Authentic Execution in Smart Farming

Second Thesis presentation

Student: Gianluca Scopelliti
Promotor: Frank Piessens
Supervisors: Jan Tobias Mühlberg, Fritz Alder

Recap

Recap: what is this thesis about?

- The main purpose of my thesis is to provide a secure implementation of a distributed, event-driven application, composed by modules developed for different architectures.

- In practice, i extend the concept of «Authentic Execution», providing new features to the current implementation, such as the support of the Intel SGX architecture, and the integration with Sancus.

- The work is then applied to Smart Farming, which is an interesting use-case where security is an huge concern.

Authentic Execution between SGX enclaves

Introduction

- The first step of my work was to implement an Authentic Execution framework for applications (enclaves) written using Intel SGX.
- The main idea was to keep the same philosophy and structure used for the Sancus implementation
  - The framework provides to the developer an abstraction where all the main functionalities are provided automatically.
  - The developer only needs to write the core logic of the module, defining also how the modules are connected each other, using a **descriptor file** and **annotations** in the code.
  - All the rest (i.e. the code for Enclaved Execution and Authentic Execution) is added at compile time.

Platform

- The applications are written in *Rust*
  - Efficient, modern programming language
  - It is a **safe** language: thanks to the powerful Rust compiler, many memory management vulnerabilities are prevented at compile time; for other ones, runtime controls are introduced (e.g. checking bounds of an array)
- *Fortanix EDP* is the framework used to write SGX applications
  - The platform provides a full abstraction over the SGX layer, where the developer can write his own application like a normal, native one

Input: from the developer to the framework

- The developer passes as input a folder, containing:
  - Description of the system:
    - The developer specifies a configuration file, containing information about the modules, nodes, connections and so on
  - Description of the single modules:
    - Each module is a separate Rust project; the developer only has to write the core logic of the module

```
example/
├── input.json
├── sm1
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
└── sm2
    ├── Cargo.toml
    └── src
        └── lib.rs
```

Defining the system

- The input JSON file contains a full description of the system
- Nodes
  - A subsystem that contains modules
  - Each node has an Event Manager, which handles the events to/from the node
- Modules
  - Each module belongs to a node
  - Except for Remote Attestation, a module directly communicates **only** with the EM
- Connections
  - A connection establishes a **trusted path** between the output of a module and the input of another

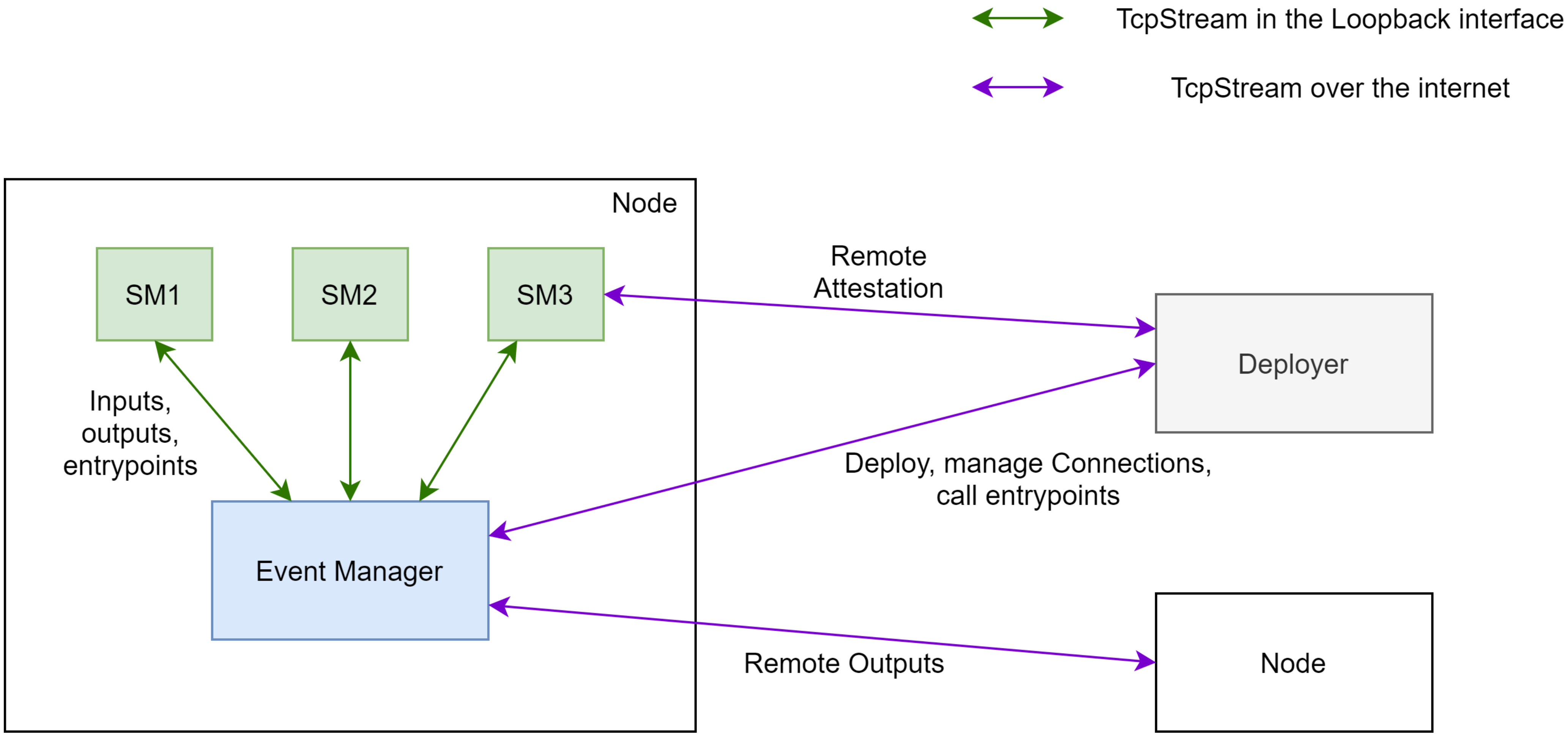

```json
{
  "nodes" : [
    {
      "name" : "node1",
      "ip" : "127.0.0.1",
      "em_port" : 5000
    },
    {
      "name" : "node2",
      "ip" : "127.0.0.1",
      "em_port" : 6000
    }
  ],

  "modules" : [
    {
      "name" : "button",
      "node" : "node1"
    },
    {
      "name" : "lcd",
      "node" : "node2"
    }
  ],

  "connections": [
    {
      "from_module": "button",
      "from_output": "button_pressed",
      "to_module": "lcd",
      "to_input": "show_value"
    }
  ]
}
```


KU LEUVEN

The complete scheme



Security concerns

- Only the SMs and the Deployer are considered **trusted**, whereas the event managers, the nodes and the communication network **are not.**

- To guarantee strong security properties in this scenario (in terms of Confidentiality, Integrity and Authenticity of the data), the same principles as described in the «Authentic Execution» paper have been implemented.

  - **Remote Attestation** ensures that a module is correctly loaded inside a node (and not tampered with). Moreover, a **Master Key** is established during the process, known only by the deployer and the module itself.

  - Each connection between modules is protected with a **Connection Key**, generated by the deployer and sent to the two modules involved (encrypted with the modules' Master Key).

Defining a Module

- The developer creates a Rust Cargo library and writes all the logic of the module, as well as defining inputs, outputs and entrypoints

- An external script takes the module as input and generate the missing code.

- This includes a main function and all the data structures and functions needed for Authentic Execution to work, as well as the code for Remote Attestation and Enclaved Execution

```rust
/* --- user-defined constants, imports, etc.. --- */

static VALUE : u32 = 42;

/* --- Inputs, Outputs, Entrypoints --- */

//@ sm_output(set_tap)

//@ sm_entry
pub fn say_hello(_data : &[u8]) -> Result<Vec<u8>, Vec<u8>> {
    authentic_execution::debug("ENTRYPOINT: say_hello");

    println!("Hello from {}!", *authentic_execution::MODULE_NAME);

    authentic_execution::success("Ok")
}

//@ sm_input
pub fn sensor_data_received(data : &[u8]) {
  authentic_execution::debug("INPUT: sensor_data_received");

  let enable = analyze_data(data);

  set_tap(&enable);
}


/* --- User-defined functions --- */

fn analyze_data(data : &[u8]) -> Vec<u8> {
  // do computation..
 }
```

Details

- The module, after performing Remote Attestation with the deployer, will listen for messages (events) sent by the Event Manager

- A message contains an Entrypoint ID and data. Based on the EID, the corresponding function is executed
  - EID 0: *set_key*
  - EID 1: *handle_input*
  - The other entrypoints are defined by the developer.

Future extensions

- Store the Master Key on disk using the *SGX **data sealing*** feature
  - Useful if the module crashes or it is stopped and runned again
  - Thanks to the SGX hardware, we have strong guarantees that only the module would be able to read its own sealed data

- N:N relationships between inputs and outputs
  - Current implementation: An output can **only** be connected to a single input and vice-versa
  - This is a limitation introduced for simplicity. However, multiple connections would be useful (e.g. a sensor output connected to both a «database» and a «computation» enclave).

Conclusions

- The framework provides a very easy way for developers to write distributed SGX applications
- The *Authentic Execution* environment provides a «trusted path» between modules, in terms of *confidentiality*, *integrity* and *authenticity* of the data
  - Since the EM, nodes and the network are not trusted, availability is out-of-scope (we cannot, for example, guarantee that the EM will deliver an event to a module)
- Unfortunately, the source of the path cannot be trusted
  - SGX does not provide support for secure I/O: in this scenario, the source can only be an entrypoint defined by the developer (but everyone can call module's entrypoints!)
- To take full advantages from this framework, SGX modules need to be connected to modules whose architecture can perform secure I/O (-> Sancus)

Next steps

Integration with Sancus

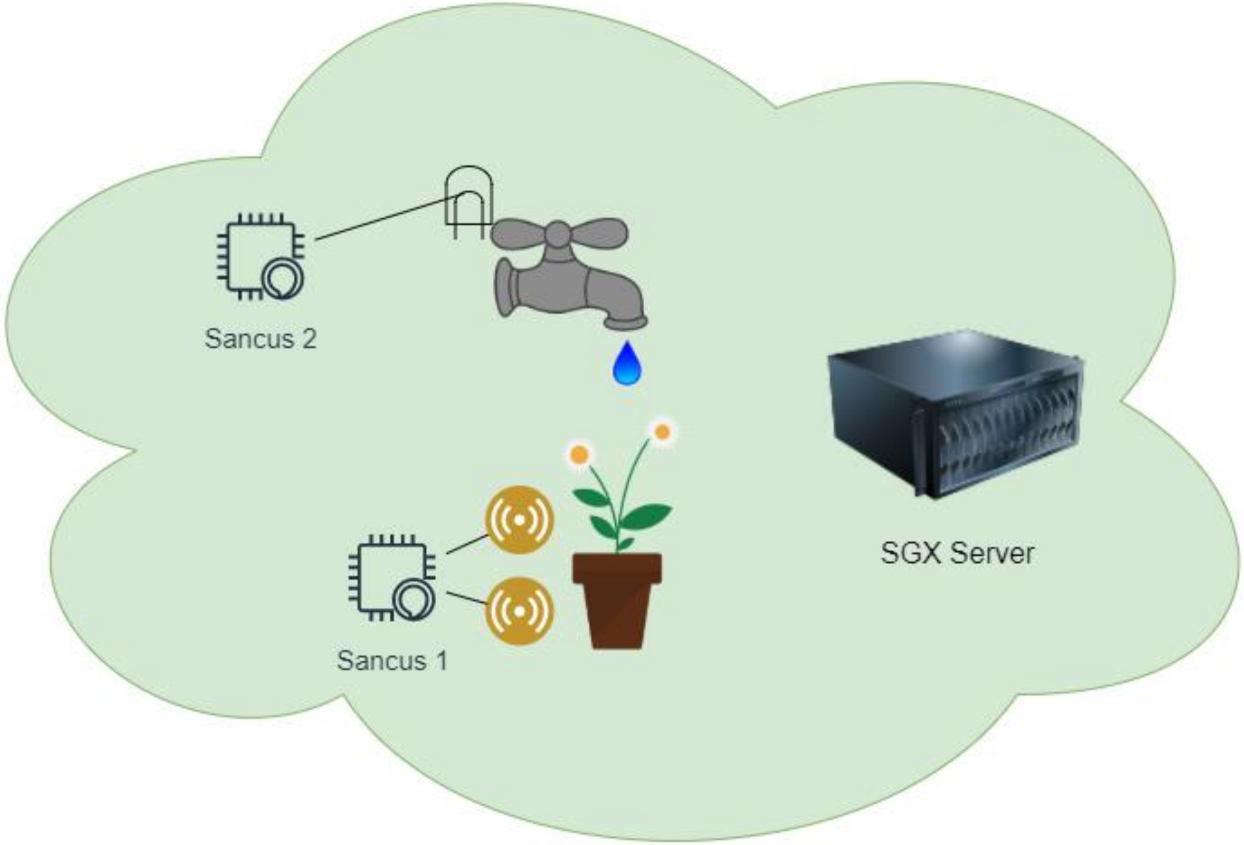
- Sancus is a Trusted Execution Environment for embedded devices; it also provides Secure I/O
  - Developing Sancus modules, we can establish a **full** trusted path from an input source to an output
  - Use case (applied to Smart Farming):
    - Sensor: collect data from the environment (Sancus) ->
    - Stats: compute statistics given the data collected (SGX) ->
    - Actuator: perform operations on the environment (Sancus)

- The code for creating an Authentic Execution environment for Sancus devices has already been implemented
  - The next step of my work is to «merge» the two frameworks into a single one, while at the same time leaving some space for the support of other architectures.

Prototype for Smart Farming

- In the first presentation i illustrated a prototype for a possible application of my work applied to the Smart Farming field.

- The prototype consisted of a simple application for automatic water supply of a flowerpot, where:
  - **An input node** (Sancus 1), retrieves data from a flowerpot using sensors
  - **A computation node** (SGX server) executes some logic given the data received and makes decisions
  - **An output node** (Sancus 2) receives commands from the SGX server to enable/disable the water tap



Other ideas

- **Availability concerns:** implement some «backup» logic to be executed when availability is not guaranteed (e.g. the network goes down)
- **General network API:** Communication between Event Managers in different nodes can also be performed using different mediums
- **End-user application:** a dashboard for the end-user (i.e. the farmer) used to monitor the system and send commands

Related work

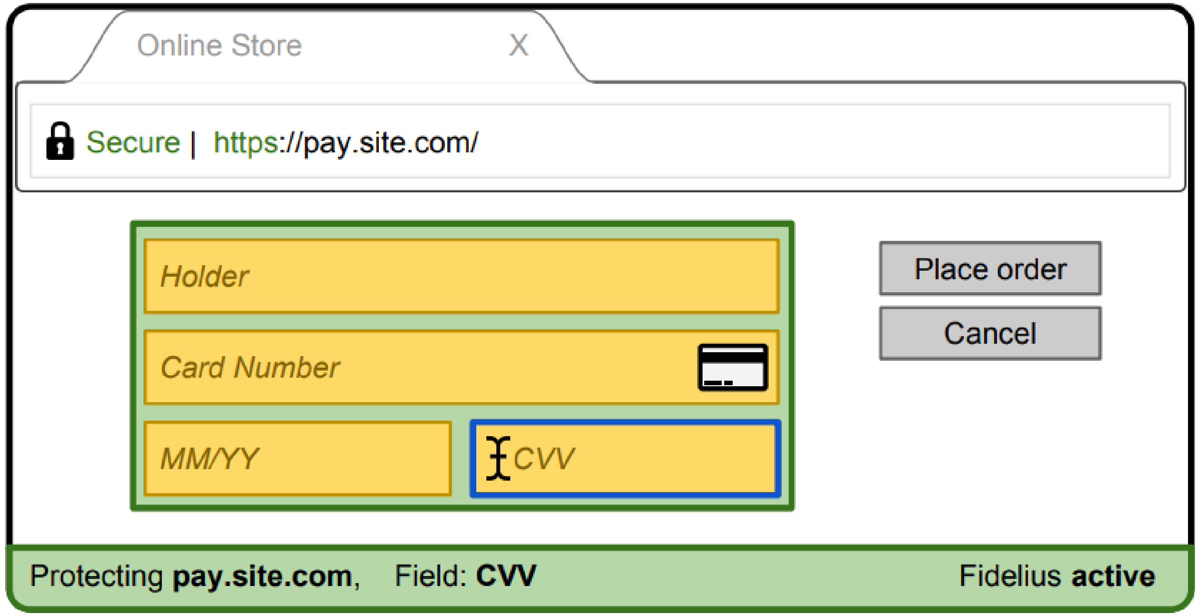
Fidelius: Protecting User Secrets from Compromised Browsers

- An interesting project developed by researchers from the Stanford University.

- Goal is to secure some sensitive form fields of a web page (e.g. credit card info), protecting the data inserted by the user from the keyboard to the remote server.

- The concept is to establish a «trusted path», where all the system is untrusted (OS, browser, etc..)

- As a comparison with our work, we can say that Fidelius' main concern is **confidentiality** of data. Our approach, instead, aims primarily on **integrity**.



Conclusions

Conclusions

- Over the past 6-8 weeks, i developed a framework for performing Authentic Execution between SGX enclaves
- Next steps: integrate SGX and Sancus enclaves
  - At the same time: improve code, provide new features
  - Demo for Smart Farming

References

- Authentic Execution
- Rust programming language
- Fortanix EDP
- Remote Attestation Rust code
- Fidelius