes them as any other event.

Some considerations need to be taken into account. Firstly, a little to no overhead is added to a node when periodic events are not actually used. The actual cost depends on the implementation of this feature. An optimized version of the EM would allocate the list of active periodic events and run the secondary thread *only* if at least one periodic event is present on that node. Secondly, the implementation of periodic events is local to each node, which means that each node acts on its own. Compared to the case where a single, external, event generator is present somewhere in the system, this solution reduces the traffic on the network and allows periodic events to work even with loss of connectivity between components. Regarding the latter, periodic events offer also an opportunity to deal with availability issues: for example, a periodic event might be scheduled to control an actuator autonomously, if the SM does not receive any commands for a long time (which might be an indicator that either the network or the controller are down).

Another important consideration is that this feature, since it is included inside the EM, is *untrusted*. There are no guarantees that events will be generated, especially in the case that an attacker takes over the node. Similarly, an attacker might tamper with the frequency of which the entry points are called. For a sensor reading it might not be a huge problem if the event is triggered more or less frequently, however this might have an impact on other tasks. The Deployer should be aware of the risks that might occur when such feature is enabled.

4.2 Application-level protocol

In a distributed application, the SMs need to communicate with each other to exchange information. In an Authentic Execution system, a SM's output can be connected to another SM's input: if they belong to different nodes, it is necessary that the EMs of the two nodes establish a connection to exchange events. Furthermore, another kind of communication is needed to *interact* with the system, e.g., to call an entry point or to deploy a SM to a node.

In this work, we designed a basic application-layer protocol to handle all of these needs. The details of this protocol are accurately described in the next sub-sections. It is noteworthy that there is little to no mention about the underlying layers, as they strongly depend on the system. For instance, in an automotive environment a protocol called Controlled Area Network (CAN) is widely used [32], whereas in a different context (e.g., Smart Farming) a wireless technology such as WiFi, ZigBee or Bluetooth would be more suited [33]. In this implementation, we used a TCP/IP stack for each component of the system.

We identified two main types of messages: a *Command* and a *Result*. A Command handles all the possible interactions between nodes, and between a node and an external entity (e.g., the Deployer). A Result, instead, is the response to a Command, which contains a result code and optional data. The former is used to inform the sender whether the command has succeeded or not, while the latter contains information, e.g., return values of entry points. A fully detailed description of these messages is provided in the next sub-sections.

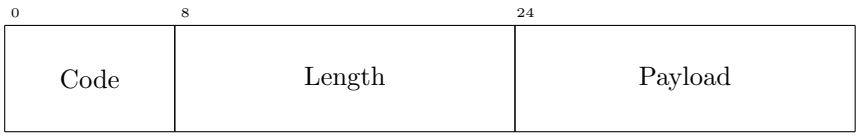
```
class ReactiveCommand(IntEnum):

    Connect             = 0x0
    Call                = 0x1
    RemoteOutput        = 0x2
    Load                = 0x3
    Ping                = 0x4
    RegisterEntrypoint  = 0x5
```

```
class ReactiveResult(IntEnum):
    Ok                  = 0x0
    IllegalCommand      = 0x1
    IllegalPayload      = 0x2
    InternalError       = 0x3
    BadRequest          = 0x4
    CryptoError         = 0x5
    GenericError        = 0x6
```

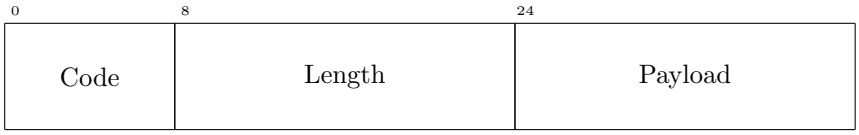
Listing 4.2: Command and Result codes.

4.2.1 Command



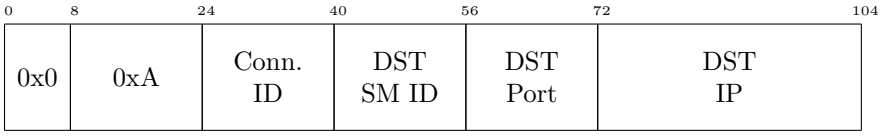
The general format of a Command consists of a 8-bit field indicating the code, a 16-bit field for the length of the payload and the payload itself. Given the size of the `Length` field, the payload can be at most 65535 bytes long, which is widely acceptable for almost all types of commands. The only exception is the *Load* command: as described in detail later, this kind of event needs a different consideration, hence the format of such message differs from the others. Listing 4.2 shows all the command codes used in this implementation.

4.2.2 Result



The format of a Result message is the same as a Command message. The code in this case reports whether the Command has succeeded or not. A result code of zero, as usual, implies that everything went well. If the code is different from zero, instead, it means that an error occurred. Listing 4.2 shows the result codes used in this implementation and to which kind of error are associated. It is worth mentioning that not every Command type requires this response.

4.2.3 Connect



A *Connect* message is sent from the Deployer to an EM to establish a connection between an output and an input of two SMs. It is sufficient to send this message only to the EM of the node where the source SM is installed: the information contained in this message is only needed to correctly deliver an event to the destination node, which does not even need to know that such connection exists.

The format of a *Connect* message is the following:

- `Conn. ID` (16 bits): Unique identifier of the connection.
- `DST SM ID` (16 bits): Identifier of the destination SM.
- `DST Port` (16 bits): TCP port used by the destination EM.
- `DST IP` (32 bits): IPv4 address of the destination EM.

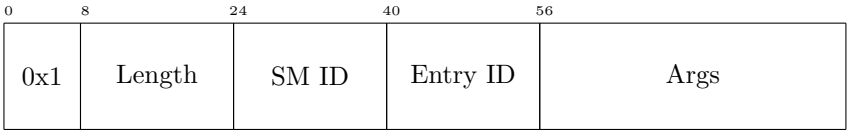
Connection identifiers are globally unique within the system. SM identifiers are locally unique inside a node, and they are assigned by either the Deployer or the node when a SM is loaded. Input/output identifiers, instead, are locally unique within a SM and they are assigned at compile time.

The format of this message is strongly dependent on the protocols used at lower levels. In our implementation, a TCP/IP stack is used, making this format acceptable. However, future work might be needed to overcome this limitation, allowing different protocols. A possible solution might be adding a field to specify the *type* of the connection, followed by an address and any other information needed to communicate with the destination EM. This way, the source EM would exactly know which media it has to use for each connection.

Result

A Result message in this case is expected, to notify the Deployer whether the Command has succeeded or not. That is, the response contains only the result code, without payload. Hence, the `Length` field is set to zero.

4.2.4 Call



A *Call* message is sent from the Deployer to an EM to trigger the execution of a specific entry point.

The format of this message is the following:

- `Length` (16 bits): Total length of the subsequent fields.
- `SM ID` (16 bits): Identifier of the SM.
- `Entry ID` (16 bits): Identifier of the entry point to be called.
- `Args` (variable size): Optional arguments to be passed as input to the entry point.

The size of `Args` can be retrieved from the `Length` field of the Command. In fact, it is sufficient to subtract four bytes to its value: two bytes for the SM ID and two bytes for the Entry ID.

Any entity (e.g., an attacker) is capable of calling a SM's entry point. Indeed, it is sufficient to know the identifiers of the SM and the entry point, as well as the address of the node where the SM is loaded. For this reason, the Deployer should take extremely care of the logic of the entry points, for instance by preventing them to return sensitive data in clear. Input parameters need the same considerations to be taken into account: this is, for instance, the case of *setKey* and *handleInput*, the two entry points defined in Authentic Execution (Section 3.3.3).

setKey

After a connection is established, a symmetric key to protect the communication is used. This key must then be transmitted to the two SMs of the connection in a secure way. For this purpose, the Deployer calls the *setKey* entry point on each SM.

The format of this message slightly differs, depending on the type of the SM. The content, instead, is the same: the connection's symmetric key encrypted with the SM's Master Key, using AEAD. As associated data the connection ID, the input/output ID and a nonce are provided. The SM, when this entry point is called, decrypts and authenticates the payload and, if the operation succeeds, sets the key of that connection. Additionally, if the SM supports different encryption algorithms an additional field needs to be provided.

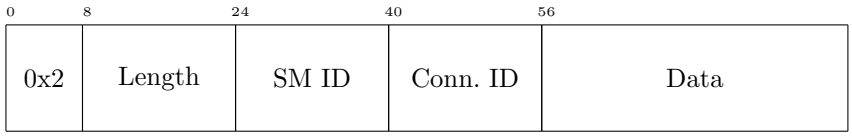
handleInput

handleInput is the entry point called by the EM to trigger a specific input of a SM. It is the consequence of a *RemoteOutput* event (Section 4.2.5). In order to secure the communication between the two SMs, data sent from the output to the input is encrypted and authenticated using the AEAD algorithm specified for that connection in the deployment phase. The symmetric connection key and a 16-bit nonce are used to encrypt and authenticate the event.

Result

A Result message is usually expected after a *Call* command. Payload is optional but might be included in most cases, especially for user-defined entry points.

4.2.5 RemoteOutput



A *RemoteOutput* message is sent from an EM to another in order to post an event. This happens when the two SMs of the same connection are in separate nodes, after the output is emitted by the source SM. The source EM, then, uses the information provided in the *Connect* message to open a connection with the destination EM which, after receiving the message, calls the *handleInput* entry point of the destination module.

The format of this message is the following:

- `Length` (16 bits): Total length of the subsequent fields.
- `SM ID` (16 bits): Identifier of the SM.
- `Conn. ID` (16 bits): Identifier of the connection.
- `Data` (variable size): Payload, encrypted and authenticated using AEAD. The payload usually includes the encrypted data attached by the source SM, as well as a MAC used for authentication, which has a fixed length according to the encryption algorithm used.

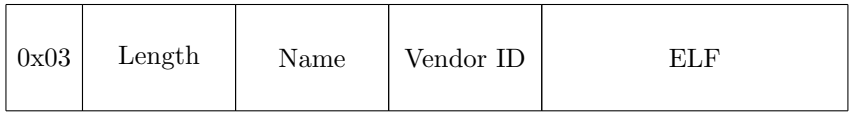
Result

A Result message is *not* expected in this case.

4.2.6 Load

The *Load* command is sent by the Deployer to an EM to install a SM inside a node. Each TEE needs different information to do this operation, hence there is no fixed format for this kind of message. The only field common to all TEEs is the command code, which is `ReactiveCommand.Load` (Listing 4.2). Below, details about the packet format are provided for Sancus and SGX.

Sancus



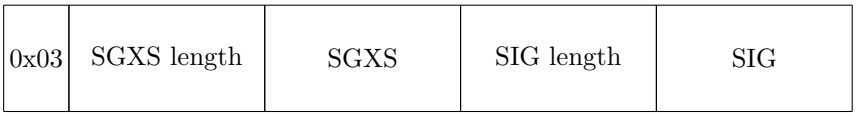
The format of a Load packet for a Sancus SM is the following:

- `Length` (16 bits): Total length of the subsequent fields.

- `Name` (variable size): Name of the SM, null-terminated.
- `Vendor ID` (16 bits): ID of the Deployer, used to derive the SM's Master Key.
- `ELF` (variable size): Binary of the SM.

During the loading process, the ELF binary is updated with the correct addresses of sections and symbols. These addresses need to be sent back to the Deployer (inside the Result message), which updates his local copy of the ELF file, to be able to derive the Master Key by himself and attest the SM (Section 2.2.1).

SGX

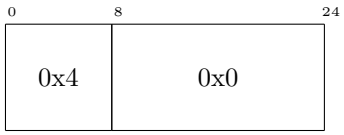


The format of a Load packet for a SGX SM is the following:

- `SGXS length` (32 bits): Length of the SGXS field.
- `SGXS` (variable size): Binary of the SM, in the processor's native enclave format (SGXS).
- `SIG length` (32 bits): Length of the SIG field.
- `SIG` (variable size): Signature of the SM, made with the Deployer's private key.

As Result message, the EM sends only the result code, indicating whether the loading process has succeeded or not.

4.2.7 Ping

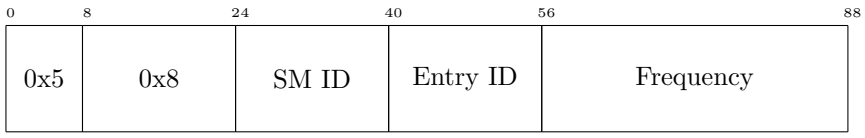


A *Ping* message might be used to assess the state of a node. It does not contain any information; if an EM receives this event, it replies with a Result message to inform that it is currently active on the node. However, the Deployer cannot be confident that the response is actually sent by the EM, as anyone on the network can forge this message. Knowing whether an EM is running or not would require the hardware support of a TEE and attestation mechanisms to be executed. Hence, this message should be used only for debug purposes.

Result

A Result message contains only the result code (which has to be zero, for obvious reasons), without payload.

4.2.8 RegisterEntrypoint



A *RegisterEntrypoint* message is sent from the Deployer to an EM in order to register a periodic event (Section 4.1.3).

The format of this message is the following:

- `SM ID` (16 bits): Identifier of the SM.
- `Entry ID` (16 bits): Identifier of the entry point to register.
- `Frequency` (32 bits): Integer representing the frequency of which the entry point will be called, in milliseconds.

The EM stores this information inside a data structure, which is accessed by the secondary thread in order to create the events that are delivered to the main thread, with the specified frequency.

Result

A Result message in this case is expected, to notify the Deployer whether the Command has succeeded or not.

4.3 Authentic Execution in Intel SGX

Our implementation of the Authentic Execution framework for Intel SGX is characterized by one, fundamental aspect: the use of *Rust* as programming language [34], which is a modern solution to develop secure applications. Section 4.3.1 provides a short description of this language, highlighting its key concepts. This choice ruled out the use of the framework developed by Intel to write SGX applications (SGX SDK [35]), as it was designed for enclaves written in C/C++. However, in recent years new solutions came up, allowing developers to write enclaves entirely in Rust. Two main frameworks need to be mentioned: Rust SGX SDK [36] and Fortanix Enclave Development Platform (EDP) [37]. This Master's Thesis used the latter, which provides a simpler and more efficient way to write SGX enclaves. Section 4.3.2 gives an overview of its characteristics. The remaining sections, instead, focus on the actual implementation of the Authentic Execution framework.

4.3.1 Rust

Rust appeared for the first time in 2010 from a group of employees at Mozilla Research. It was only in 2015 that the first stable version (Rust 1.0) was published. Following that, new stable releases have been periodically released every six weeks. A major update came out in 2018 (called Rust 2018), which provided a huge number of improvements and new features. At the time of writing, the language reached version 1.44.1.

The key concepts of this language are the following:

- *Performance.* The goal of Rust is to be "as fast as C++ for most things"[1]. Rust uses the *zero-cost abstractions* paradigm, trying to provide high-level constructs and functionalities without incurring in unnecessary overheads. However, due to the runtime safety checks mentioned below, it is not always possible to obtain the same performance as C/C++ applications.
- *Reliability.* Rust does not use a garbage collector, as this would have gone in contrast with the performance requirements mentioned above. Instead, it exploits the concept of *ownership and borrowing* to keep track of the memory used. In short, memory and other resources belong to a *scope* (e.g., a function). After the execution changes scope, all the memory that was associated with the old scope is released, except for returned values. This is also known as Resource Acquisition Is Initialization (RAII) convention. A resource can also be borrowed, meaning that it might be used outside of its scope, which still keeps ownership of it. The borrowed object can either be immutable or mutable.

  The compiler plays a huge role on this approach. It knows when a resource goes out of scope and should then be destroyed, therefore produces a compiler error whenever an illegal access is made, e.g., through NULL or dangling pointers. Additionally, it also forbids that multiple references of the same resource exist at the same time, if at least one of them is mutable: this prevents data races. Hence, many memory management vulnerabilities are detected during the compilation process. Furthermore, runtime checks are performed to deal with vulnerabilities that cannot be identified by the compiler (e.g., runtime bounds checks).

  Rust also provides the developers a functionality called *Unsafe Rust*, to circumvent these restrictions to allow low-level tasks. However, it should be noted that all the memory safety checks described before are not enforced in this case. Hence, a developer should use this feature carefully and for limited operations.
- *Productivity.* Rust is relatively easy to learn and it includes plenty of modern programming techniques taken from other languages. It provides a complete and extensive documentation, called *The Book*[2]. Its compiler is friendly, with useful error messages. Besides, tools are included to easily manage projects, to import external libraries into a project and to write tests and documentation in a simple and fast way. For all of these reasons, Rust has become one of the most loved programming languages by developers[3].

In conclusion, we chose Rust in this Master's Thesis to provide additional security to our framework. As a matter of fact, even though an application is isolated inside an enclave, if the code is bugged the system still remains exposed to attackers. Thanks to Rust, instead, a huge number of memory management vulnerabilities is avoided at the root, making the system more robust.

4.3.2 Fortanix EDP

In 2019, Fortanix released Enclave Development Platform (EDP), a framework to write SGX applications using Rust. Its main approach is, unlike Rust SGX SDK, to be fully integrated

[1] https://doc.rust-lang.org/1.0.0/complement-lang-faq.html#how-fast-is-rust?

[2] https://doc.rust-lang.org/book/

[3] https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved

with the Rust compiler, allowing developers to immediately use new features as soon as they are released. It is straightforward to write EDP applications, because they look like native ones (Listing 4.3): they, too, contain a main function and may have multiple threads and network connections. No additional code is required to run the application inside an enclave. The magic happens at compile time, where the target `x86_64-fortanix-unknown-sgx` is passed as parameter and the compiler automatically builds the SGX application, adding the abstractions and interfaces provided by EDP to manage the enclave.

```
fn main() {
    println!("Hello, world!");
}
```

Listing 4.3: Hello world application, written to be run inside an enclave using Fortanix EDP. As can be noted, there is no difference in terms of code with a native application.

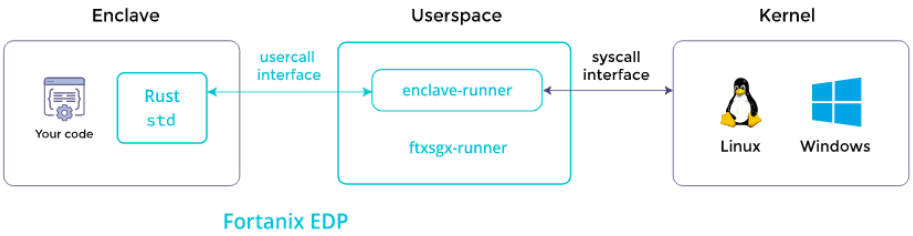


Figure 4.2: Overview of the main components in Fortanix EDP.

Figure 4.2 shows the main architecture of an EDP application: the program written by the developer resides inside an enclave and it is integrated with the Rust standard library (std). Outside the enclave runs `enclave-runner`, a crate responsible for loading the enclave and communicating with it through the *usercall interface*, which is the equivalent of the ECALLs/OCALLs in SGX SDK. In short, the enclave uses this interface to communicate to the external world, in both directions. The figure shows that `enclave-runner` resides into `ftxsgx-runner`, which is the default executable provided by EDP. However, a developer might want to write his own runner, if he needs particular functionalities. Finally, `enclave-runner` communicates with the kernel with the usual syscall interface.

In terms of security functionalities, EDP applications are not different from SGX enclaves written with Intel's official SDK. Naturally, the enclave is fully isolated from the rest of the system and it can provide proof of its identity using Remote Attestation (RA). Also, data can be persistently stored in a secure way using the SGX's *sealing* feature [20]. As already mentioned before, enclaves are able to use the Rust standard library. Nevertheless, not all functionalities work as usual inside the enclave: for security reasons, some primitives are slightly different (for instance, environment variables cannot be passed into the enclave, or hostname resolution is not available for network communications), whereas some others are not provided at all (e.g., file system management and timeouts).

Van Bulck et al. investigated the implementation of various TEE runtime libraries to assess the security w.r.t. a set of low-level vulnerabilities [38]. The analysis shows that EDP is a solid choice as it offers protection to almost all the vulnerabilities, thanks also to the use of the safe Rust as programming language. The few weaknesses found in EDP have been reported to

Fortanix and patches have been issued.

4.3.3 General Overview

A node is a subsystem that contains an EM, the *untrusted* component that manages SMs assigned to the same node. All of these entities are reachable through a well-defined interface, which is a TCP socket where they listen for connections. That is, each component is associated to a specific port and reachable through it. The EM port is passed as command line argument, whereas a SM is assigned to a port at compile time, which is the sum of the EM port and the SM ID. For instance, if the EM listens to port 5000, SM1 listens to 5001, SM2 to 5002, and so forth.

This behavior was decided for the sake of simplicity. Since, at the time of writing, it is not yet possible to pass arguments to an enclave in Fortanix EDP[4], we chose to hardcode the port in the SM code at compile time. Given that, defining the SM port in this way does not require extra efforts, as it is simply a sum of two numbers. Similarly, there is no need to store additional information in the EM about the SMs, since their port can be easily derived as explained before.

On the other hand, this design choice leads to a few limitations: if a port is not available in a machine for some reason (e.g., it is already used by another service), the SM fails to initialize and will not be reachable. Other issues might happen if two nodes are placed on the same machine. For instance, if the two EMs listen to two adjacent ports such as 5000 and 5001, SM1 loaded on the first node cannot work, because its port would be 5001 as well. A possible solution for these problems might require the implementation of some logic to find an available port before loading a SM. The EM might scan for an usable one (e.g., by trying to listen to a specific port: if the instruction succeeds, it means that the port is available) and inform the Deployer, which will hardcode its value in the code. The implementation of a more advanced solution for this problem is left as future work. In this application, we assumed that all the ports in a machine are available, and all the EMs loaded in the same machine listen to ports far away from each other (e.g., 5000 and 6000).

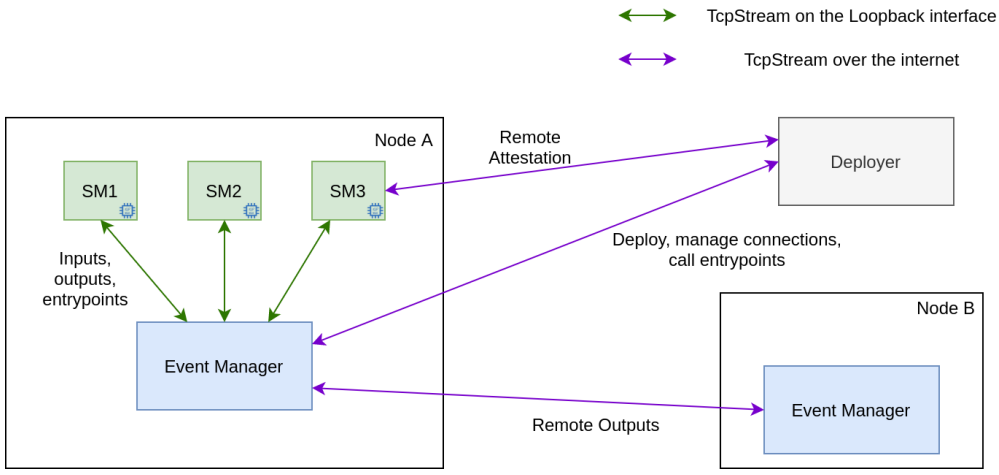


Figure 4.3: Overview of an Authentic Execution system with Intel SGX.

Figure 4.3 shows an overview of an Authentic Execution system in this configuration. The

[4] https://github.com/fortanix/rust-sgx/issues/136

SMs listen for connections on the *loopback* interface (except for the initial RA phase, where they need to directly communicate with the Deployer). In this way, an external entity *cannot* open a connection with the SM directly, but only through the EM of the same node. This is not true if the entity lies on the same machine as the SM: in this case, since the loopback interface is the same for both of them, they might communicate directly. However, this is not an issue at all, as the entity would not gain additional powers over the SM by doing so. Instead, it would be only able to call its entry points, just as the EM.

4.3.4 Event Manager implementation

The EM was implemented using Rust. It provides two working modes: the difference relies on the loading process of the SMs. The default mode handles SGX enclaves, which are loaded using `ftxsgx-runner`, the default runner provided by Fortanix EDP. A runner is responsible for executing the SGX SM inside an enclave, as well as granting it the ability to communicate with the external world (Section 4.3.2). In the alternative mode, the SMs are not executed inside enclaves, but as normal applications. This operating mode was called *NoSGX* and does not provide any support for isolated execution and RA. Hence, it should only be used for testing purposes. More details are provided in Section 4.3.5.

The implementation of the EM is straightforward. A `TcpListener` waits for connections on the port passed as argument; when a new TCP connection is established, data is read and the event processed according to the protocol described in 4.2. Then, a response is sent back to the sender, if that event expected one. Finally, the connection is closed. Hence, each TCP connection carries only one event. The EM is single threaded: this choice was made for simplicity, as each TCP connection has a very short life and for this reason it was not really necessary to introduce parallelism to improve performance. However, it is effortless to make the EM multithreaded, if necessary.

4.3.5 SM implementation

One of our main contributions is the implementation of a framework to easily develop SGX SMs in the context of Authentic Execution, using Rust as programming language. The framework was called `rust-sgx-gen`, a Python module that allows a developer to define a SM by simply implementing its main logic: inputs, outputs, entry points, and other user-defined functions and data structures. The code needed to implement the Authentic Execution framework, including the management of events and the RA process, is automatically injected. `rust-sgx-gen` expects as input a Rust Cargo library (created with the command `cargo new <name> --lib`). Being a library means that the `main()` function does not exist, as it is added by the framework itself.

Inputs, outputs and entry points are detected by means of *annotations* in the code. In practice, such annotations are nothing more than comments with the format `//@ <keyword>`. The framework only searches for annotations in the `lib.rs` file of the input project: this limitation was introduced not only for simplicity, but also in order to keep all the annotations in the same file.

```
// Imports and other stuff

//@ sm_output(button_pressed)
//@ sm_output(output1)

//@ sm_entry
pub fn press_button(data : &[u8]) -> ResultMessage {
```

```
    debug!("ENTRYPOINT: press_button");

    if data.len() > 0 {
        return failure(ResultCode::IllegalPayload, None);
    }

    button_pressed(&[]);

    success(None)
}

//@ sm_input
pub fn input1(data : &[u8]) {
    info!("INPUT: input1");

    output1(data);
}

// User-defined functions and other stuff
```

Listing 4.4: Example of a possible lib.rs file provided as input to rust-sgx-gen

```
pub fn {name}(data : &[u8]) {{
    debug!("OUTPUT: {name}");
    let id : u16 = {id};

    handle_output(id, data);
}}
```

Listing 4.5: Stub of the output functions used by rust-sgx-gen

Listing 4.4 shows an example of a `lib.rs` file passed as input to `rust-sgx-gen`. As shown in the code, there are some helper functions and macros available to the developer: the functions `success` and `failure` create the `ResultMessage`[5] object, given a result code (for success messages the `Ok` code is implicit) and data (Section 4.2.2). In addition, simple macros for printing messages to standard output are provided. Outputs are declared with the annotation `//@ sm_output(<name>)`: no code needs to be provided, as the implementation of outputs is predefined. Listing 4.5 illustrates the stub that is parsed and then injected for each output annotation. Except for the `debug!` instruction, the output function simply works as a wrapper around `handle_output`, which encrypts `data` using the appropriate connection key and sends the payload to the EM as a *RemoteOutput* event (Sections 3.3 and 4.2.5). `{name}` and `{id}` are replaced by `rust-sgx-gen` with the actual name and ID of the output. Inputs are declared with the annotation `//@ sm_input`, which has to be placed right above the target function. Such function requires a specific *signature*: a *slice*[6] of `u8` elements as input and no return value. Entry points are declared with the annotation `//@ sm_entry`. The requirements are the same as inputs, except that the return value of the function in this case must be a `ResultMessage`. Each input, output and entry point is automatically assigned to a 16-bit ID, which is used internally by the

[5] https://github.com/gianlu33/rust-sgx-libs/blob/master/reactive_net/src/result_message.rs

[6] https://doc.rust-lang.org/book/ch04-03-slices.html

framework. Input/output IDs are used to map a certain input/output to a specific connection, while entry point IDs are used to identify the entry points when a user wants to call one of them remotely.

`rust-sgx-gen` allows two different working modes for a SM, called *NoSGX* and *SGX*. One could be misled by their name: the difference between the two only relies on how the Master Key is obtained, rather than whether the SM is executed inside an enclave or not. In the former case, the key is hardcoded inside the SM code, whereas in the latter case the key is obtained through RA. Notice that even if *SGX* mode is chosen, it does not mean that the SM cannot run as a native application. However, if the SM does not run inside an enclave, RA fails and the SM cannot work. On the other hand, if *NoSGX* mode is chosen and the SM is run inside an Enclave, the Deployer cannot attest it, even if the SM runs correctly and under the SGX TEE. The *NoSGX* mode was intended mainly for testing purposes. Building and running a SM in this way is much faster and it can significantly speed up the development process, as the code written by the Deployer is exactly the same for either working modes.

To provide the confidentiality and integrity guarantees needed for Authentic Execution (Section 3.3), SMs deployed as SGX enclaves use AES [39] in the Galois/Counter Mode (GCM) [40], an authenticated encryption algorithm which is widely adopted for its performance. In this implementation, 128-bit symmetric keys are used. Encryption is needed to secure *setKey* events between the Deployer and a SM, as well as connections between two SMs. About the latter, to provide compatibility between a SGX and a Sancus SM, support for SPONGENT cryptographic functions was included as well. When setting up a connection between two SMs, the Deployer can choose which algorithm to use between the supported ones (at the moment of writing, either AES or SPONGENT). The implementation of the Rust SPONGENT library used in this work was developed in a previous Master's Thesis [41].

Remote Attestation

RA is performed as soon as a SM compiled in *SGX* mode is loaded on a node. The SM, once it is executed, listens for a connection coming from the Deployer: this is the only time the two entities communicate directly (i.e., without going through the EM). We used in this Master's Thesis an existing implementation of RA for EDP enclaves [42], which is based on the official example provided by Intel [43].

The full RA process is the same described in Section 2.2.2, hence it is not explained again in this section. In this RA implementation, the verification of the *QUOTE* generated by the QE is done by the IAS. Therefore, the Deployer needs to provide his API keys from the Intel SGX Attestation Service Utilizing Enhanced Privacy ID (EPID) [44], as well as the IAS root certificate.

4.4 Authentic Execution in Sancus

The Authentic Execution framework for Sancus was implemented in previous work [2]. Nevertheless, some changes have been made, as described in the next sub-sections.

4.4.1 Event Manager implementation

The previous implementation of the untrusted components of the framework is based on Contiki [45]. However, support for the Sancus Microcontroller Unit (MCU) in Riot OS [46] is currently under development, hence we decided to port the EM implementation to this platform. Compared to Contiki, Riot offers several advantages, such as less resources used, full support to

multithreading and real-time capabilities. A thorough comparison of the two OSes is out of scope of this Master's Thesis, however it was already made in other research, such as [47].

Although the underlying operating system is different, the behavior of the EM remains the same in terms of offered functionalities. In the new implementation, three threads are used:

- *Main Thread.* This thread processes the events, which are read from a queue. Events are sent to the queue by the other threads. Hence, the main thread is independent of the media by which events are sent from other EMs.
- *UART Reader.* This thread reads for events received from the Universal Asynchronous Receiver-Transmitter (UART) interface. Each received event is then stored in the Main Thread's queue.
- *Event Generator.* This thread implements the *periodic events* feature (Section 4.1.3). If the feature is enabled and a SM has an entry point subscribed for periodic events, the Event Generator is responsible to generate and send the proper event to the Main Thread, with the specified frequency.

Since the Sancus version of Riot is currently in a development state, not all features were available or properly working at the time of the EM implementation. Particularly, we noticed issues with the `xtimer` module, used to switch threads and keep track of time. Hence, conditional compilation was used to either use or not `xtimer` in the EM code. Without this module, threads are switched with `thread_yield` and the Event Generator is disabled, as it would not be possible to deliver events with precise timing. The implementation of the prototype (Chapter 5) suffers from this limitation.

Events from the outside world, as already mentioned above, come only from the UART serial interface. Future work would be needed to use wireless technologies instead (e.g., Wi-Fi, Bluetooth), as Smart Environment applications typically use a wireless network to connect the components of a system. In this implementation, an abstraction layer over the UART was used to connect two different Sancus nodes (or a Sancus with a SGX node), using a TCP/IP stack. In essence, we developed a Python script called `reactive-uart2ip`, which is responsible for managing events to/from the board, running on the "host" machine where the Sancus MCU is connected through UART. The script takes as argument the path of the UART interface (e.g., `/dev/ttyUSB1`), a port (e.g., 5000) and the baud rate of the serial communication (e.g., 115200 bps). Events addressed to the host machine at the specified port (through TCP connections) are sent to the specified UART interface. Similarly, events coming from the UART interface are sent to the destination address indicated in the event header.

The implementation of `reactive-uart2ip` follows the same application-level protocol specified in Section 4.2. However, the script and the Sancus board exchange additional information to ensure that events are correctly handled, e.g., by providing IP address and port in events coming from Sancus that need to be sent to another node. The software also makes sure to avoid overlaps by sending events to the board one at a time, using synchronization techniques to manage multiple concurrent TCP connections.

4.4.2 SM implementation

The implementation of a SM was subject to a few changes in its stub code (i.e., the code injected by `sancus-compiler`[7]), mainly to implement many-to-many relationships (Section 4.1). As a

[7] https://github.com/sancus-tee/sancus-compiler

matter of fact, each entry in the `connections` array was changed to contain three additional 16-bit fields beside the connection key: the connection identifier, the input/output identifier and an unique nonce. Furthermore, `handleInput` and `handleOutput` were modified accordingly, to find the correct input given a connection ID in the former, and all the connection IDs given an output in the latter.

Other modifications in the stubs concerned the use of nonces: no checks on them were included in the initial code, making the implementation vulnerable to replay attacks. For instance, Listing 4.6 shows the original `handleInput` stub, which presents this vulnerability. If the same *RemoteOutput* event (Section 4.2.5) is sent multiple times (e.g., by an attacker that intercepted the original packet), the corresponding input of the recipient SM would be triggered multiple times, because `sancus_unwrap_with_key` is called without a prior control on the content of `payload`. To avoid this issue, the corrected version of `handleInput` added a check on the nonce (which is part of `payload`), to verify whether the same event was already processed or not.

```
void SM_ENTRY(SM_NAME) __sm_handle_input(uint16_t conn_id,
    const void* payload, size_t len)
{
    if (conn_id >= SM_NUM_INPUTS)
        return;

    const size_t data_len = len - AD_SIZE - SANCUS_TAG_SIZE;
    const uint8_t* cipher = (uint8_t*)payload + AD_SIZE;
    const uint8_t* tag = cipher + data_len;

    // TODO check for stack overflow!
    uint8_t* input_buffer = alloca(data_len);

    if (sancus_unwrap_with_key(__sm_io_keys[conn_id],
        payload, AD_SIZE, cipher, data_len, tag, input_buffer))
    {
        __sm_input_callbacks[conn_id](input_buffer, data_len);
    }
}
```

Listing 4.6: Vulnerable `handleInput` original stub: no checks on the nonce (which is part of the associated data) are performed, hence an attacker might perform a replay attack to trigger the same input multiple times.

Remote Attestation

As discussed in Section 2.2.1, RA in Sancus requires a challenge-response protocol where the SM produces a MAC using its master key $K_{N,SP,SM}$ over a fresh nonce sent by the Deployer, which then checks on his platform if the MAC is correct. In Authentic Execution, this procedure is executed during the establishment of a connection in a SM, with the *setKey* event. During this communication, the Deployer not only attests the SM, but also obtains a proof that the connection has been correctly established. If the SM has no connections, instead, it is never attested. However, the SM in this case would not provide any utility to the system, therefore RA is not needed in this case.

4.5 Deploying the system

To deploy an Authentic Execution system, a script called `reactive-tools`[8] was designed in previous work. This software takes as input a *deployment descriptor* (Section 3.3.3) and automatically configures the whole system. Three main steps are taken:

- *Deployment.* All the SMs are built and sent to the nodes where they belong to.
- *Remote Attestation.* Depending on the TEE, the RA process is explicit (e.g., Intel SGX, Section 4.3.5), or implicit (e.g., Sancus, Section 4.4.2).
- *Establishment of connections.* For each connection, a *Connect* message (Section 4.2.3) is sent to the source EM and `setKey` input is called on each of the two SMs.

These procedures are not strictly sequential, e.g., the *Connect* message of a connection might be sent to the EM before the deployment of the two involved SMs. However, `setKey` cannot clearly be called on a SM if it is not yet deployed. The implementation of `reactive-tools` is made using asynchronous tasks[9], with special constructs (`async/await`) to satisfy all the rules of precedence such as the ones just mentioned.

Due to the modifications introduced in this Master's Thesis, we updated `reactive-tools` to handle new information coming from the deployment descriptor:

- *SGX SMs and nodes.* SGX components are marked with `"type": "sgx"` in the deployment descriptor. Functions have been implemented to handle this TEE. Additionally, `"nosgx"` type can be also specified to deploy a SM as a native application instead of an enclave (Section 4.3.5).
- *Periodic events.* A task responsible for registering periodic events on the nodes was implemented (Section 4.1.3). This task is executed after all the connections have been established.
- *Different encryption algorithms.* Information about the encryption algorithm to use is added to each connection entry (e.g., `"encryption": "aes"`). The software checks whether the encryption algorithm is supported by both modules or not, generating an error in the latter case.

[8] https://github.com/sancus-tee/reactive-tools

[9] https://docs.python.org/3/library/asyncio.html

Chapter 5

Prototype

In this Master's Thesis, we developed a prototype to evaluate the framework described in the previous chapters. This chapter covers this part and it is structured as follows: Section 5.1 introduces the use case, pointing out its critical aspects and the benefits that our framework would bring in terms of security. Section 5.2 shows an abstract view of the components of the distributed application, along with their connections. Section 5.3 provides a concrete implementation of the prototype, given the abstract model, in three different variants. Section 5.4 gives an evaluation of the prototype's code, whereas Section 5.5 discusses about its performance. Finally, Section 5.6 presents a security discussion about the main possible threats of our use case, explaining whether our solution offers protection to such threats or not.

5.1 Motivation

The prototype implements an application for a simple smart irrigation system. In the field of agriculture, proper irrigation of the crop is an important task, as each product needs a different timetable and amount of water. This requires a large effort from the farmer if the irrigation is performed manually. Besides, a manual supply of water might lead to additional issues: firstly, a wrong supply of water (e.g., incorrect quantity or improper time of the day) might damage the crop, causing economic loss. Secondly, if the supplied quantity of water is higher than needed, there is a waste of resources: this is particularly critical in places where water is scarce.

For all of these reasons, different kinds of automatic irrigation systems have been proposed in recent years. There are essentially two types of smart irrigation controllers: weather-based and sensor-based[1]. Particularly, research has shown that it is possible to build simple and effective sensor-based systems that partially or totally solve the problems mentioned before.

For small systems, it is possible to use a single microcontroller that includes the sensing peripherals, the logic to start/stop the irrigation and the irrigation actuator itself [48, 49]. That is, each microcontroller works on its own and does not depend on external entities. This brings an advantage from a security point of view, because an attacker has a very low attack surface in this scenario, as his only option is to physically tamper with the device.

However, in bigger systems this solution is not enough: for instance, in wide areas it might be needed to have sensors placed in different positions. Some scenarios also require different types of sensors. All data from the various sensors needs to be gathered together and processed

[1] https://www.hydropoint.com/what-is-smart-irrigation/

by a centralized controller, which is responsible for sending the commands to actuate one or more output devices. All the components of the system are connected together, usually using a wireless network [50, 51].

Unlike the previous one, this implementation is much more attractive for an adversary. If no security mechanisms are implemented, an attacker might intercept and alter messages exchanged over the network, as well as providing forged ones. This way, the attacker might be able to control the irrigation system either directly (e.g., by controlling the actuator) or indirectly (e.g., by providing fake sensor data to trick the controller to enable/disable the water supply). Of course, this is a very high threat which might cause significant financial and reputational loss to the farmer. For this reason, a secure implementation of such applications is needed. Hence, our prototype tries to achieve this goal.

5.2 Building blocks

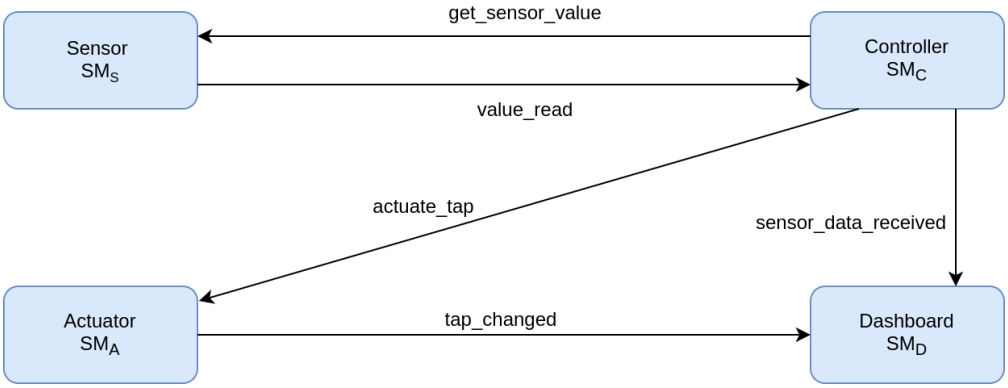


Figure 5.1: Abstract view of the components of the prototype and their connections.

The abstract model of the prototype is shown in Figure 5.1. It is composed of four Software Modules (SMs):

- *Sensor (SM_S).* This SM should be directly connected to a soil moisture sensor connected to the ground, which provides information about the current moisture level of the soil. The SM provides sensor data *on demand*, triggered by SM_C.
- *Actuator (SM_A).* This SM is directly connected to the irrigation system, which is turned on or off according to the controls received from SM_C.
- *Controller (SM_C).* This SM includes the main logic of the application. Periodically, it requests sensor data from SM_S. As soon as the value is received, it is compared to two predefined thresholds: if the value is above the upper threshold, it means that the soil is too moist, therefore the irrigation system needs to be turned off. On the other hand, if the value is below the lower threshold, the soil is too dry: hence, the irrigation system needs to be turned on. In both cases, a command is sent to SM_A. If the value is between the two thresholds, nothing happens.
- *Dashboard (SM_D).* This is an optional SM, to allow interaction between the system and an external user (e.g., the farmer). In this prototype implementation, it only provides

information about the value of the last sensor data read and the current state of the irrigation system. However, more advanced implementations could allow the user to directly control the system through specific entry points.

As shown in Figure 5.1, connections are established between the SMs, to implement the logic of the application as described above: `get_sensor_value` and `value_read` between SM_C and SM_S, `sensor_data_received` from SM_C to SM_D, `actuate_tap` from SM_C to SM_A and `tap_changed` from SM_A to SM_D. In addition, the soil moisture sensor and the irrigation actuator need to be connected to the system so that only SM_S must be able to read data from the soil moisture sensor and, more importantly, only SM_A must be able to control the irrigation system. How this functionality is implemented depends on the architectures chosen.

5.3 Implementation

In the prototype, we implemented SM_S to simulate a soil moisture sensor: rather than reading values from a physical device, it produces "random" values that simulate the conditions of the soil over time. In practice, SM_S keeps a counter that is updated every time a sensor reading is requested. It is incremented if the irrigation system is on (over time, the soil becomes more and more wet), and decremented otherwise (over time, the soil becomes more and more dry). This is a very simple implementation that does not take into account external conditions (e.g., rain); however, it is sufficient to evaluate the prototype. Additionally, the irrigation system has been replaced by a simple LED, connected to a Sancus board using Secure I/O (Section 2.2.1). The LED is piloted by a SM called *LED_MMIO*, whose data section is mapped on the LED's MMIO region, gaining exclusive access to it. An additional SM (called *LED_driver*) was used to control *LED_MMIO*, to add a separation layer between the physical device and SM_A. In this way, if the device changes (e.g., from an LED to a real irrigation system), SM_A's code remains unchanged.

SM_C uses the *periodic events* feature (Section 4.1.3) to request sensor data from SM_S every second. That is, the registered entry point simply generates the `get_sensor_value` output, initiating the exchange of events between the SMs.

5.3.1 Configurations

The prototype was implemented and evaluated using three different configurations, as shown in Figure 5.2:

- *Distributed, wide ($Conf_W$).* SM_S and SM_A are deployed to two different Sancus nodes, whereas SM_C and SM_D are deployed to the same SGX node. This is a possible solution for big systems, where the input and output physical devices are not necessarily close to each other, or where numerous sensors and actuators are used to cover all the area. Furthermore, the SGX node provides enough hardware and software resources to allow additional tasks: it might be used as a storage unit or also to perform expensive computation (e.g., to compute statistics using Machine Learning, in order to find the optimal values of the two thresholds).

- *Distributed, narrow ($Conf_N$).* This configuration is similar to $Conf_W$, but SM_S and SM_A are deployed on the same Sancus node. This might be for instance the case of an irrigation system of a flowerpot, where the system is small enough that the physical devices are spatially close to each other.

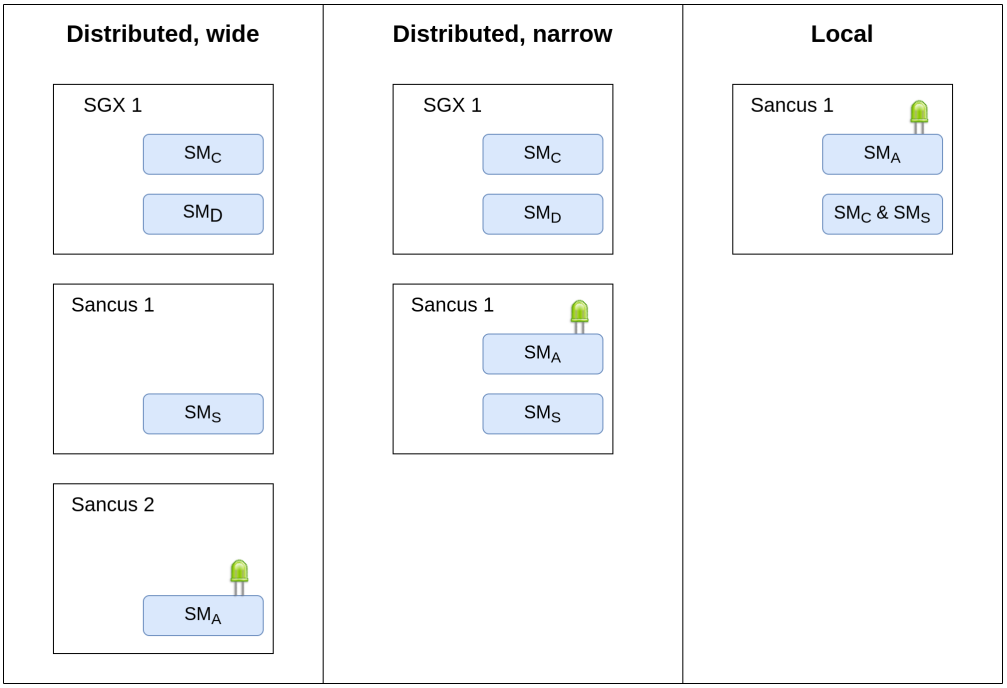


Figure 5.2: Deployment of the prototype SMs in three different configurations.

- *Local (Conf$_L$).* If the system is simple enough, SM_C's logic might be included in SM_S. Benefits are that the whole system is deployed on the same Sancus node, which means faster transmission of events and less resources used (no SGX nodes are needed). Furthermore, another advantage is that the system is more resilient: if the network is down, due to either a malfunction or an attack, the prototype can still continue to work. However, due to the constraints in Sancus mentioned in Section 4.4.1, it is not possible to use the periodic events feature, hence an external SM was used to trigger sensor readings. For simplicity, SM_D is not present in this configuration.

With respect to the encryption algorithms used, AES-128 is used between two SGX SMs, whereas in all the other cases (Sancus-SGX and Sancus-Sancus) SPONGENT with 128 bits of security is used.

5.4 Code evaluation

This section discusses the implementation details of each component of the prototype, considering both the Source Lines of Code (SLOC) and the binary size. The former are calculated using `cloc`[2], considering `.rs` and `.toml` files in Rust applications, and source and header files (`.c` and `.h`) in C code. The latter, instead, is computed using `size`[3].

[2] https://github.com/AlDanial/cloc

[3] http://manpages.ubuntu.com/manpages/bionic/man1/arm-none-eabi-size.1.html

Furthermore, we assessed the TCB size for each architecture (Intel SGX, Sancus), and compared with the untrusted software. Reducing the TCB size as much as possible is a major objective of this Thesis work: the smaller it is, the lesser attack surface an adversary has. Thus, it is easier to check and verify the security of such systems.

5.4.1 Intel SGX implementation

Module	SLOC	Binary size (KB)
EM	531	1316
SM_C	33	4507
SM_D	53	4467

Table 5.1: Source Lines of Code (SLOC) and binary size of the Rust implementation of the Event Manager (EM) and the two SM of the prototype, SM_C and SM_D.

Table 5.1 shows the size of the SGX components of the prototype. The SGX's Event Manager (EM), which is the *untrusted* component that processes the events (as described in Section 3.1), required 531 SLOC (external dependencies excluded), generating a binary file of 1316 KB. On the other hand, the *trusted* units SM_C and SM_D required only 33 and 53 SLOC respectively. Yet the binary size is not so small, 4507 and 4467 KB, due to the stub code injected by `rust-sgx-gen` with its external dependencies and the overhead for running the SMs inside an enclave. However, such sizes widely fit into the EPC, the protected region of memory that contains SGX enclaves (Section 2.2.2).

It is worth noting that the biggest impact derives from the Remote Attestation (RA) process (Section 4.3.5), which is not caused by the RA code itself, but rather by the huge number of the external dependencies that it needs for executing. In fact, as shown in Table 5.2, the actual RA code (without considering the dependencies) only consists of 678 SLOC.

This table also reveals another important aspect: the code we wrote for each component of the prototype is very tiny compared to the total size of a SM. For instance, SM_C's logic is only 9% of the main code (logic + stub code), and less than 0.5% of the total SLOC.

5.4.2 Sancus implementation

Table 5.3 shows a comparison between trusted and untrusted components of the Sancus implementation. In terms of lines of code, the untrusted part (10761 SLOC) is dominant compared to the trusted one (108 SLOC): the latter is only 1% of the total code. However, it must be considered that the SLOC of the trusted components do not include stub code and other libraries. In fact, the binary size comparison is less prominent, with respectively 34794 and 8428 bytes, making the trusted part 19,5% of the total size. Nevertheless, the TCB reduction is still relevant, which results in a more limited attack surface on each node.

It is noteworthy that having a separate driver for the LED (*LED_driver*), instead of including the code in SM_A, is not the most efficient choice: as a matter of fact, an additional SM means more memory used (both RAM and ROM) and a higher performance overhead due to the transmission of events from SM_A to *LED_driver*.

Module	Source	SLOC
Stub code	Thesis work	320
reactive_crypto	Thesis work	144
reactive_net	Thesis work	262
lazy_static[1]	Open source library	163
base64[2]	Open source library	4452*
aes_gcm[3]	Open source library	263
spongent_rs[4]	External code	316
rust_sgx_remote_attestation[5]	External code	678
TOTAL		**6598**

* this number includes about 1000 SLOC of test code, and 2000 SLOC of tables.

[1] https://docs.rs/lazy_static/1.4.0/lazy_static/
[2] https://docs.rs/base64/0.12.3/base64/
[3] https://docs.rs/aes-gcm/0.6.0/aes_gcm/
[4] https://github.com/stenverbois/spongent-rs
[5] https://github.com/ndokmai/rust-sgx-remote-attestation

Table 5.2: Source Lines of Code (SLOC) of the SGX's Authentic Execution framework implementation, with its main dependencies. External dependencies of these modules were not considered in the computation.

5.5 Performance evaluation

To evaluate the performance of the prototype, we collected time measurements to determine the responsiveness of the system. We tested all the three configurations introduced in Section 5.3. In particular, we calculated the elapsed time (called Round-Trip Time (RTT)) from when a sensor value is requested by SM_C (generating `get_sensor_value`) to when SM_A activates the LED. Due to the issues with timers in Sancus, all the time measurements were carried out on the SGX's side. Therefore, the end time does not match with the exact moment when the LED is turned on or off, but rather with the moment when SM_D receives `tap_changed`, which includes the transmission time of that event and its processing.

Table 5.4 shows the average RTT calculated over 15 measurements for each of the three configurations. $Conf_W$ and $Conf_N$ share almost the same RTT: 379.5 ms and 371.7 ms respectively. Indeed, these two configurations do not differ from a performance point of view: the two Sancus nodes are connected using the same media to the SGX node, hence it does not matter where SM_S and SM_A are actually located from the two SGX SMs' perspective. Furthermore, SM_S and SM_A do not exchange events each other. On the other hand, $Conf_L$ is, as expected, more efficient: with an average RTT of 108 ms, it is more than three times faster than the other two configurations.

In addition to measuring the performance of the prototype, other experiments were conducted, in order to understand what are the most critical components in terms of performance and give

Module	SLOC	Binary size (B)
Untrusted components		
Riot OS	9713	-
EM	1048	-
TOTAL	**10761**	**34794**
Trusted components		
SM_S	25	2925
SM_A	24	2916
LED_driver	30	2471
LED_MMIO	29	116
TOTAL	**108**	**8428**

Table 5.3: Source Lines of Code (SLOC) and binary size of the Sancus trusted and untrusted components.

Configuration	Average RTT (ms)
$Conf_W$	379.6
$Conf_N$	371.7
$Conf_L$	108

Table 5.4: Average RTT measured for the three configurations of the prototype. Start time is when a sensor data is requested by SM_C, end time is when SM_D is notified by an LED state change.

some hints for future optimizations.

5.5.1 Intel SGX impact

The first additional experiment was carried out in order to measure the impact of Intel SGX on the overall performance of the prototype. In particular, we made the same measurements explained previously for $Conf_W$ and $Conf_N$, with the only difference that SM_C and SM_D are deployed as native applications instead of SGX enclaves (using the NoSGX mode).

Table 5.5 shows the average RTT computed over 15 measurements. In both configurations, there is a substantial improvement on the performance of the system: in fact, the average RTT is 272.8 ms for $Conf_W$ and 272.47 ms for $Conf_N$. Compared to these numbers, the protection offered by Intel SGX in the prototype worsen the performance by respectively 39% and 36%. Indeed, we expected a performance degradation caused by Intel SGX. The calculated overhead is a result of two main aspects: first of all, the actual overhead of executing CPU instructions and accessing encrypted memory in an enclave. Secondly, the cost associated with entering and

Configuration	Average RTT (ms)
$Conf_W$	272.8
$Conf_N$	272.47

Table 5.5: Average RTT measured for $Conf_W$ and $Conf_N$, where SM_C and SM_D are deployed as native applications (NoSGX mode).

exiting the enclave (through the so-called ECALLs/OCALLs). A thorough analysis of the SGX performance is out of the scope of this Master's Thesis, however it is well researched [52, 53, 54].

5.5.2 Authentic Execution impact

We made an effort to measure the impact of the Authentic Execution framework. We paid particular attention to compare the two encryption algorithms used, AES and SPONGENT.

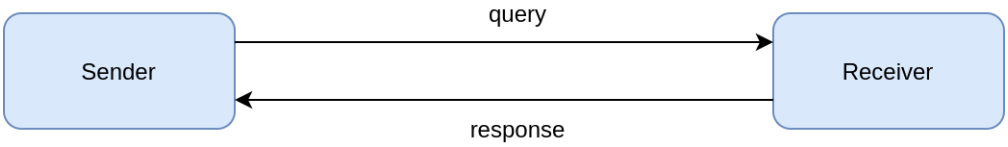


Figure 5.3: Toy example used to evaluate the impact of the Authentic Execution framework.

Figure 5.3 shows the example used for these measurements. Two SMs are involved, called Sender and Receiver, which are connected to each other by two different connections: `query`, from Sender to Receiver, and `response`, from Receiver to Sender. The example works in the following way: An entry point in Sender triggers the emission of `query`, whose event is sent to Receiver which, as a consequence, generates `response`. Experiments measured the RTT between the two SMs: from the moment when Sender's entry point is called to the moment when the same SM receives the `response` event. No data is exchanged between the two SMs, in both connections. Two outputs and two inputs are present in the example: this means that a total of two encryptions and two decryptions are computed for each query-response cycle.

This example was tested in three different configurations:

- *SGX, using AES ($Conf_{AES}$).* Sender and Receiver are two SGX SMs deployed on the same node, using AES-128 as encryption algorithm.
- *SGX, using SPONGENT ($Conf_{SPONGENT}$).* Similar to $Conf_{AES}$, but SPONGENT is used instead of AES, with 128 bits of security.
- *Hybrid ($Conf_{Hybrid}$).* Sender is a SGX SM, Deployer is a Sancus SM. Naturally, SPONGENT-128 is used for encryption, since it is the only algorithm supported by both architectures.

Table 5.6 shows the average RTT calculated over 15 measures for each of the three configurations. First of all, $Conf_{AES}$'s results show an average RTT of only 9.73 milliseconds: thanks to the hardware support, in fact, AES operations are very fast in a SGX platform. Different is the case of SPONGENT, which is completely implemented in software. Results show that $Conf_{SPONGENT}$ executed 36 times slower, with an average RTT of 354.47 ms. The encryption

Configuration	Average RTT (ms)
$Conf_{AES}$	9.73
$Conf_{SPONGENT}$	354.47
$Conf_{Hybrid}$	244.07

Table 5.6: Average Round-Trip Time (RTT) between two SMs deployed in three different configurations, to assess the performance of AES and SPONGENT encryption algorithms.

overhead in this case is not negligible and hugely affects the overall performance of the system. Fortunately, there is no need to use SPONGENT between two SGX SMs, however this encryption algorithm must be used when the connection is established with a Sancus SM. As a consequence, performance issues arise in those applications. $Conf_{Hybrid}$, in fact, results with an average RTT of 244.07 ms, which is 25 times slower than $Conf_{AES}$, but 45% faster than $Conf_{SPONGENT}$, even if the measurements include also the transmission time of events between SGX and Sancus nodes. This suggests that SPONGENT operations are faster on Sancus's side: in fact, they are performed in hardware.

5.5.3 Transmission time

Additional tests analyzed the performance of the transmission channel between a Sancus node and a SGX node. Recall from Section 4.4.1 that, in this Master's Thesis work, Sancus boards are connected to the SGX machine using UART as a communication media, and that events from different nodes are exchanged through TCP/IP packets, running `reactive-uart2ip` tool for each Sancus device to map the UART connection with a TCP/IP socket. Additionally, the UART baud rate is set to 57600 bps.

The tests measured the average RTT of a communication between sample Sancus and SGX endpoints (without using the Authentic Execution framework), in two different cases: firstly using TCP/IP, then using the serial port directly, without any other abstraction layer. The total length of the payload is 23 bytes in both cases: this number was not randomly chosen, in fact it is the length of an Authentic Execution *RemoteOutput* event with no attached data (Section 4.2.5), using 128 bits of security (total length: 7 bytes header + 16 bytes MAC). This allows to deduce the transmission time of the events exchanged during the $Conf_{Hybrid}$ tests in the previous section, as the size of packets is the same.

Configuration	Average RTT (ms)
TCP/IP	28.15
Bare UART	16.19

Table 5.7: Average Round-Trip Time (RTT) between a SGX and a Sancus node, to measure the transmission time.

Results are shown in Table 5.7: the average RTT over 15 measurements using TCP/IP is 28.15 ms, 74% slower than using bare UART (16.19 ms). Two main aspects can be observed: first of all, that the serial communication is, in general, pretty slow. Due to the very low baud rate, in fact, even a transmission of a handful of bytes takes some milliseconds. Secondly, that

the TCP/IP abstraction over the UART introduces a not insignificant overhead. Indeed, the tests have shown that, for 23 bytes of data, a single transmission is almost 6 ms slower.

5.5.4 Discussion

Some conclusions can be drawn from these experiments. Firstly, the actual implementation of the prototype might be already capable of satisfying the performance requirements of a smart irrigation system: the real time constraints in this case are not so stringent, as the response time of the system might be acceptable if it is equal or smaller than a second. The measured performance of the prototype largely satisfies these restrictions, hence it is expected that a real-world implementation of this prototype would still be feasible for this scenario.

Secondly, for applications where the response time needs to be smaller (in the order of milliseconds), the actual implementation might not be feasible anymore. However, the experiments carried out in this work revealed which are the most critical tasks in terms of performance: the overhead caused by the enclaved execution in Intel SGX and the software implementation of cryptographic algorithms (SPONGENT, in this case). This is a starting point for future research, which might come up with optimizations that would improve the overall responsiveness of the system.

In summary, we can conclude that this prototype is feasible for a smart irrigation application, because the time constraints of such use case are widely affordable for our implementation.

5.6 Security evaluation

In this section, we assess the security properties of the prototype. As it implements a smart irrigation system, the evaluation takes into account some hypothetical threat scenarios of a generic Smart Farming application, as described in [9]. The paper highlights eight different cases under the CIA triad (Confidentiality, Integrity, Availability). Table 5.8 shows a summary of these scenarios, each of which is analyzed thoroughly in the next sub-sections. All the security aspects of our implementation derive from the Authentic Execution framework, described in Section 3.3.

Table 5.8: Security analysis of the prototype, given eight hypothetical threat scenarios. For each scenario, the table shows its type (Confidentiality, Integrity or Availability) and whether the prototype offers protection against such threat or not.

No.	Scenario	CIA	Support
1	Confidential data stored on a server is leaked or stolen	C	Yes
2	Confidential data stored on foreign cloud services is accessed by foreign governments	C	Yes
3	Attackers introduce fake data on the system or modify existing data	I	Yes
4	Fake sensor data is introduced on the system to influence decision support systems or machine learning software	I	Yes

No.	Scenario	CIA	Support
5	Sensor data or algorithms are altered by an attacker	I	Yes
6	Software does not work properly, due to a bad update or a pirated version that contains a backdoor used to turn off the system	A	Partial
7	Foreign agricultural equipment does not work properly, due remote tampering	A	No
8	External events (e.g., natural disasters) bring down the communication systems	A	Minimal

5.6.1 Confidentiality scenarios

Scenario 1 describes a situation where confidential data is leaked or stolen from a storage unit (e.g., a database). Our implementation offers protection against this threat, since sensitive data can be encrypted using a SM's Master Key. This process is also known as *sealing*, particularly used by Intel SGX (Section 2.2.2).

Scenario 2 is similar to the first, however in this case data is stored on foreign cloud services. In these circumstances, the foreign government might want to access such data, e.g., to assess the agricultural yields of its competitors. The solution is the same as Scenario 1: confidential data can be encrypted, meaning that no one except the owner of the system would be able to access it.

5.6.2 Integrity scenarios

Scenarios 3 and 4 describe the situation where data is manipulated: either fake data is introduced on the system (e.g., forged sensor readings), or original data is altered (e.g., during the transmission of information between two SMs). This might cause several issues to the system, e.g., fake sensor data might influence decision support systems such as the logic to enable or disable an actuator. Fortunately, our solution strongly focuses on data integrity, as every event is authenticated using the *connection key*, known only by the two SMs that participate in the communication and the Deployer. Any alteration of this data results in a decryption failure and the event is discarded. Physical attacks on sensors, instead, might succeed. However, this kind of attacks is explicitly out of scope for our system, i.e., an attacker cannot tamper with the hardware.

Scenario 5 illustrates the case where, in addition to the manipulation of data, algorithms or software are tampered with. Indeed, in our implementation a SM might be altered either during the transmission of the binary to a node or during the loading process. However, RA is used to obtain a proof of authenticity of the SM at runtime. Additionally, the SM runs in a protected environment, therefore it cannot be manipulated during its execution.

5.6.3 Availability scenarios

Scenario 6 reveals possible issues related to the software installed on the system. Software might not work properly, due to a bad update which might introduce a bug or a pirated version of the same software, that might contain malicious code. Our solution offers partial protection in

this case: it is not possible to install a modified version of a SM, because RA would fail if the software is not authentic. Nevertheless, if the code written by the Deployer himself contains bugs or backdoors, the system would be unavoidably exposed.

In Scenario 7, foreign agricultural equipment (i.e., the hardware) is compromised by foreign governments, to cause damage during specific periods (e.g., planting, harvesting). Our implementation does not offer any kind of protection in this case: nodes are *untrusted* components and there is no guarantee that a SM installed on a node is actually executed.

Finally, Scenario 8 considers the case where the communication systems are broken due to external events (e.g., natural disasters). By being a distributed application, our solution is hugely affected if the network is down, as all events exchanged by two SMs in different nodes are inevitably lost. However, an application entirely deployed on the same node (e.g., $Conf_L$ of our prototype) might still be able to work.

5.6.4 Discussion

As shown in this analysis, the framework provides strong confidentiality and integrity guarantees. On the other hand, there is little to no support to availability. However, this is not a surprise, as availability was ruled out of scope of this Master's Thesis work from the beginning (Section 3.3.2).

Chapter 6

Conclusion

In this Master's Thesis, we extended the Authentic Execution framework in order to support a real Smart Environment application. The most important contribution of our work is the support for Intel SGX TEE, which allows the deployment of a *heterogeneous* system composed by both low-end embedded systems and high-end computation systems. We revised the implementation of Authentic Execution for Sancus to provide compatibility with SGX. Furthermore, we enriched the tools to automatically deploy a system, reducing the developer's effort to the minimum necessary. Finally, we designed and evaluated a prototype for a secure smart irrigation system.

The security evaluation showed that the prototype provides strong confidentiality and integrity guarantees. The isolation mechanisms of a TEE prevents code and data to be leaked and altered inside a platform. Remote Attestation, instead, ensures the authenticity of every module in the system. Additionally, encryption techniques protect the communication between two modules, whereas Sancus' Secure I/O functionality establishes a trusted path between a Sancus module and an I/O device. The implementation of the Authentic Execution framework for SGX was made entirely in Rust, a modern and secure programming language that prevents many low-level attacks such as buffer overflows.

On the other hand, experimental tests to measure its performance revealed mixed results: our implementation should be largely feasible for a smart irrigation system, which does not have strict real time requirements. For other applications, instead, our solution might not meet these constraints, as our tests showed the weaknesses of our prototype from a performance point of view. We believe that future optimizations can improve the performance of our implementation, allowing the Authentic Execution framework to be feasible in more scenarios.

6.1 Limitations and future work

This section discusses the limitations of our implementation and provides some hints for future work.

Improving the performance

The performance evaluation in Section 5.4 revealed that our implementation, in general, is not suited for applications with strict real time constraints. Our tests showed that the bottleneck of our solution is the software implementation of the SPONGENT cryptographic library, used by SGX SMs to communicate with Sancus SMs. Indeed, past work already proved the poor performance of a SPONGENT software implementation [55]. Hence, a different approach is

needed to improve the performance of the whole system. Since cryptographic functions are computationally expensive, the ideal solution would be executing them in hardware. Further research is needed to develop a feasible solution of this issue. A possible one would be the implementation of a crypto unit based on AES in Sancus.

A complete prototype

The prototype we developed for a smart irrigation system and discussed in Chapter 5 does not use real I/O devices, except for an LED to show whether the irrigation needs to be turned on or off at a particular time. The soil moisture sensor is emulated in software, by means of a simple logic. Furthermore, the Sancus devices are connected to the SGX backend through UART (Section 4.4) instead of a wireless connection. Additional work might be carried out in future to extend the current implementation of the prototype in order to remove these limitations and support a real application (e.g., watering a flowerpot). In such scenario, a performance evaluation would certainly be more accurate.

Different communication media

All the components of our system use a TCP/IP stack. This might be acceptable for high-end nodes, however embedded systems do not always use these protocols to communicate with the external world. There are many different wireless technologies that a microcontroller can adopt, such as Wi-Fi, Bluetooth, ZigBee, LoRa, etc. Therefore, the Authentic Execution framework should support different media to provide compatibility with as much devices as possible. In particular, modifications are needed in the application-level protocol for the *Connect* message (Section 4.2.3), as well as in the SGX and Sancus implementations to receive events from multiple media.

SGX extensions

The SGX implementation of the Authentic Execution framework might be improved not only to support different communication media (as described above), but also to provide new features. A SGX SM, to be initialized correctly, needs to perform Remote Attestation with the Deployer: in the current implementation, the process must be repeated for every execution of the SM, to re-establish a secure channel between the two entities. Additionally, also the symmetric keys of the connections have to be sent to the SM every time. To mitigate this problem, the *sealing* feature (Section 2.2.2) might be used to securely store all the keys on disk. With data sealing, the SM could retrieve the keys at startup, without the need of exchanging further messages with the Deployer.

Data sealing might be used for storing additional information as well, such as sensor data. Alternatively, encrypted databases implemented with Intel SGX might be used for this purpose, such as EnclaveDB [56] and StealthDB [57].

The prototype used a very basic *dashboard* to interact with the system (Section 5.2). The dashboard was implemented as a regular SM that provides entry points to get information about the irrigation system. However, more sophisticated interfaces might be implemented, e.g., a website that dynamically shows the current status of the system, as well as some statistics about the data collected and an interface to send commands. This might be achieved by having a SGX SM which additionally implements a web server, with TLS termination inside the enclave [58, 59]. The owner of the system could then connect to the web server using his browser.

Naturally, authentication mechanisms should be implemented to identify a client that wants to access sensitive data or send commands.

Dealing with availability

As stated in Section 3.3.2, availability is not in the scope of this Master's Thesis work. Nevertheless, some improvements might be taken into account to improve the robustness of the system. Firstly, the Authentic Execution framework uses nonces to provide freshness and avoid replay attacks. In our implementation, if an event is lost or intercepted on the network, the two SMs of the connection would not be able to communicate anymore, because their nonces would be out of sync from that moment (the destination SM, by not receiving the event, would not increase its nonce). Hence, recovery mechanisms might be implemented to deal with this issue. A second improvement might be the introduction of some backup logic on each SM to be executed when the system is not working as expected. For instance, if an actuator (e.g., the irrigation system) does not receive events for a very long time, it might be an indicator that either the network or the controller is not working properly. In this case, code might be executed on the SM to activate/deactivate the actuator autonomously, e.g., every few hours.

Bibliography

[1] Diane Cook and Sajal Kumar Das. *Smart environments: technology, protocols, and applications*, volume 43. John Wiley & Sons, 2004.

[2] Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. Authentic execution of distributed event-driven applications with a small TCB. In *13th International Workshop on Security and Trust Management (STM'17)*, volume 10547 of *LNCS*, pages 55–71, Heidelberg, 2017. Springer.

[3] Chris Mitchell. *Trusted computing*, volume 6. Iet, 2005.

[4] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[5] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2):63–81, 2011.

[6] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):7:1–7:33, September 2017.

[7] Intel. Intel Software Guard Extensions. https://software.intel.com/en-us/sgx. Accessed: 2020-08-18.

[8] Athanasios T Balafoutis, Bert Beck, Spyros Fountas, Zisis Tsiropoulos, Jürgen Vangeyte, Tamme van der Wal, I Soto-Embodas, Manuel Gómez-Barbero, and Søren Marcus Pedersen. Smart farming technologies–description, taxonomy and economic impact. In *Precision Agriculture: Technology and Economic Perspectives*, pages 21–77. Springer, 2017.

[9] United States. Department of Homeland Security. Office of Intelligence and Analysis. Threats to Precision Agriculture. *2018 Public-Private Analytic Exchange Program*, 2018.

[10] Wikipedia. Trusted Computing. https://en.wikipedia.org/wiki/Trusted_Computing. Accessed: 2020-08-18.

[11] Jan Tobias Mühlberg and Jo Van Bulck. Tutorial: Building distributed enclave applications with sancus and sgx. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN'18)*. IEEE/IFIP, 2018.

[12] Trusted Computing Group. https://trustedcomputinggroup.org. Accessed: 2020-08-18.

[13] Steven L Kinney. *Trusted platform module basics: using TPM in embedded systems*. Elsevier, 2006.

[14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *International Workshop on Selected Areas in Cryptography*, pages 320–337. Springer, 2011.

[15] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. Spongent: A lightweight hash function. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 312–325. Springer, 2011.

[16] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In *International Conference on Security and Privacy in Communication Systems*, pages 344–361. Springer, 2010.

[17] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[18] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, page 7. Citeseer, 2013.

[19] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper*, 1(1-10):119, 2016.

[20] Introduction to Intel® SGX Sealing. https://software.intel.com/content/www/us/en/develop/blogs/introduction-to-intel-sgx-sealing.html. Accessed: 2020-08-18.

[21] Naiqian Zhang, Maohua Wang, and Ning Wang. Precision agriculture—a worldwide overview. *Computers and electronics in agriculture*, 36(2-3):113–132, 2002.

[22] Jonathan M McCune, Adrian Perrig, and Michael K Reiter. Bump in the ether: A framework for securing sensitive user input. In *USENIX Annual Technical Conference, General Track*, pages 185–198, 2006.

[23] Jonathan M McCune Adrian Perrig and Michael K Reiter. Safe passage for passwords and other sensitive data. In *Proceeding of the 16th annual network and distributed system security Symposium*, 2009.

[24] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[25] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 261–268, 2017.

[26] Samuel Weiser. *Secure I/O with Intel SGX*. PhD thesis, Institute of Applied Information Processing and Communications (7050), 2016.

[27] Yeong Jin Jang. *Building trust in the user I/O in computer systems*. PhD thesis, Georgia Institute of Technology, 2017.

[28] Saba Eskandarian, Jonathan Cogan, Sawyer Birnbaum, Peh Chang Wei Brandon, Dillon Franke, Forest Fraser, Gaspar Garcia Jr., Eric Gong, Hung T. Nguyen, Taresh K. Sethi, Vishal Subbiah, Michael Backes, Giancarlo Pellegrino, and Dan Boneh. Fidelius: Protecting User Secrets from Compromised Browsers. *IEEE Symposium on Security and Privacy (SP)*, 2019.

[29] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2(12):10–1571, 2006.

[30] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[31] Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. Towards availability and real-time guarantees for protected module architectures. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 146–151, 2016.

[32] Jon Bell. Network protocols used in the automotive industry. *The University of Wales, Aberystwyth*, pages 07–24, 2002.

[33] Jin-Shyan Lee, Yu-Wei Su, and Chung-Chou Shen. A comparative study of wireless protocols: Bluetooth, uwb, zigbee, and wi-fi. In *IECON 2007-33rd Annual Conference of the IEEE Industrial Electronics Society*, pages 46–51. Ieee, 2007.

[34] Rust. Rust programming language. https://www.rust-lang.org. Accessed: 2020-08-18.

[35] Intel(R) Software Guard Extensions SDK. https://software.intel.com/en-us/sgx-sdk/documentation. Accessed: 2020-08-18.

[36] Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei, and Huibo Wang. Poster: Rust sgx sdk: Towards memory safety in intel sgx enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2491–2493, 2017.

[37] Fortanix. Fortanix Enclave Development Platform – Rust EDP. https://edp.fortanix.com. Accessed: 2020-08-18.

[38] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[39] Joan Daemen and Vincent Rijmen. Reijndael: The advanced encryption standard. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 26(3):137–139, 2001.

[40] David McGrew and John Viega. The galois/counter mode of operation (gcm). *submission to NIST Modes of Operation Process*, 20:10, 2004.

[41] Sten Verbois. Safe interacting enclaves for heterogeneous protected module architectures. Master's thesis, KU Leuven, 2018. https://distrinet.cs.kuleuven.be/software/sancus/publications/verbois18thesis.pdf.

[42] Ko Dokmai. Remote attestation framework for Fortanix EDP. https://github.com/ndokmai/rust-sgx-remote-attestation. Accessed: 2020-08-18.

[43] Code Sample: Intel® Software Guard Extensions Remote Attestation End-to-End Example. https://software.intel.com/content/www/us/en/develop/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example.html. Accessed: 2020-08-18.

[44] Intel® SGX Attestation Service Utilizing Enhanced Privacy ID (EPID) . https://api.portal.trustedservices.intel.com/EPID-attestation. Accessed: 2020-08-18.

[45] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*, pages 455–462. IEEE, 2004.

[46] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C Schmidt. Riot os: Towards an os for the internet of things. In *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*, pages 79–80. IEEE, 2013.

[47] Aleksandar Milinković, Stevan Milinković, and Ljubomir Lazić. Choosing the right rtos for iot platform. In *Proceedings of the international scientific professional symposium Infoteh, Jahorina*, pages 18–20, 2015.

[48] Mehamed Ahmed Abdurrahman, Gebremedhn Mehari Gebru, and Tsigabu Teame Bezabih. Sensor based automatic irrigation management system. *International Journal of Computer and Information Technology*, 4(3):532–535, 2015.

[49] Karan Kansara, Vishal Zaveri, Shreyans Shah, Sandip Delwadkar, and Kaushal Jani. Sensor based automated irrigation system with iot: A technical review. *International Journal of Computer Science and Information Technologies*, 6(6):5331–5333, 2015.

[50] Y Kim and RG Evans. Software design for wireless sensor-based site-specific irrigation. *Computers and Electronics in Agriculture*, 66(2):159–165, 2009.

[51] Yunseop Kim, Robert G Evans, and William M Iversen. Remote sensing and control of an irrigation system using a distributed wireless sensor network. *IEEE transactions on instrumentation and measurement*, 57(7):1379–1387, 2008.

[52] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. On the performance of intel sgx. In *2016 13th Web Information Systems and Applications Conference (WISA)*, pages 184–187. IEEE, 2016.

[53] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for intel sgx enclaves. In *Proceedings of the 19th International Middleware Conference*, pages 201–213, 2018.

[54] Anders T Gjerdrum, Robert Pettersen, Håvard D Johansen, and Dag Johansen. Performance of trusted computing in cloud infrastructures with intel sgx. In *CLOSER*, pages 668–675, 2017.

[55] Alexandru Madalin Ghenea. A security kernel for protected module architectures. Master's thesis, KU Leuven, 2017. https://distrinet.cs.kuleuven.be/software/sancus/publications/madalinghenea17thesis.pdf.

[56] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[57] Alexey Gribov, Dhinakaran Vinayagamurthy, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full sql query support. *arXiv preprint arXiv:1711.02279*, 2017.

[58] Pierre-Louis Aublin, Florian Kelbert, Dan O'keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. Talos: Secure and transparent tls termination inside sgx enclaves. *Imperial College London, Tech. Rep*, 5(2017), 2017.

[59] Rust MbedTLS: Rust implementation of TLS. https://crates.io/crates/mbedtls. Accessed: 2020-08-18.