End-to-End Security for Distributed Event-driven Enclave Applications on Heterogeneous TEEs

GIANLUCA SCOPELLITI, Ericsson AB and KU Leuven
SEPIDEH POUYANRAD, JOB NOORMAN, and FRITZ ALDER, KU Leuven
CHRISTOPH BAUMANN, Ericsson AB
FRANK PIESSENS, KU Leuven
JAN TOBIAS MÜHLBERG, KU Leuven and Université Libre de Bruxelles

This article presents an approach to provide strong assurance of the secure execution of distributed event-driven applications on shared infrastructures, while relying on a small Trusted Computing Base. We build upon and extend security primitives provided by Trusted Execution Environments (TEEs) to guarantee authenticity and integrity properties of applications, and to secure control of input and output devices. More specifically, we guarantee that if an output is produced by the application, it was allowed to be produced by the application's source code based on an authentic trace of inputs.

We present an integrated open-source framework to develop, deploy, and use such applications across heterogeneous TEEs. Beyond authenticity and integrity, our framework optionally provides confidentiality and a notion of availability, and facilitates software development at a high level of abstraction over the platform-specific TEE layer. We support event-driven programming to develop distributed enclave applications in Rust and C for heterogeneous TEE, including Intel SGX, ARM TrustZone, and Sancus.

In this article we discuss the workings of our approach, the extensions we made to the Sancus processor, and the integration of our development model with commercial TEEs. Our evaluation of security and performance aspects show that TEEs, together with our programming model, form a basis for powerful security architectures for dependable systems in domains such as Industrial Control Systems and the Internet of Things, illustrating our framework's unique suitability for a broad range of use cases which combine cloud processing, mobile and edge devices, and lightweight sensing and actuation.

CCS Concepts: • **Computer systems organization** → **Sensors and actuators**; • **Security and privacy** → **Distributed systems security**; *Embedded systems security*; *Domain-specific security and privacy architectures;*

This research is partially funded by the Research Fund KU Leuven and by the Flemish Research Programme Cybersecurity. Specific funding was provided under the SAFETEE project by the Research Fund KU Leuven. This research has received funding under EU H2020 MSCA-ITN action 5GhOSTS, grant agreement no. 814035. Fritz Alder is supported by a grant of the Research Foundation – Flanders (FWO).

Authors' addresses: G. Scopelliti, Ericsson AB, Stockholm, Sweden, imec-DistriNet and KU Leuven, Leuven 3001, Belgium; email: gianluca.scopelliti@ericsson.com; S. Pouyanrad, J. Noorman, F. Alder, and F. Piessens, imec-DistriNet, KU Leuven, Leuven 3001, Belgium; emails: sepideh.pouyanrad@kuleuven.be, job@noorman.info, fritz.alder@acm.org, frank.piessens@kuleuven.be; C. Baumann, Ericsson AB, Stockholm, Sweden; email: christoph.baumann@ericsson.com; J. T. Mühlberg, imec-DistriNet, KU Leuven, Leuven 3001, Belgium, Ecole Polytechnique, BEAMS & Cybersecurity Research Center and Université Libre de Bruxelles, Brussels 1050, Belgium; email: jan.tobias.muehlberg@ulb.be.

2471-2566/2023/06-ART39 $15.00
https://doi.org/10.1145/3592607

Additional Key Words and Phrases: Trusted Execution, Software Development Kit, IoT security, distributed systems security

ACM Reference format:
Gianluca Scopelliti, Sepideh Pouyanrad, Job Noorman, Fritz Alder, Christoph Baumann, Frank Piessens, and Jan Tobias Mühlberg. 2023. End-to-End Security for Distributed Event-driven Enclave Applications on Heterogeneous TEEs. *ACM Trans. Priv. Sec.* 26, 3, Article 39 (June 2023), 46 pages.
https://doi.org/10.1145/3592607

1 INTRODUCTION

Trusted Execution Environments (**TEEs**) allow an application to execute in a hardware-protected environment, often called *enclave*. Enclaves are isolated and protected from the rest of the system, ensuring strong confidentiality and integrity guarantees. Cryptographic primitives and credential management, with keys that are unique per enclave and that can only be used by that enclave, enable secure communication and remote attestation; the latter is a mechanism to obtain cryptographic proof that an application is running under enclave protection on a specific processor. Several Trusted Execution Environments (TEEs) are available in industry and research. Open-source TEEs include Sancus and Keystone; proprietary options are, e.g., **Software Guard Extensions** (**SGX**) for Intel processors, **Secure Encrypted Virtualization** (**SEV**) for AMD, TrustZone for ARM, and others [41]. Developing applications that execute on heterogeneous TEEs is difficult, in particular for scenarios that combine Internet-of-Things, Edge, and cloud hardware: each TEE requires a platform-specific software implementation, comes with different approaches to key management and attestation, a different **Trusted Computing Base** (**TCB**) footprint, and provides different hardware features and security guarantees.

The development of a distributed application composed of multiple modules running on heterogeneous hardware is non-trivial by itself, but becomes an even bigger challenge when the application has stringent security requirements that demand the use of TEE architectures. A developer needs to make choices as to which security features are required for which components, adapt the code of each component to multiple specific platforms, arrange for different deployment and attestation strategies, and implement secure interaction between the components. Open-source projects such as Open Enclave SDK and Google Asylo aim at bridging the development gap between different TEEs. However, software engineers still need to account for the communication between different modules, which has to be properly secured with cryptographic operations for data encryption and authentication. In particular, the responsibility for deploying the distributed application, loading and attesting each enclave, establishing session keys, and securing connections between distributed components, is still left to the application developer and operator. Overall, ensuring strong security guarantees in distributed scenarios poses a challenge to the adoption of TEE technology.

This article studies the problem of securely executing distributed applications on such a shared, heterogeneous TEE-infrastructure, while relying on a small run-time Trusted Computing Base (TCB). We want to provide the owner of such an application with strong assurance that their application is executing securely. We focus on *(1) authenticity* and *integrity* properties of *(2) event-driven* distributed applications. For this selection of security property and class of applications, we specify the exact security guarantees offered by our approach. We believe our approach to be valuable for any kind of distributed application (event-driven or not). In particular, our prototype supports arbitrary C and Rust code.

We distinguish physical events, such as sensor inputs or actuation, from logical events that are generated and consumed by application components. Roughly speaking, our notion of *authentic*

execution is the following: 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. This notion of security is roughly equivalent to the concept of *robust safety* in later literature [2]. Our approach has been experimented with in previous work, where we secure smart distributed applications in the context of smart metering infrastructures [47], automotive computing [65], and precision agriculture [59], which we generalize and partly formalize under the name "authentic execution" in [51], and use as a training and tutorial scenario to explain attestation and secure communication with heterogeneous TEEs [48].

The main contributions of this article are

- We reflect on the design and implementation of an approach for authentic execution of event-driven programs on heterogeneous distributed systems, under the assumption that the execution infrastructure offers specific security primitives – standard enclaves [61] plus support for secure I/O [52] (Section 3); we comprehensively discuss corner cases and hurdles regarding the use of secure I/O in distributed enclave applications;
- We provide a revised open-source implementation of the approach for Intel SGX, ARM TrustZone and Sancus, which supports software development in Rust and C (Section 4); we elaborate on implementation challenges towards achieving comparable security guarantees in heterogeneous TEE deployments;
- We conduct an end-to-end evaluation of a concrete application scenario, considering the performance and security aspects of our framework (Section 5), that considers dynamic system updates, and showing that our approach allows for the deployment of complex distributed software systems with a very small run-time application TCB;
- We design and implement a light-weight symmetric attestation scheme for ARM TrustZone, inspired by and providing security guarantees comparable to Sancus.

Our complete implementation of the authentic execution framework and the evaluation use case are available at https://github.com/AuthenticExecution/main, a formalization and proof sketch of our security guarantees is also available there.

2 RUNNING EXAMPLE, INFRASTRUCTURE & OBJECTIVES

As a running example we discuss an automated irrigation system, as illustrated in Figure 1, which involves a series of light-weight sensors and actuators in the field that, e.g., monitor soil moisture and crop growth, and control water supply. The system can be connected to edge infrastructure or cloud services for centralized configuration and maintenance, to integrate reporting and billing, and to minimize water consumption based on weather predictions. Naturally, smart farming systems are security critical since malicious interactions can potentially lead to huge costs and may destroy a crop; they also feature a high level of dynamicity where equipment needs to be reconfigured for specific sensing and actuation tasks that depend on the type of crop [57] and, e.g., sustainability objectives [62], and demand a high level of dependability where events must be guaranteed to be processed in a timely manner.

Figure 2 (source code in Figure 3) zooms in on the light-weight in-field sensing and actuation on the left side in Figure 1 and details application modules and event flows in an agricultural sensor network with two soil moisture sensors. The infrastructure can be reused for multiple applications which can be provided by different stakeholders. Applications include visualizing soil moisture, targeted irrigation, or detection of flooding or leakage. We show two of these applications: one (A_{Flo}) that detects flooding or leakage and disconnects the water supply in case of emergency, and another (A_{Agg}) that aggregates and displays data on soil moisture.

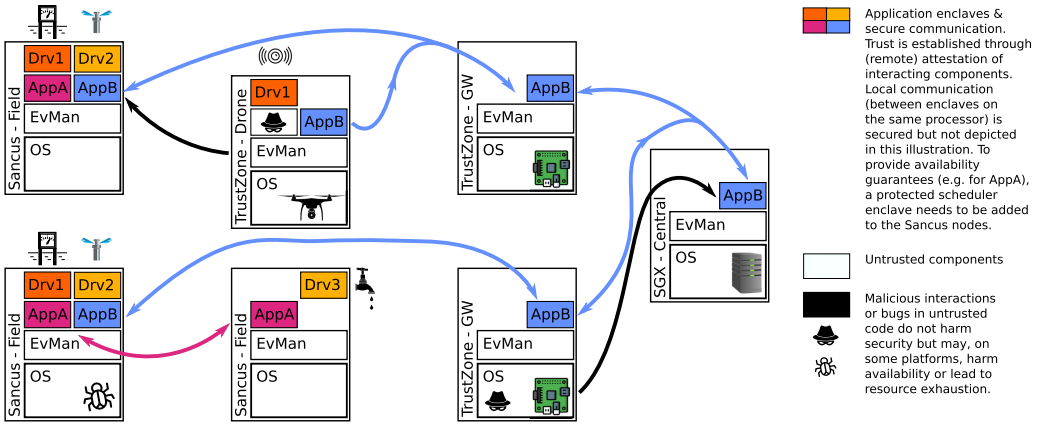


Fig. 1. A smart irrigation system as an example for distributed application networks we can support. Lightweight sensing and actuation nodes are deployed in a field. Application *AppA* controls irrigation units (through driver *Drv2*) and water supply (Drv3), based on soil moisture (obtained through *Drv1*). Application *AppB* provides similar functionality but has access to additional data sources, e.g., aerial surveillance and data aggregation on central infrastructure. All application components execute in enclaves (colored) and the actual composition and configuration of components can change dynamically. Directed data flows through untrusted networks (colored arrows) are at least authenticated and integrity protected; attestation precedes the establishment of all data flows, and mutual authentication is established between enclaves. All other software in the scenario is untrusted regarding our security properties, which leads to a very small run-time application TCB. Guaranteeing availability properties may require a different compartmentalization strategy. The concept is also applicable across, e.g., the different control units within a car or in an autonomous robot.

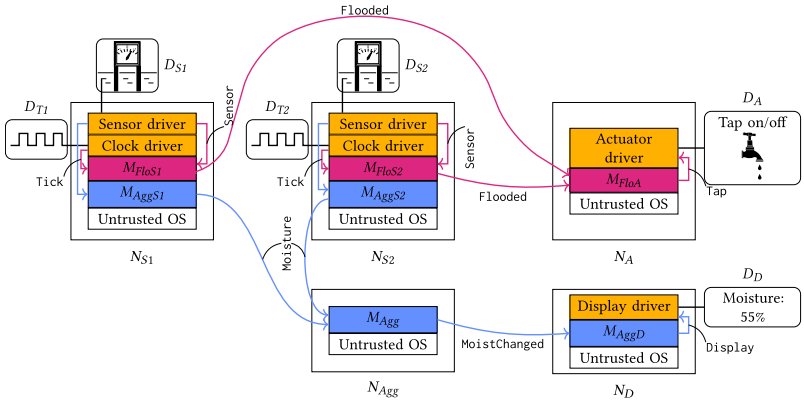


Fig. 2. Our running example with two applications, A_{Agg} (blue) and A_{Flo} (red). Hardware (N_* and D_*) is trusted; the OS and the network are untrusted. E.g., the A_{Flo} deployer creates the three red SMs (cf. Figure 3(a)) with a trusted compiler, attests the shared sensor-, clock-, and display drivers and sets-up connections between the SMs. Remote attestation assures authentic execution of A_{Agg} and A_{Flo}.

2.1 The Shared Infrastructure

The infrastructure is a collection of *nodes* (N_i), where each node consists of a processor, memory, and a number of *I/O devices* (D_i). Multiple mutually distrusting stakeholders share the infrastructure to execute distributed *applications* (A_i). For simplicity, we assume processors are simple

microprocessors, such as the OpenMSP430 used in our prototype, to explain the underlying concepts and security guarantees of our approach. As we detail in Section 4 and thereafter, our implementation does support the commercial TEEs TrustZone and Intel Software Guard Extensions (SGX), and we evaluate our approach on an integrated scenario that involves these TEEs.

An I/O device interfaces the processor with the physical world and facilitates *(1)* sensing physical quantities (e.g., the state of a switch), *(2)* influencing physical quantities (e.g., an LED), and *(3)* notifying the processor of state change (e.g., a key being pressed) by issuing an interrupt.

In our running example, there are 5 nodes. Two of these (N_{S1} and N_{S2}) are each attached to two input devices (a clock D_{Ti} and a soil moisture sensor D_{Si}), and are installed in a field, e.g., along a row of crops. Two other nodes (N_A and N_D) are connected to the actuator and display devices (D_A, D_D) to control the water supply and show application output. One node (N_{Agg}) is not connected to any I/O device but performs general-purpose computations, e.g., aggregate data from sensor nodes.

2.2 Modules & Applications

We use an event-driven application model and *modules* (M_i) contain input- and output channels. Upon reception of an event on an input channel, the corresponding event handler is executed atomically and new events on the module's output channels may be produced.

An application, then, is a collection of modules together with a *deployment descriptor*. This descriptor specifies on which nodes the modules should be installed as well as how the modules' channels should be connected. Channels can be connected in two ways. First, one module's output channel can be connected to another's input channel, behaving like a buffered queue of events. Second, the infrastructure can provide a number of *physical* I/O channels which can be connected to a module's I/O channels. The infrastructure must ensure that events on such channels correspond to physical events: An event received on a physical input could correspond to a button press or, similarly, an event produced on a physical output could turn on an LED. An important contribution of this article is a design to securely connect modules to physical I/O channels (Sections 3.4 and 4.1.5).

In our example applications (Figure 2), A_{Flo} consists of three modules: two (M_{FloS1} and M_{FloS2}) are deployed in a field and detect excess moisture (flooding or leakage) and one (M_{FloA}) actuates a central tap to disconnect water supply if needed. The two in-field modules have two inputs that are connected to input devices provided by the infrastructure: one that produces events for changes in soil moisture (D_{Si}) and another that sends periodical timer events (D_{Ti}). As the source code (Figure 3(a)) shows, these modules first wait for changes in critical changes in soil moisture, then wait for a maximum number of timer events, and finally produce an output event to indicate an emergency. These output events are connected to the inputs of M_{FloA} which in turn produces output events and sends them to the output actuator D_A. A second application A_{Agg} shares access to the sensor drivers with A_{Flo} to obtain sensor readings that are then aggregated and displayed.

2.3 Attacker Model

We consider powerful attackers that can manipulate all the software on the nodes. Attackers can deploy their own applications on the infrastructure, but they can also tamper with the OS. Attackers can also control the communication network that nodes use to communicate with each other. Attackers can sniff the network, modify traffic, or mount man-in-the-middle attacks. With respect to the cryptographic capabilities of the attacker, we follow the Dolev-Yao model [19].

Attacks against the hardware are out of scope. We assume the attacker to not have physical access to the nodes, neither can they physically tamper with I/O devices. We also do not consider side-channel attacks against our implementation. Physical protection and side-channel resistance are important but orthogonal and complementary to the protection offered by our approach.

```
module FloS1
on Sensor(reading):
  if reading >= SATURATED: flooded = 1
  else:
    flooded = 0
    count   = 0
    Flooded(false)
on Tick():
  if flooded: count = count + 1
  if count > MAX: Flooded(true)
module FloS2
# Similar to FloS1
module FloA
on Flooded1(true)
  f1 = true;
  if f2 = true: Tap(off)
on Flooded2(true):
  # Similar to Flooded1
```

(a) A_{Flo}

```
module AggS1
on Sensor(reading):
  Moisture(reading)
module AggS2
# Similar to AggS1

module Agg
on Moisture1(reading):
  l1.append(reading)
  del l1[:-100]
  MoistChanged(reading - avg)
  avg = sum(l1) / len(l1)
on Moisture2(reading):
  # Similar to Moisture1

module AggD
on MoistChanged(delta)
  Display(delta)
```

(b) A_{Agg}

Fig. 3. Pseudo-code of the applications from Figure 2. in a Python-like syntax. Enclaved application components are declared using the `module` keyword and span until the next `module` or the end of the file. The `on` statement starts an event handler which can connect to an output of another application component, or to a physical I/O channel. Outputs are implicitly declared when invoked through a function call-like syntax.

2.4 Security Objective

The deployer uses his own (trusted) computing infrastructure to compile the application A, to deploy the modules to the nodes in the shared infrastructure, and to configure connections between modules, and between modules and physical I/O channels. At run-time, a sequence of physical input events will happen, and the deployer can observe the sequence of physical output events that return. We say that this sequence of outputs is *authentic* for an application A if it is allowed by A's modules and deployment descriptor in response to the actual trace of input events: the source code of A explains the physical outputs on the basis of actual physical inputs that have happened. Section 3.5 will detail this sequence of physical input, processing, and lastly physical output further.

Our objective is to design a deployment algorithm such that the deployer can easily certify the authenticity of sequences. If the correctness of the deployment is verified, then our approach guarantees that any subsequent trace of output events observed by the deployer will be authentic. This security notion rules out a wide range of attacks, including attacks where event transmissions on the network are spoofed or reordered, and attacks where malicious software tampers with the execution of modules. Other relevant attacks are *not* covered by this security objective. We will explain more nuances regarding deployment in Section 3.5.

As discussed earlier, there are no general availability guarantees – e.g., the attacker can suppress network communication. However, TEE extensions such as [4] and the use of real-time operating systems can provide notions of availability that are relevant to maintain system safety in, e.g., autonomous control systems. There are also no strong confidentiality guarantees: Although this is not the focus of our design, our implementation *does* provide substantial protection of the confidentiality of application state as well as event payloads. Yet, the attacker may still observe the occurrence and timing of events in the system, and specifically on the sensing-and-actuation side of our systems, this information may very well reveal the state of the system. While it is technically possible to close these side channels, e.g., with artificially generated noise, this is infeasible for our light-weight TEEs and under constrained energy consumption (cf. Section 6.1 for details).

Many systems similar to our running example from Figure 1 exist, e.g., in the context of the Internet of Things, smart building, supply chain management, or intelligent transport systems. These may come with different requirements regarding security and privacy, potentially beyond the baseline guarantees of our framework. In Section 5.1 we look specifically at a smart-home scenario and discuss performance characteristics and security guarantees (cf. Section 6).

3 AUTHENTIC EXECUTION OF DISTRIBUTED APPLICATIONS

We outline our requirements for the infrastructure wrt. security features, and show how these features are used effectively to accomplish our security goals.

3.1 Underlying Architecture: TEEs

Given the shared nature of the infrastructure assumed in our system model, we require the ability to isolate source modules from other code running on a node. Since an important non-functional goal is to minimize the TCB, relying on a classical omnipotent kernel to provide isolation is ruled out. Therefore, we assume the underlying architecture is a TEE [61].

While details vary between TEEs, isolation of software modules is understood as follows: A module must be able to specify memory locations containing data that are accessible by the module's code only (*data isolation*). The code of a module must be immutable and a module must specify a number of *entry points* through which its code can be executed (*code isolation*). For simplicity, we further assume that both a module's code and data are located in contiguous memory areas called, respectively, its *code section* and its *data section*. We expect TEEs to implement a $W \oplus X$ policy so as to prevent code modification and code generation at runtime.

We also expect the availability of a compiler that correctly translates **Software Modules** (**SMs**) to the underlying architecture. The input to this compiler is as follows: *(1)* a list of entry point functions; *(2)* a list of non-entry functions; *(3)* a list of variables that should be allocated in the isolated data section; and *(4)* a list of constants that should be allocated in the isolated code section. The output of the compiler should be an SM suitable for isolation on the underlying architecture.

Besides isolation, we expect the TEE to provide a way to *attest* the correct isolation of an SM. Attestation provides proof that an SM with a certain identity has been isolated on the node, where the *identity* of an SM, usually based on a measurement of the deployed binary code, should give the deployer assurance that this SM will behave as the corresponding source code module.

After enabling isolation, the TEE should be capable of establishing a confidential, integrity protected, and authenticated communication channel between an SM and its deployer. Although the details of how this works may differ from one TEE to another, for simplicity, we assume the TEE establishes a shared secret between an SM and its deployer and provides an authenticated encryption primitive. We refer to this shared secret as the *module key*. The authentication property of the communication channel refers to an SM's identity and hence to attestation. Thus, the TEE ensures that if a deployer receives a message created with a module key, it can only have been created by the corresponding, correctly isolated, SM.

3.2 Mapping source modules to SMs

To map a source module to an SM, we use the following procedure. First, each of the source module's inputs and outputs is assigned a unique *I/O identifier*, and each of the connections between source modules is assigned a unique *connection identifier*. The format of these identifiers is unimportant, provided that they uniquely specify a particular input/output or a particular connection, respectively. By having a clear distinction between I/O identifiers and connection identifiers, many-to-many connections are supported, which means that: *(1)* a source module's output can be connected to multiple inputs of other source modules (e.g., a keyboard that sends key press events to both a key-logger and an LCD screen), and *(2)* a source module's input can be reached by multiple outputs of other source modules (e.g., a key-logger that records key presses of two different keyboards).

Second, a table (`ConnectionTable`) is added to the SM's variables that maps connection identifiers to a `Connection` data structure, such that every connection has one entry associated with it. These entries will be initialized to all zeros by the underlying architecture, which is interpreted as

```
def SetKey(payload):
    try:
      conn_id, io_id, key = Decrypt(payload)
      conn = Connection(conn_id, io_id, key)
      ConnectionTable[conn_id] = conn
    except: pass
```

Fig. 4. Pseudocode of the SetKey entry point. Note that Decrypt uses the module key to decrypt the payload and throws an exception if the operation failed (i.e., the payload's MAC is incorrect).

```
def HandleInput(conn_id, payload):
  try:
    conn = ConnectionTable[conn_id]
    if conn != 0:
      cb = CbTable[conn.io_id]
      key = conn.key
      nonce = conn.nonce
      cb(Decrypt(nonce, payload, key))
      conn.incrementNonce()
  except: pass
```

Fig. 5. Pseudocode of the HandleInput entry point. Erroneous accesses to the tables as well as errors during Decrypt cause exceptions. Thus, these events, as well as those for which no input key has been set, are ignored. Decrypt takes a key and the expected associated data as arguments.

an unestablished connection. For establishing a connection, an entry point is generated (SetKey). This entry point takes a connection identifier, an I/O identifier, and a key – encrypted using the module key – as input and updates the corresponding mapping in ConnectionTable (Figure 4). Since every connection needs to be protected from reordering and replay attacks, the Connection structure also includes a nonce.

Third, all the module's event handlers are marked as non-entry functions. A callback table (CbTable) is added to the SM's constants, mapping input identifiers to the corresponding event handlers. This table is used by the entry point HandleInput, which is called when an event is delivered to the SM. HandleInput takes two arguments: a plain-text connection identifier and an encrypted payload. If ConnectionTable has a connection for the given identifier, its key is used to decrypt the payload (using the *expected* nonce as associated data), which is then passed to the callback function stored in CbTable, retrieved using the I/O identifier stored in the Connection structure. If any of these operations fails, the event is ignored (Figure 5). From a programmer's perspective, an input callback will only be called for events generated by entities with access to valid connection keys.

Fourth, each call to an output is replaced by a call to a non-entry wrapper function HandleOutput. This function takes a connection identifier and a payload, encrypts the payload together with the current nonce using the corresponding connection key, then publishes the encrypted event (via HandleLocalEvent, cf. 3.3.2). If the output is unconnected, the output event is dropped (Figure 6).

Fifth, an entry point Attest is generated, which will be called by the deployer to attest the SM. This function takes a challenge as input and returns an *attestation evidence* as response. The attestation procedure and the content of the attestation evidence may vary from one TEE technology to the other; nonetheless, the input challenge is typically used as a freshness parameter to prevent replay attacks. On some TEEs, the attestation process also includes the generation of a shared secret between the SM and the deployer, which will be used as the module key (Figure 7).

To conclude, the following SM definition is given as input to the TEE compiler: *(1)* SetKey, Attest, and HandleInput as entry points; *(2)* input event handlers and HandleOutput as

```
def HandleOutput(conn_id, data):
  conn = ConnectionTable[conn_id]
  if conn != 0:
    nonce = conn.nonce
    key = conn.key
    payload = Encrypt(nonce, data, key)
    conn.incrementNonce()
    HandleLocalEvent(conn_id, payload)
```

Fig. 6. Pseudocode of the generated output wrapper. Since the compiler generates calls to this function that cannot be called from outside the module, the connection identifier is always valid and no error checking is necessary.

```
def Attest(challenge):
  try:
    ev = GenAttestationEvidence(challenge)
    return ev
  except:
    return None # attestation failed
```

Fig. 7. Pseudocode of the `Attest` entry point. Since the attestation of an SM is TEE-specific, we use a high-level function `GenAttestationEvidence` to retrieve the attestation evidence. The actual implementation of this function depends on the TEE used.

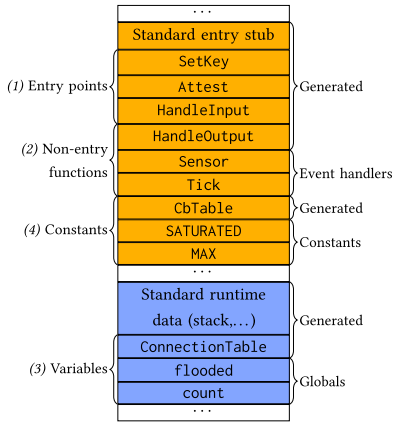


Fig. 8. Memory layout of the compiled version of module M_{FloS1} in application A_{Flo} (Figure 3(a)). The code- and data sections are shaded in yellow and blue, respectively. The numbers on the left labels correspond to the compiler inputs, while the right labels indicate whether parts are implicitly generated by the compiler or correspond to source code.

non-entry functions; *(3)* `ConnectionTable` as a global variable; and *(4)* `CbTable` as a constant. Figure 8 shows the compiled memory layout of one of the example modules.

3.3 Untrusted Software on the Nodes

To support the deployment of modules and the exchange of events between modules, untrusted (and unprotected) software components need to be installed on each node, as outlined below.

3.3.1 Module Loader. The module loader is an untrusted software component running on every node. The module loader provides services for external entities to interact with modules on a node. To this end, the module loader listens for two types of remote requests: `LoadModule` and `CallEntry`.

`LoadModule` takes a compiled SM as input, loads it into the TEE, and returns the module's unique identifier together with all information necessary for attestation and module key establishment. What exactly this information is and how the attestation and key establishment are performed is specific to the used TEE. `CallEntry` takes an SM's identifier, the identifier of an entry point, and potentially some arguments and calls the entry point with the given arguments.

3.3.2 Event Manager. The event manager is another untrusted software component running on every node that is used to route events from outputs to inputs. It recognizes three types of requests: `AddConnection`, `HandleLocalEvent`, and `HandleRemoteEvent`. A deployer can invoke `AddConnection` remotely to connect the output of a module to the input of another. How exactly inputs and outputs are identified is implementation specific but it will in some form involve specifying *(1)* a node address (e.g., an IP address); *(2)* an SM identifier; and *(3)* a connection identifier. As will become clear later, `AddConnection` only needs to be called on the event manager of the node where the output source module is deployed.

`HandleLocalEvent` is used by modules to publish an event; i.e., inside the output wrappers. The arguments are the module and connection identifiers and the event payload. Based on the identifiers, the event manager looks up the destination event manager and invokes its `HandleRemoteEvent` API, providing the identifiers of the input to which the request should be routed. For a `HandleRemoteEvent` request, the event manager will check if the destination module exists and, if so, invoke its `HandleInput` entry point, passing the connection identifier and payload as arguments.

3.3.3 Implementation. Although we introduced the module loader and the event manager as two separate software components, in our prototype we merged the functionalities of both in the same application, which we simply call the *Event Manager*. This design choice was made for the sake of simplicity: the actual implementation of such logical components is unimportant as long as each node provides the functionalities described above.

3.4 Physical Input and Output Channels

We assume that the infrastructure offers physical input and output channels using *protected driver modules* that translate application events into physical events and vice versa. Driver modules control these physical events – sensing and actuation – within the boundaries of the processor executing the driver. For input channels, these modules generate events that correspond to physical events and provide a way for application modules to authenticate the generated events. For output channels, a driver module (M_D) must have exclusive access to its I/O device (D) and allow an application module (M_A) to take exclusive access over the driver. That is, the driver will only accept events – and hence translate them to physical events – from the application module currently connected to it. Below, we put explicit requirements on the implementation of secure I/O and mark them for later reference.

First, we demand that **IO1**: *the infrastructure provider configures the physical I/O devices as expected*, i.e., the desired peripherals are connected to the right pins and thus mapped to the correct **Memory-Mapped I/O** (**MMIO**) addresses in the node. Since misconfiguration in the physical world cannot be detected by remote attestation, we need to require that I/O devices are set up correctly.

However, **IO2**: *the infrastructure must provide a way for the deployer of M_A to attest that it has exclusive access to the driver module M_D and that M_D also has exclusive access to its I/O device D.* The deployer must be able to attest M_D to ensure that it indeed only accepts events from the module currently having exclusive access and that it does not release this exclusive access without being asked to do so by the module itself.

We also need to require that, **IO3**: *after a microcontroller is turned on, only authenticated driver modules may take control of I/O devices. Once control is taken it is exclusive and never released.* This prevents attackers from taking direct control of output devices after a node resets. Driver modules are thus part of the trusted computing base and their module keys are only known to the infrastructure provider who authorizes exclusive connections to driver modules upon request from the deployer and keeps track of the "ownership" of the driver modules.

Finally, we demand that **IO4**: *output driver modules do not produce outputs unless requested by the application module, while input driver modules only generate outputs to application modules that correspond to physical inputs.*

To summarize, we have modeled physical *output* channels in such a way that:

(1) At startup, authenticated output driver modules take ownership over the (correct) output devices. Unauthorized access and connection attempts from other software are forbidden;
(2) Application modules can take ownership of an output driver module only if the latter is not already connected to other modules, and only if authorized by the infrastructure provider;
(3) After connecting to an output driver module, an application module retains exclusive access to the output device until explicitly released. Thanks to point (1), exclusive access is retained even across resets, although all connections need to be re-established.

For input modules, exclusive access is not strictly required. However, exclusive access to input devices can be configured in the same way as for outputs, if desired. These conditions lead to the following security properties:

— *Assume that for a time interval T the infrastructure provider only ever authorizes application module M_A to take control of device D via driver module M_D. If the deployer attests within T that M_A has taken exclusive access over output driver module M_D, then from that moment until the end of T every output from D can be explained alone by the code semantics of M_A and M_D in relation to the inputs received by M_A, i.e., no other module may cause outputs of D.*
— *If the deployer attests that M_A has taken access over an input driver module M_D, then from that moment on until the access is released any output from M_A can be explained alone by the code semantics of M_A and M_D in relation to the physical inputs to D and any other inputs received by M_A, i.e., an attacker cannot spoof inputs to D for M_A.*

In practice, I/O devices may produce initialization outputs when a driver module takes control of them after a reset. Handling these outputs is part of the specificities of a driver implementation, which we ignore here for simplicity.

3.5 Deployment

Deployment is the act of installing application modules on their nodes and setting up connections between outputs and inputs. As a requirement for deployment, **D1**: *the channel between deployer and infrastructure provider is assumed to be secure*, whereas communication with modules is performed over an untrusted network.

In phase 1, the deployer requests access to the driver modules connected to physical I/O devices from the infrastructure provider who provides these modules. For output devices, the deployer requests *exclusive* access, while for input devices the deployer can choose between exclusive or shared access. Before granting such access, **D2**: *the infrastructure provider needs to ensure the authenticity of the driver module controlling the I/O device*, e.g., via attestation. If, for any reason, the infrastructure provider cannot give access to some I/O devices (e.g., because an output device is already taken by someone else), the deployment is aborted. Otherwise, the deployer receives connection keys from the infrastructure provider (one for each I/O device), along with the guarantee that driver modules have correctly taken control of the physical I/O devices (Section 3.4).

Phase 2 consists of deploying the application on the infrastructure. First, the deployer starts by compiling each source module into a loadable image. Second, the deployment descriptor is used to find the node on which the module should be deployed and sends its module loader a `LoadModule` request. The deployer then performs the TEE-specific method of attestation calling the `Attest` entry point of each SM, and eventually setting up the module key. At the end of this step, the deployer has a secure communication channel with each of its deployed source modules. Finally, the deployer sets up the connections between modules, as well as the connections to the driver modules. Regarding the former, the deployer generates a unique connection key and sends it to both endpoints of the connection; for each endpoint, such key is encrypted using the module key and passed to the `SetKey` entry point using the `CallEntry` API. Concerning the latter, instead, connections are established via a similar interface, using the keys obtained in phase 1.

Security Discussion. During phase 1, the infrastructure provider plays an important role as a trusted party, providing exclusive access to driver modules. **D3**: *It is required that the provider does not leak the driver module keys and that exclusive access to a device is reserved for the deployer until they request to release it or a time T has passed (to which both the deployer and the provider have agreed upon).* In other words, no one else can acquire credentials to connect to the driver module during that time.

If a reset on a node N occurs, then all driver modules belonging to that node need to be re-initialized, and exclusive access re-configured (if previously set). Concerning the latter, **D4**: *the infrastructure must provide replay protection for messages exchanged in the protocol to establish exclusive driver access*, otherwise replay attacks might cause exclusive access to be unintentionally set up for modules that formerly had access, but should not have it anymore.

Concerning application modules instead, a reset on node N would cause all modules of that node to be re-deployed, re-attested, and connections for them to be re-established. Here, an attacker could technically mount a sophisticated replay attack to restore a previous configuration of an application; however, if replay protections for driver modules are correctly in place (as discussed above), the attacker would not be able to trigger any illicit or stale outputs.

Note that the strong requirements on physical output drivers do not apply for virtual outputs to software endpoints, such as secure databases or nodes belonging to a trusted party like the deployer. Such receivers can verify the authenticity and freshness of outputs via cryptographic means and reject replayed messages automatically, which is not possible for physical outputs to the real world.

3.6 Security Argument

Our goal is to ensure that all physical output events can be explained by the application's source code and the observed physical input events. More precisely: *Consider a time frame starting at the end of phase 1 of deployment (Section 3.5), and ending at a point where the deployer releases exclusive access to any of the output devices they control. Assume the deployer has observed a specific sequence of physical output events on one of these devices D_O in the considered time frame, then there have been contiguous sequences of physical input events on the input devices connected to the application such that the observed outputs follow from these inputs according to the application source code semantics.*

As an example, consider A_{Flo} (Figure 3(a)). If, after the application has been deployed, the water supply is turned off, then there must have been physical input events that caused the field to flood.

Output events can only be produced by the application's SMs; the assumption of a correct compiler and the successful attestation then lead to the desired property. Due to requirements **IO1**-**IO4**, *(1)* a physical output event can only be produced by the corresponding device (D_O); *(2)* output drivers have exclusive access to their device; and *(3)* an SM (M_O) has exclusive access to the driver;

thus only M_O can initiate physical outputs on D_O. Even if exclusive access is lost, e.g., due to a reset of the corresponding node triggered by an attacker, the infrastructure prevents that any other module can take control of D_O and produce rogue outputs, thanks to requirements **D1-D4**. The construction of SMs ensures that a module can only be invoked through its two entry points. Of these, only `HandleInput` can result in output events (Figures 4 and 5). Since `HandleInput` authenticates its input, output events are always the result of correct input events. Finally, as our deployment scheme only allows for two types of correct input events, physical input events and outputs from other modules, our security property follows.

3.7 Software Updates and Re-Deployment

In Section 3.5, we described a *static* deployment where the whole application is deployed at once; however, it is also possible to deploy application modules dynamically. Specifically in scenarios where the configuration of an application may change during its lifetime due to the reconfiguration, addition, or removal of sensors or processing nodes, or the deployment of additional applications on existing infrastructure, software updates become essential. In our smart irrigation system from Figure 1, this may, e.g., happen with changing environmental conditions or when plots are re-allocated for different crops.

Two common scenarios that may occur at runtime are software updates and re-deployments. The former consists of deploying an updated version of an SMs that, e.g., provides new functionality or fixes some bugs; the latter does not necessarily require code changes but may be needed when an SM crashes or needs to be migrated to another node. Our design for software updates and re-deployments consists of three steps:

(1) *Deployment of the new SM.* In the first step, the new SM is compiled, deployed on the infrastructure alongside the already running SMs, and finally attested. If any of these operations fails, the update is aborted. Note that the new SM will not process any events until connections are configured through its `SetKey` entry point (step 3).
(2) *Deactivation of the old SM.* In this step, the old SM is removed from the infrastructure. This is the point in time from which the application will start suffering from connectivity loss, until the new SM will be completely configured. Optionally, state may be transferred to the new SM before taking down the old one.
(3) *Re-establishment of connections.* Here, all the connections associated with the old SM must be re-configured in order to point to the new SM. This process involves calling the `SetKey` and `AddConnection` entry points on the involved SMs and event managers, as for a static deployment (Section 3.5); each connection retains its connection identifier, but keys are rotated for security reasons. Similarly, connections between the SM and I/O devices must be re-configured as well, if any (Section 3.4). At the end of this step, the update will be completed.

We evaluate the impact of software updates in Section 5.3.3. Note that actions 2 and 3 can also be executed in reverse order, if desired. Re-establishing the connections before disabling the old SM would result in a smaller availability disruption, but it may also cause an inconsistent state if events are generated during the update. For example, some events might be processed by the old SM and some other events by the new SM, causing undefined behavior. As our framework mainly focuses on integrity rather than availability, we should prevent such inconsistencies from occurring, for example by implementing a sort of synchronization mechanism between old and new SMs during the update. However, this would increase the complexity of our framework without providing substantial benefits compared to our original update strategy. Hence, we decided not to investigate this option further.

Transferring the state from the old to the new SM may be done in several ways. For instance, the *sealing* feature of certain TEEs (such as Intel SGX) allows an SM to securely store data persistently on disk, although it requires the new SM to be deployed on the same node as the old one. Our design already provides some basic support for state transfer by creating a *temporary connection* between the two SMs. In particular, a *transfer* output in the old SM may be connected to a *restore* input in the new SM, then the state migration can be triggered by calling a *save* entry point in the old SM that generates the *transfer* event. This would allow the state to be securely transferred between the two modules, even in different nodes. Of course, the deployer still needs to manually implement the *save* and *restore* functions in order to specify what data needs to be migrated. This protocol, however, requires the old SM to be up and running at the time of the update, and therefore cannot be applied if the module was disabled before, e.g., due to a crash or a reset on the node.

4 IMPLEMENTATION

Originally [51], we created a fully functional prototype of our design based on the hardware-only embedded TEE Sancus [50]. In later work [59], we extended our support to the commercial Intel SGX [46], a TEE for high-end x86 processors. Lately, we added support for a third TEE, based on ARM devices equipped with TrustZone technology [5].

As the implementations for all TEEs are compatible with each other, we can support heterogeneous deployments in which an SM can send protected events to any other SMs, regardless of the actual TEE technologies used. This enables a wide variety of use cases that combine sensing and actuation with clod and edge processing. As such, our running example in precision agriculture is scalable from small local applications such as an automated irrigation system to more complex deployments that feature, e.g., a wide range of sensors and actuators, and that involve complex data aggregation and processing. For example, we could have an authentic trace that starts from a Sancus sensor that leverages Secure I/O to collect measurements, then continues on a TrustZone gateway that forwards the measurements to an Intel SGX aggregator in the cloud, and finally ends with a command to actuate an output I/O device, sent back to a Sancus output driver through the same TrustZone gateway.

This chapter is structured as follows: Sections 4.1, 4.2, and 4.3 describe each of these three implementations separately, emphasizing on the features of their underlying TEEs. For each implementation we describe (1) the underlying security architecture, including data and code isolation as well as the root of trust, (2) the process of compiling applications into security modules and the processing of inputs and outputs, (3) the untrusted runtime, in particular the realization of the *LoadModule* and *CallEntry* requests, (4) the attestation protocol including the generation of attestation evidence and the establishment of module keys, (5) the configuration of secure communication endpoints, and (6) where available, the realization of secure I/O channels, protected driver modules, and relevant implementation requirements (**IO1**-**D4**). The main differences of the three implementations are then summarized in Section 4.4. Next, Section 4.5 illustrates the deployment phase. Finally, Section 4.6 introduces an optional component used to facilitate attestation and key management.

4.1 Sancus

Sancus [50] is an OpenMSP430-based TEEs designed for low-cost and low-power embedded applications. As described in Section 3.1, Sancus divides SMs in two sections, the (public) code section and the (private) data section, enforcing strict access rules for both. An SM's code section can only be entered through a *single* entry point: its first instruction. The compiler assigns each user-defined entry point an integer identifier and adds an entry stub that evaluates such an identifier, dispatching to the correct entry point. Besides, private data is only accessible by the module's code.

```
SM_OUTPUT(FloS1, Flooded);
SM_DATA(FloS1) int flooded, count;

SM_INPUT(FloS1, Sensor, data, len) {
  if (len > 0) {
    if (data[0] >= SATURATED) {
      flooded = 1;
    } else {
      flooded = count = 0;
      Flooded(&flooded, sizeof(flooded));
    }
  }
}
SM_INPUT(FloS1, Tick, data, len) {
  if (flooded && ++count > MAX) {
    Flooded(&flooded, sizeof(flooded));
  }
}
```

Fig. 9. A translation of module M_{FloS1} (Figures 2 and 3(a)) to C using the annotations understood by our compiler.

4.1.1 Enclave Development. The Sancus toolchain comes with a C compiler, libraries, and other tools that are needed to develop and build applications composed of one or more Sancus enclaves. Developers can define enclaves by annotating code and data using special macros defined in the Sancus trusted library: functions can be marked as either entry points or internal functions, using the `SM_ENTRY` and `SM_FUNC` macros respectively. Besides, variables annotated with `SM_DATA` will be allocated in the private data section, making it accessible only from inside the enclave.

Furthermore, Sancus provides an untrusted library to initialize enclaves and call their entry points. Pointers passed to an entry point must be properly sanitized to ensure that they do not point to enclave memory. Return values, instead, are copied from trusted to untrusted memory. The untrusted library also provides a function to dynamically load new enclaves at runtime; in our framework, this functionality is used to deploy application SMs.

At compile time, the Sancus compiler builds the application into a single binary that includes both untrusted runtime (e.g., the `main` function) and static enclaves. Such binary can then be loaded into a Sancus microcontroller using the dedicated loader. Besides, the Sancus toolchain also provides a tool to retrieve the module key of an enclave given a binary file, which is essential in order to attest the enclave from outside the microcontroller (Section 4.1.4).

4.1.2 APIs. Our implementation is a literal translation of the design outlined in Section 3.2. All modifications to the Sancus compiler are *extensions*, and all original Sancus features are still available to programmers (e.g., calling external functions or other SMs). On top of the existing annotations, we added two new ones: `SM_INPUT` and `SM_OUTPUT` for specifying inputs and outputs. Figure 9 shows an example module written in C using our annotations.

`SM_OUTPUT` expects a name as argument (more specifically, a valid C identifier). For every output, the compiler generates a function with the following signature: `uint16_t name(char* data, size_t len)`. This function can be called to produce an output event. For input handlers, `SM_INPUT` generates a function with the same signature as above. In this function, the programmer has access to a buffer containing the (unwrapped) payload of the event that caused its execution. For both inputs and outputs, the names provided in annotations are used in the deployment descriptor.

4.1.3 Untrusted Runtime. The untrusted runtime consists of a single application, the event manager, which implements all the untrusted API requests described in Section 3.3. The ELF binary is compiled together with a ported version of RIOT OS [7], an operating system specifically designed for devices with limited resources such as our OpenMSP430-based Sancus.

Our event manager was initially developed for an experimental version of Sancus RIOT [4]. Future work will adapt the event manager to the latest Sancus hardware and RIOT, in order to fully leverage common OS features as well as the protected scheduler, a fundamental component for providing availability guarantees to Sancus enclaves, as discussed in Section 6.2.

4.1.4 Attestation and Secure Channels. Sancus uses a three-level key hierarchy for remote attestation and secure communication. Every node contains a *node key*, which is only known by the node's owner, the *infrastructure provider*. Every vendor who is to install SMs on a particular node is assigned a unique identifier. The second level of keys, *vendor keys*, is derived from the node key and these vendor identifiers. Finally, *module keys* are derived from a vendor key using an SMs *module identity*. This module identity – the concatenation of the contents of the module's code section and the load addresses of both its sections – is used for authentication and attestation. The module key is calculated by the Sancus hardware when a module is loaded and can also be calculated by the module's vendor. Since the hardware ensures that module keys can only be accessed by the corresponding SM, it is guaranteed that the *use* of a certain module key (e.g., by creating a **Message Authentication Code** (**MAC**)) implicitly attests the module's identity.

The `Attest` entry point (Section 3.2) leverages a challenge-response protocol to verify that a Sancus SM is up and running on a specific node at a certain time. The challenge consists of a random sequence of bytes generated by the deployer, long enough to prevent replay attacks. The Sancus SM computes a Message Authentication Code (MAC) over the challenge using its module key, and sends the result back to the deployer. The latter performs the same operation: attestation can be considered successful if the result matches the response provided by the SM.

Sancus' crypto engine uses SpongeWrap [9] as encryption algorithm with spongent [10] as the underlying hash function to calculate MACs. The interface to the crypto engine is provided by two instructions: `Encrypt` takes a plaintext buffer, associated data (which will be authenticated but not encrypted), and a key and produces the ciphertext and an *authentication tag* (i.e., a MAC); `Decrypt`, given the ciphertext, associated data, tag and key, produces the original plaintext or raises an error if the tag is invalid. For both instructions, the key is an optional argument with the calling module's key as a default value. As with the original version of Sancus, this is the *only* way for a module to use its key.

4.1.5 Secure I/O. This section describes how protected drivers can be implemented using Sancus. Remember that, for output channels, we want an application module to have exclusive access to a driver (Section 3.4). This, in turn, implies that the driver should have exclusive access to the physical I/O device. Although for input channels the requirements are less strict – we only need to authenticate a device – for simplicity, we also use exclusive device access here.

Exclusive Access to Device Registers. Sancus, being based on the OpenMSP430 architecture, uses MMIO to communicate with devices. Thus, providing exclusive access to device registers is supported out of the box by mapping the driver's private section over the device's Memory-Mapped I/O (MMIO) region. There is one difficulty, however, caused by the private section of Sancus modules being contiguous and the OpenMSP430 having a fixed MMIO region (i.e., the addresses used for MMIO cannot be remapped). Thus, a Sancus module can use its private section either for MMIO or for data but not for both. Therefore, a module using MMIO cannot use *any* memory, including a stack, severely limiting the functionality this module can implement.

We decided to solve this in software: Driver modules can be split into two modules, one performing only MMIO (`mod-mmio`) and one using the API provided by the former module to implement the driver logic (`mod-driver`). The task of `mod-mmio` is straightforward: for each available MMIO location it implements entry points for reading and writing this location, and ignores calls by

modules other than `mod-driver`. This task is simple enough to be implemented using only registers for data storage, negating the need for an extra data section.

This technique lets us implement exclusive access to device registers on Sancus without changing the hardware representation of modules. Yet, it incurs a non-negligible performance impact because `mod-mmio` has to attest `mod-driver` on *every* call to one of its entry points. Doing the attestation once and only checking the module identifier on subsequent calls is not applicable because it requires memory for storing the identifier. We address this by hard-coding the *expected* identifier of `mod-driver` in the code section of `mod-mmio`. During initialization, `mod-driver` checks if it is assigned the expected identifier and otherwise aborts. `mod-driver` also attests `mod-mmio`, verifying module integrity and exclusive access to the device's MMIO registers. On failure, `mod-driver` aborts as well. The expected identifier of `mod-driver` can be easily deduced as Sancus assigns identifiers in order: the first loaded module takes identifier 1, the second 2, and so forth.

Sancus did not support caller authentication [50], which we require for `mod-mmio` to ensure invocation by `mod-driver` only. We added this feature by storing the identifier of the *previously* executing module in a new register, and added instructions to read and verify this SM identity.

Secure Interrupts. On the OpenMSP430, interrupt handlers are registered by writing their address to the interrupt vector, a specific memory location. Thus, handling interrupts inside SMs is done by registering a module's entry point as an interrupt handler. If the SM also supports "normal" entry points, a way to detect whether the entry point is called in response to an interrupt is required.

More generally, we need a way to identify *which* interrupt caused an interrupt handler to be executed. Otherwise, an attacker might be able to inject events into an application by spoofing calls to an interrupt handler. To this end, we extended the technique used for caller authentication. When an interrupt occurs, the processor stores a special value specific to that interrupt in the new register to keep track of the previously executing module. This way, an interrupt handler can identify by which interrupt it was called in the same way modules can identify which module called one of their entry points. The processor ensures that these special values used to identify interrupts are never assigned to any SM.

Interfacing with Applications. Our implementation follows the design in Section 3.4 to connect an application SM to a driver SM via connection keys. As we assume that driver modules are provided by the infrastructure, a deployer needs to interact with the infrastructure provider for the distribution of connection keys to the drivers. In fact, these keys need to be encrypted and authenticated using an SM's module key which, for driver modules, are only known by the infrastructure provider.

In our implementation, driver and MMIO modules are static enclaves embedded in the application binary loaded in the Sancus microcontroller (Section 4.1.1). At startup, these modules are immediately initialized to gain exclusive access to their I/O devices (**IO3**). To change this behavior, an attacker would require physical access to the microcontroller in order to upload a malicious binary; however, this is out of scope (Section 2.3).

The infrastructure provider can then keep track of which deployer has exclusive access to a driver SM at a certain time, denying requests coming from other deployers for the whole duration of the exclusive access period, or until the deployer explicitly gives up access (**D3**).

In our case, giving exclusive access means encrypting and authenticating the connection key (provided by the deployer) with the driver's module key, sending the result data back to the deployer; the communication between deployer and infrastructure provider should be done via a secure channel. The deployer would then be able to gain exclusive access to the driver SM by calling its `SetKey` entry point (or equivalent) using the encrypted payload as argument. Regarding

input drivers, the infrastructure provider may allow non-exclusive access by simply accepting requests from multiple deployers at the same time; if needed, a deployer may ask for exclusive access by adding an additional bit of information to the request.

To prevent replay attacks, the payload passed to the `SetKey` entry point contains a unique nonce that the driver SM can verify against a reference stored in memory (**D4**). This nonce must be rotated every time exclusive access is reconfigured.

In summary, our design for Secure I/O in Sancus consists of the following protocol:

(1) The deployer sends an initial request to the driver SM asking for its current nonce. The driver retrieves the nonce from secure memory and sends it back to the deployer.
(2) The deployer then sends the nonce and the connection key to the infrastructure provider via a secure channel (**D1**). Due to the static driver setup in Sancus, the provider knows that only authentic drivers have control of I/O devices (**D2**). The provider only ensures that the driver SM is not already taken by someone else and, if so, may grant the deployer exclusive access. To this end, the provider encrypts and authenticates the input data with the driver's module key, returning it to the deployer.
(3) The deployer then forwards the encrypted payload to the driver SM, which decrypts and authenticates it. If the nonce matches with the one stored in memory, the driver establishes exclusive access by storing the connection key into memory and using said key to decrypt future payloads. Finally, the driver sends an authenticated confirmation to the deployer using the same connection key, then generates a new nonce for the next run of the protocol.

Since only authentic drivers can decrypt the data using their module keys and complete the protocol illustrated above, the reception of the confirmation message allows the deployer to conclude that they indeed have acquired exclusive access to the desired I/O device (**IO2**). If the protocol does not complete for any reason (e.g., an attacker blocks one of the messages between driver and deployer), the deployer would never receive a confirmation from the driver SM; as such, the deployer would not trust outputs generated by that driver.

In case of node resets, the deployer would not lose exclusive access to the driver because the infrastructure provider would not complete step (2) of the protocol with another deployer (**D3**). However, the original deployer would have to run the protocol again to restore connectivity, rotating connection keys to prevent replay attacks.

An important aspect to consider is that monotonic counters as nonces are not enough to prevent replay attacks, because in case of resets on the node, the driver module would lose information about the current nonce. This problem can be solved by either *(1)* storing the current nonce in secure persistent storage, or *(2)* generating a random nonce each time. As Sancus currently provides neither persistent storage nor an RNG, we leave this issue to future work.

4.2 Intel SGX

Intel SGX is a TEE included in commercial Intel processors, consisting of hardware primitives and a set of instructions that can be used to isolate the code and data of an application in protected memory regions called *enclaves*. Access to enclave memory is restricted at runtime and only accessible by code within the enclave, and the enclave can only be entered through specific entry points. Thus, neither the host OS nor other software can access enclaves' code and memory, which results in a reduced TCB.

The architecture of Intel SGX provides an enclave with an enclave measurement called MRENCLAVE. This reflects the *enclave identity* and consists of a SHA-256 hash over the content of the enclave's code and data at initialization time. This enclave identity is complemented by the *author*

```
/* global variables definition omitted for brevity */

//@ sm_output(Flooded)

//@ sm_input
pub fn Sensor(data : &[u8]) {
  if data.len() > 0 {
    if data[0] >= SATURATED {
      flooded = true;
    } else {
      flooded = false;
      Flooded(&[0]);
    }
  }
}
//@ sm_input
pub fn Tick(_data : &[u8]) {
  if flooded {
    count += 1;
    if count > MAX {
      Flooded(&[1]);
    }
  }
}
```

Fig. 10. A translation of module M_{FloS1} (Figures 2 and 3(a)) to Rust using the annotations understood by our Intel SGX parser.

identity, called the MRSIGNER, which reflects the hash of the enclave's author's public key. This author is the entity who signs the enclave before distribution.

These two identities are then used as reference values for both local and remote attestation, as well as to generate *sealing keys* that allow the enclave to securely store persistent data on a disk.

4.2.1 Enclave Development. To develop an Intel SGX application, Intel provides its own SDK written in C/C++. Here, the application is partitioned into two sections: the untrusted code and the enclave. Communication between trusted and untrusted applications is made through a specific interface, defined by the developer using a C-like syntax called **Enclave Definition Language** (**EDL**). Calls from the untrusted application to the enclave are made via well-defined entry points named ECALLs, while untrusted functions are made available to the enclave via OCALLs.

Recently, new frameworks have been introduced that allow developers to write enclaves in modern languages and with reduced effort. In our framework, we leverage Fortanix **Enclave Development Platform** (**EDP**) [22], an SDK written entirely in the memory-safe Rust. Enclave Development Platform (EDP) abstracts the Intel SGX layer away from a developer, who can write an enclave similar to a native application; the necessary bindings are automatically added at compile time. EDP is seamlessly integrated with the Rust standard library, although some functionalities are not available for security reasons.

4.2.2 APIs. Our implementation of the design illustrated in Section 3.2 aims at reducing the development effort as much as possible. As described above, Fortanix EDP already gives huge benefits to a developer by abstracting the Intel SGX layer away, taking care of secure argument passing, data and code isolation, and so on. In addition, our framework provides a simple mechanism to declare outputs and inputs by using *code annotations*, which are parsed by the framework at deployment time in order to retrieve information about the module's endpoints and to inject required code. These annotations are analogous to the Sancus implementation described in Section 4.1.2, although we rely on special single-line comments instead of using macros (Figure 10).

Output and input events are designed to be *asynchronous*: after an output event is generated, the SM resumes its execution immediately, and no return values are expected. Asynchronous events

work well if they are generated from physical events (e.g., a button press), as the only purpose of the associated SM is to notify one or more other SMs to which it has active connections. However, in some other cases, *synchronous* events might be necessary, for example for querying a database. Here, a return value is expected and the SM that generated the output event might want to block the execution until the value is received. To address this need, the Intel SGX implementation comes with two additional annotations: `//@ sm_request` and `//@ sm_handler`. Request-handler events are similar to output-input events, however a request blocks the execution until the connected handler (if exists) returns a value.

4.2.3 Untrusted Runtime. The untrusted runtime consists of an untrusted Linux process that implements the logic illustrated in Section 3.3. Concerning the execution of new SMs, Fortanix EDP provides a default runner for Intel SGX enclaves called `ftxsgx-runner`, responsible for loading and executing an enclave. Thus, when a new `LoadModule` request arrives, the event manager spawns a new process using the Rust's `std::process::Command` API, executing `ftxsgx-runner` and passing as an argument the enclave's binary and signature. To realize `CallEntry` requests, our SGX compiler adds a simple TCP server as a front-end to the SM. At compile time, each SM is assigned a free port in the same network namespace as the event manager, so that they can exchange events with each other via *localhost*.

4.2.4 Remote Attestation and Secure Channels. In short, the remote attestation of an Intel SGX enclave consists of the following steps: *(1)* a remote entity (the *challenger*) sends an attestation request to the enclave. The request should contain a challenge that will be included in the attestation evidence to provide freshness; *(2)* the enclave (the *prover*) generates an attestation evidence, also called *Quote*, that is returned to the challenger. This Quote includes information about the identity of the enclave and the platform on which the enclave is running (hardware information, microcode version, etc.); *(3)* the challenger verifies the quote by first ensuring its authenticity and second verifying that the enclave and hardware are in the expected state.

A quote is generated by a dedicated enclave called **Quoting Enclave** (**QE**), which resides on the same platform as the enclave to be attested. The quote is protected using cryptographic keys unique to that particular platform. A trusted third party is then responsible for the quote verification, to ensure that *(1)* the quote is authentic, and *(2)* the hardware and firmware of the platform are up to date. After that, the challenger verifies the identity of the enclave, ensures that the quote is fresh (i.e., the challenge is included in the quote), and finally decides whether to trust or not the enclave.

Our framework supports the **Enhanced Privacy ID** (**EPID**) attestation scheme[1] [29], for which a Rust implementation already existed.[2] The scheme also involves a mutually-authenticated Diffie-Hellman key exchange between challenger and enclave; we leverage this protocol to establish a shared secret between a module and its deployer, which will be used as the *module key*.

4.2.5 Secure I/O. Unfortunately, SGX does not support secure I/O channels out of the box. In fact, it is by design impossible to map DMA devices into enclave memory [14]. While academic solutions have been proposed to overcome these issues (Section 7.3), they usually require additional trusted hardware [72] and software [37, 55, 69] to establish tamper-proof, exclusive channels to I/O devices. Hence, our framework does not support physical I/O channels for SGX-based SMs. Nevertheless, in the IoT settings we envision, SGX-based modules will likely be deployed in edge or cloud computing platforms where physical inputs and outputs are less relevant. Instead, such

[1]We implemented EPID for simplicity, though other attestation schemes could be supported as well. From a security perspective, all these schemes would provide similar guarantees.
[2]https://github.com/ndokmai/rust-sgx-remote-attestation.

modules may be used for monitoring and control of a number of IoT gateways or devices. As such, they would be connected to virtual endpoints in the deployer's trusted back-end system using credentials that are securely stored in enclave memory.

4.3 ARM TrustZone

TrustZone is a very common TEE implemented in the different flavors of recent ARM processors, which applies strong system-level isolation by separating both system hardware and software resources into two domains, namely the Normal World, and the Secure World, protecting the code and data in the secure world from being directly accessed or modified by the normal world [56].

To perform the secure context switching between worlds, ARM Cortex-A processors introduced a new CPU mode called monitor mode, which runs at the highest execution level of the secure world. The monitor mode can be entered via an interrupt, external abort, or explicit call of a special privileged instruction: *Secure Monitor Call*. Then, the value of the least significant bit of the Secure Configuration Register, known as the Non-Secure bit is changed and propagated through the AXI system bus to the memory and peripherals to preserve the process security state. On ARM Cortex-M processors, hardware is responsible for the transition between worlds, which optimizes switching overhead [49, 56]. In this article, we only focused on the features of TrustZone for Cortex-A processors.

The normal world runs a general-purpose operating system, such as Linux or Android, and untrusted (client) applications, while a lightweight TEE operating system and **trusted applications** (**TAs**) run in the secure world. In this article, we rely on **Open Portable Trusted Execution Environment** (**OP-TEE**) [38], which implements a secure operating system (the OP-TEE OS), a secure monitor, a non-secure user-space library called OP-TEE client exposed to the client applications, and a set of build toolchains to facilitate the development of TAs.

TrustZone relies on a secure boot process to prevent a system from being attacked during the booting process and enforce a strong system integrity policy. Secure boot forms a Chain of Trust by leveraging a signature scheme based on RSA to verify system images before their execution [41]. Concretely, the boot process operates as a sequence of stages, each former boot stage loads and checks the signatures of the next follow-on stage, usually initiated from the **root of trust** (**RoT**). RoT is usually a public key that is stored in the trusted root-key storage registers and cannot be modified [6].

4.3.1 TA Development. To develop a TA, OP-TEE provides its own SDK to both build and sign the TA binary with a private RSA key, then the signature is verified by OP-TEE OS upon loading of the TA to check its integrity. A TA must implement a couple of mandatory C-like syntax entry points to allocate some session resources and execute the target services. Communication with a TA is established by initiating a request from the client application through some defined entry points in the OP-TEE client to load the intended TA binary and its libraries into the secure memory. To identify a TA to be used, the client provides a unique 16-byte value called UUID that is generated by the developer. Then, the desired services in the TA are invoked from the client application by passing a so-called command ID which is also defined at compile time.

4.3.2 API. In our framework, we intend to provide a high-level abstraction over the TEE layer that allows developers to focus only on application logic. Further to the abstraction provided by GlobalPlatform TEE Internal Core API in OP-TEE, we define annotations `SM_ENTRY`, `SM_INPUT`, and `SM_OUTPUT` similar to the mechanism described in Section 4.1.2 to specify respectively entry points, inputs, and outputs of each TA. The provided annotations take a name as an argument that is chosen by the developer. Then, our framework produces the desired functions using some defined macros at deployment time. Likewise, the remaining required codes for developing a TA

such as communication with a client application and cryptographic operations are injected by our framework automatically. Thereby, the provided abstraction enables the developer to implement complex logic without knowledge of the TA implementation in OP-TEE.

4.3.3 Untrusted Runtime. The untrusted runtime involves an unprotected application running on Linux which is called the event manager to implement all the components outlined in Section 3.3. Our event manager leverages the GlobalPlatform TEE Client API implemented in OP-TEE client library to communicate with TAs. This API provides a set of functionalities for initializing and running TAs in the secure world as well as mechanisms for passing data to them. Therefore, when a new `LoadModule` request arrives, the event manager first establishes a connection with TEE through *TEEC_InitializeContext(...)* function which returns a context object *ctx*. Second, a logical connection with a specific TA within the scope of *ctx* is created by passing UUID of the desired TA to *TEEC_OpenSession(...)* function. When a session is opened, the TA's binary and its libraries are loaded into the secure memory. This session is then used to invoke the TA's entry points for `CallEntry` requests using *TEEC_ InvokeCommand(...)*, which takes as parameters a command ID defined for the intended entry point as well as the expected payload. Our event manager uses this operation payload to exchange events with a TA through shared memory. It is noteworthy to mention that the entry points verify the type of parameters before using them to check its value according to the expected parameter. In addition, the data passed to entry points is encrypted and authenticated. Thereby, any malicious attempts to call an undesired entry point or modify the payload would result in a failed cryptographic operation.

4.3.4 Remote Attestation and Secure Channels. Remote attestation is a crucial security feature for our framework that provides an authorized party with cryptographic proof that a known and benign TA is initially loaded on a valid TrustZone device. In contrast to Sancus and Intel SGX, the TrustZone specification offers no mechanism for remote attestation. Some existing approaches leverage hardware components such as a hardware **Trusted Platform Module** (***TPM***) or a firmware-based TPM (e.g., Microsoft's *fTPM* [58]) that provides security guarantees similar to a hardware TPM, to perform attestation. In our work, we avoid the added complexity and substantial increase of the TCB, and introducing no additional hardware. Instead, we propose an attestation mechanism based on native TrustZone features.

Our design is based on Sancus' remote attestation scheme (Section 4.1). Analogously, our approach relies on symmetric cryptography and a three-level key hierarchy. We build upon an OP-TEE proof-of-concept implementation for remote attestation that requires minimal hardware features and assumptions,[3] avoiding the use of an TPM/fTPM as they are not available on many platforms. We implemented an attestation protocol derived from the OP-TEE proof-of-concept to address the requirements of our framework, while minimizing the run-time TCB.

Our scheme assumes that each TrustZone device is equipped with a **Hardware Unique Key** (**HUK**) (also called *endorsement key* in other technologies), which can be created at manufacture time and stored either in hardware fuses or the secure eMMC partition. Similar to hardware devices like the TPM or Sancus, the Hardware Unique Key (HUK) is protected from unauthorized access and can be used as root-of-trust of the TrustZone device. Since the secure boot process guarantees system integrity, OP-TEE can be treated as the trusted base for accessing the HUK, and the isolation properties provided by TrustZone protect secure-world code at run-time. Thus, we get a strong guarantee that the HUK can only be accessed by privileged code in the secure world.

[3]See https://github.com/OP-TEE/optee_os/issues/3189/ and https://github.com/OP-TEE/optee_os/pull/4011/. Very recently, attestation is being picked up and included in release 3.17 (April 2022) by the OP-TEE team again.

To leverage the HUK and provide attestation primitives to the TAs, we developed an attestation **Pseudo Trusted Application** (**PTA**)[4] that is statically built into the OP-TEE core and runs at the same privileged execution level as the OP-TEE OS. Analogous to Sancus attestation, this Pseudo Trusted Application (PTA) uses a **Key Derivation Function** (**KDF**) to retrieve symmetric vendor and module keys from the HUK. In particular, the vendor key is computed as follows:

$$vendor\_key := KDF(HUK \parallel vendor\_id).$$

Identically to Sancus (cf. Section 4.1), `vendor_id` is a unique identifier assigned to the software vendor, and the `vendor_key` needs to be securely communicated out-of-band. The `vendor_id` is then embedded in the TA code and passed to the PTA during the attestation process. The module key, instead, is computed as

$$module\_key := KDF(vendor\_key \parallel TA\_hash).$$

The TA hash is used by the OP-TEE OS to verify the integrity of the TA at load time.[5] We made minimal changes to the OP-TEE OS to store the hash in secure memory after loading a TA, so that it could be retrieved by the attestation PTA for computing the module key. The module key can be also used to establish a secure communication channel between the TA and the remote party.

Our remote attestation process shown in Figure 11 is illustrated in detailas follows:

(1) The verifier sends an attestation request to the client application running in the normal world on the target device. The request contains a randomly generated *challenge* to provide freshness. Then, the client application passes the received *challenge* to the intended TA through shared memory.
(2) The TA leverages the attestation PTA to obtain the module key K. The attestation PTA performs a double derivation to retrieve the module key from the HUK, as explained above. To do so, the vendor identifier is passed as a parameter by the TA itself, whereas the TA hash is fetched from secure memory. After receiving K, the TA calculates a MAC D of the provided *challenge*.
(3) The TA sends D to the client application through shared memory and then D is forwarded to the remote verifier.
(4) The verifier derives the same module key K from the vendor key and TA hash. Then, the verifier uses K to compute a MAC of *challenge* and then compares it with the received MAC. If the two values match, the target device is authenticated and the TA integrity is verified.

The Key Derivation Function (KDF) used in our proof-of-concept is a simple SHA-256 hash, though we truncate the module key to 128 bits to align its size with the other TEE implementations. For MAC function we leveraged the AES-128-GCM authenticated encryption scheme, using the challenge as associated data.

Similarly to Sancus, attestation binds a TA to a specific node because the module key is (indirectly) derived from the HUK. Deploying the TA on a different node would result in a different module key. Since the TA's UUID is hard-coded, multiple instances of the same TA would have different hashes and consequently different module keys, provided that each instance uses a different UUID.

This attestation scheme gives strong guarantees that a TA is running expected code on a particular TrustZone node. The freshness of the challenge prevents replay attacks, ensuring that the

[4]PTAs are an OP-TEE concept. They provide interfaces that are exposed by the OP-TEE Core to client TAs. See https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html for more details.
[5]The TA's binary structure and loading process are described in detail at https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html.

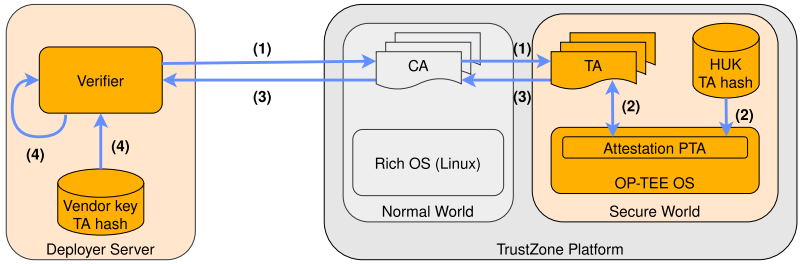


Fig. 11. Our remote attestation process for ARM TrustZone with OP-TEE. Colored components are considered trusted in our architecture.

TA is up and running at the time of the attestation. Adversaries cannot learn the module key by sniffing the network traffic, nor can they impersonate a TA to compute the response to the challenge. Besides, any attempt to modify the messages between TA and verifier would only result in a failed attestation. However, the scheme relies on the security of the HUK and vendor key: if a HUK is leaked, all TAs deployed on its corresponding TrustZone node are compromised, while leaks of a vendor key only compromises the TAs of its corresponding software vendor deployed on its corresponding node. Hence, a device manufacturer should provide software vendors with a secure interface (e.g., an authenticated API) to retrieve the vendor key for a specific TrustZone node.

4.3.5 Secure I/O. TrustZone leverages dedicated hardware components to enforce hardware isolation to the I/O devices. Specifically, TrustZone introduces the TrustZone Protection Controller to define the access restrictions for peripherals and configure them as secure and non-secure dynamically or statically. This is achieved by the reflection of Non-Secure bit into the respective peripheral. Several works take the advantage of TrustZone to establish a trusted I/O path for different purposes [15, 35, 63, 77] (Section 7.3). Our framework does not support physical I/O channels for TrustZone-based modules yet but rather interfaces Sancus-enabled I/O devices.

4.4 Comparison: SMs on Different TEEs

Table 1 summarizes the features of each TEE and highlights their main differences. This table is derived and adapted from [41]. Note that for TrustZone we consider OP-TEE OS with our attestation extensions. The key points that distinguish our TEEs are *(a)* software isolation which is supported by all three TEEs and where we provide common interfaces to control isolation and event handling; *(b)* attestation primitives, which are not available on all TEEs and we provide attestation support as necessary and establish common Attest entry point and event handler abstracting for our framework; and *(c)* secure I/O, which is only natively supported with Sancus and where we design generic scheme that can hopefully be instantiated by other TEEs, in particular by ARM TrustZone.

Software isolation and attestation are strict requirements in our design (Section 3.1), which are fully satisfied by Intel SGX and Sancus. For TrustZone, instead, we relied on OP-TEE OS for isolation, whereas we designed and implemented our own attestation protocol based on Sancus (Section 4.3.4). Other than code integrity and authenticity, attestation may also provide additional information, such as the security configuration of the underlying TCB (Intel SGX), or the guarantee that an SM is running on a specific node (Sancus and TrustZone). The latter is particularly useful when dealing with driver modules (Section 3.4). Besides, only Intel SGX protects the integrity and authenticity of data from physical attacks, thanks to its **Memory Encryption Engine** (**MEE**).

Table 1. Comparison of TEE Hardware Features and the Resulting Security Guarantees and Application Capabilities in Our Framework

	Isolation	SW Attestation	TCB Attestation	Memory Protection	Sealing	Code Confidentiality	Secure I/O	HW-Only TCB	Upgradeable TCB	Preemption	Dynamic Layout	Target ISA
Sancus	●	●	●	○	○	◓	●	●	○	◓	○	MSP430 (16-bit)
SGX	●	●	●	●	◓	◓	◓	○	●	●	●	x86_64 (64-bit)
TrustZone[a]	●	●	●	○	◓	◓	◓	○	●	●	●	ARM (32-bit)

● = Yes; ◓ = Possible; ○ = No

This table is derived and adapted from [41].
[a]: We consider TrustZone with the OP-TEE OS and our extensions for attestation.

Both Intel SGX and TrustZone support sealing, i.e., storing data securely on permanent storage. However, our framework does not currently provide any abstraction over sealing primitives, meaning that developers need to manually implement code to call such primitives. Similarly, code confidentiality (i.e., deploying encrypted SMs in order to protect sensitive code and static data) is a feature supported by all architectures, though not essential for the use cases we consider. Nevertheless, both sealing and code confidentiality could be addressed in future work. Besides, Sancus fully supports secure I/O (Section 4.1.5), while on TrustZone and Intel SGX, some extensions are needed, as proposed in related work (Section 7.3).

Concerning architectural features, only Sancus provides an hardware-only trusted computing base. In fact, Intel SGX relies on software parts such as microcode and attestation enclaves, while TrustZone includes the whole OP-TEE operating system in its TCB. However, the software can be easily upgraded to provide new functionalities or fix bugs, which is why Intel SGX and TrustZone support TCB upgrades. Furthermore, both TrustZone and Intel SGX have full support for interrupts and dynamic layout (i.e., virtual memory). Sancus, being an embedded TEE, does not support virtual memory, whereas for interrupts we offer partial support (Section 4.1.5).

4.5 Deployment

We developed a Python script called `reactive-tools` to ease the deployment and initialization of a distributed application. In essence, this script builds the SMs and then communicates with the remote event managers to bootstrap enclaves and establish connections (Section 3.3). `reactive-tools` takes as input a *deployment descriptor*, a configuration file that contains a high-level description of the application to deploy.

4.5.1 Deployment Descriptor. The deployment descriptor contains three main sections: `nodes`, `modules`, and `connections`. Figure 12 shows an excerpt from one of our sample applications.

As shown in the figure, a module is statically assigned to a node and a specific TEE technology. This is a limitation of our framework as each TEE implementation uses its own API and programming language. A possible direction for future work could be to use an unified language for defining modules, which can be an existing programming language such as Rust or a custom language such as the pseudocode we used in Figure 3; then, the framework could automatically select the right compiler/toolchain according to the node in which the SM is going to be deployed. However, it is worth noting that not all TEEs provide the same features: for example, Intel SGX supports data sealing, whereas Sancus does not support persistent storage at all.

In the `connections` section, the deployer declares connections between SMs. A connection links together two endpoints, each of them identified by the pair *<SM_name,endpoint_name>*. Moreover, each connection must indicate the authenticated encryption algorithm to use for protecting events

```
nodes:
- type: sancus
  name: node_sancus
  host: node-sancus
  vendor_id: 4660
  vendor_key: 0b7bf3ae40880a8be430d0da34fb76f0
  reactive_port: 6000
modules:
- type: sgx
  name: webserver
  node: node_sgx
  vendor_key: cred/vendor_key.pem
  ra_settings: cred/settings.json
connections:
- from_module: button_driver
  from_output: button_pressed
  to_module: gateway
  to_input: button_pressed
  encryption: spongent
```

Fig. 12. Excerpt from a deployment descriptor in YAML format. Three main sections are defined: nodes, `modules`, and `connections`. Each node and module has a `type` field indicating the TEE technology used; besides the fields that are common to all TEEs, each platform also requires other specific parameters. The fields of a connection, instead, do not change. The complete deployment descriptor (in JSON format) used in this example can be found at https://github.com/AuthenticExecution/examples/blob/main/button-led/descriptor.json.

in transit. Currently, both Intel SGX and TrustZone support AES-**Galois-Counter Mode** (**GCM**) and SPONGENT (both with 128 bits of security), whereas Sancus only supports SPONGENT.

4.5.2 Deploying a New Application. `reactive-tools` provides separate commands for the deployment and attestation of modules, as well as for the establishment of connections. To this end, the deployer would run the `deploy`, `attest`, and `connect` commands in sequence, each time providing as input the same deployment descriptor. If all these commands succeed, the deployment is complete and the application is ready to process events.

4.5.3 Interfacing with the Application. At runtime, the deployer might want to interact with the application in a secure way, e.g., to initialize a web server or to get some metrics from a database. However, the `CallEntry` API provided by the event managers to call modules' entry points is not secure, as for this kind of message the data is transmitted unencrypted over the network. Hence, the deployer would need to manually implement encryption and/or authentication mechanisms for data exchanged with the application.

Nevertheless, to facilitate the secure communication between a deployer and their SMs, our framework provides a convenient feature called *direct connections*. Unlike normal connections between two modules, a direct connection uses the deployer's machine as source endpoint. Since connection keys are generated and distributed at deployment time by our script, it is possible to also use such keys to generate output (or request) events and send them directly to running modules.

These connections can be defined in the deployment descriptor together with the others, and are characterized by the additional `direct:true` field. The deployer can then either generate new events manually or automatically using `reactive-tools`; either way, the receiving module processes these events just as any other event. Since the deployer's machine is assumed to be trusted, the same security guarantees as normal connections apply for direct connections as well.

4.6 Attestation and Key Management

To ease the attestation of modules and the storage of encryption keys, we developed an optional component called the *Attestation Manager*. It consists of an Intel SGX enclave that is deployed on the infrastructure together with the applications, and to which the deployer and `reactive-tools` can send requests through a dedicated command-line interface.

When the attestation manager is deployed, it first needs to be attested using the usual Intel SGX attestation process. Similarly to the attestation of SMs, a shared secret is exchanged between the challenger (i.e., the deployer) and the enclave, which is then used to establish a secure channel.

After the initialization is complete, the attestation manager is ready to receive `Attest` requests from the deployer; the attestation manager then takes care of executing the actual attestation logic, based on the TEE technology used by the module. If the attestation succeeds, the attestation manager stores the module key securely in the enclave memory, and it can be fetched (or used indirectly) by the deployer at any time.

4.6.1 Advanced Usages. The attestation manager, when used, may enable additional use cases. Below, we shortly describe some of them. Note that we only aim at providing an informal and high-level outline of such use cases, to motivate the usage of a centralized attestation/key management component. As such, we will not go into details on the design nor perform a security analysis.

- *Key management.* The simplest use case is to merely store credentials (i.e., module and connection keys), which can be accessed by the deployer or any other authorized entities when needed. Credentials can be stored in enclave memory or persisted in storage using sealing capabilities of TEEs such as Intel SGX. Note that the latter case may introduce vulnerabilities against rollback attacks [3].
- *Distributed confidential computing.* Certain applications process privacy-sensitive information, such as health-related data. TEEs already protect against honest-but-curious infrastructure providers, while attestation ensures the authenticity of application code and proves that the isolation mechanisms of a TEE are correctly in place. However, in some cases, it may be desirable to conceal sensitive data even from the deployer, who normally has access to connection keys and could potentially decrypt data in transit between modules. Therefore, the attestation manager could be used to generate and distribute connection keys to modules without the deployer learning any information about their value.
- *Root of Trust in edge devices.* In some edge scenarios such as automotive/**Vehicle-to-Everything (V2X)**, nodes may be part of a local network that does not have continuous access to Internet services. In such cases, it might be useful to deploy an attestation manager on the network responsible for the initialization and attestation of local enclaves. For instance, we could envision a TrustZone gateway in a car that initializes all Sancus' **Electronic Control Units (ECUs)** when the engine is turned on, refusing to start the car if any of the attestations fails [65]. In this scenario, the gateway acts as the *root of trust* of the system, and its remote attestation would be possible as soon as the system comes back online.

5 EVALUATION

To evaluate our framework, we developed a prototype for a Smart Home application that combines together Intel SGX, ARM TrustZone, and Sancus. In this chapter, we introduce our Smart Home environment and motivate security requirements (Section 5.1); then, we describe the hardware and software setup of our evaluation (Section 5.2). After that, we provide a detailed performance analysis of our application (Section 5.3), and finally discuss TCB size and development effort (Section 5.4).

5.1 Smart Homes

A smart home is a residential property integrated with technology to remotely control appliances and systems, such as lighting, heating and cooling, and entertainment. The adoption of smart home devices has increased rapidly, with an estimated 250 million smart homes worldwide in 2021, expected to reach 350 million by 2023.[6] Despite the benefits such as enhanced comfort and energy efficiency, security concerns arise from inadequate security measures in many smart home devices. Studies like [16] have shown that many vulnerabilities come from poor authentication, missing encryption, insecure software updates, and insufficient access control, leading to potential privacy violations and unauthorized access to personal information and control of home devices.

Thus, our framework can be utilized in the context of smart homes to provide robust security guarantees. Through the use of TEEs, both code and data are protected in use, and attestation ensures code integrity. Additionally, our deployment approach supports secure software updates and guarantees that all connections are encrypted and mutually authenticated. This eliminates a broad range of attacks while minimizing the development and deployment effort.

5.2 Testing Environment

To evaluate our framework, we developed a prototype for a small smart home application consisting of three simulated IoT devices: a temperature sensor, a smart thermostat, and a light switch. A smart home gateway, similar to existing projects such as Home Assistant,[7] manages the IoT devices and enforces some custom-defined logic, e.g., to automatically turn on or off the heating if the temperature goes below or above predefined thresholds. Furthermore, a web application is made available to local and remote users, to monitor the house and perform some operations on demand, e.g., to switch the lights on or off. Both smart thermostat and light switch are connected via Secure I/O to an LED, indicating whether heating and lights are on or off at a certain time. The source code of our prototype is publicly available on Github.[8]

We implemented an application such as described above using five SMs:

- *web*: Exposes a web application to allow external users to interact with the smart home;
- *gateway*: Manages all the IoT devices and enforces a user-defined logic, while at the same time interacting with *web* to send status data and receive commands from external users;
- *temp_sensor*: Simulates a sensor that provides the current temperature in the house;
- *thermostat*: Simulates a smart thermostat, to enable or disable the heating system;
- *light_switch*: Simulates a smart light switch, to enable or disable the lights.

As depicted in Figure 13, we deployed our smart home application as follows:

- *web* as an Intel SGX SM, on an Intel NUC7i3BNHXF with Intel Core i3-7100U;
- *gateway* as a TrustZone SM, on a BD-SL-i.MX6 board with ARM Cortex-A9 running at 1GHz;
- *temp_sensor*, *thermostat* and *light_switch* as Sancus SMs, on three different 16-bit OpenMSP430 microcontrollers running at 8 MHz.

In our setup, all nodes communicate over TCP/IP in the same local network; However, multiple deployment strategies may be adopted, e.g., deploying *web* in a public cloud. Since our Sancus microcontrollers can only communicate through UART, we wrote a Python script that acts as a passive bridge, converting UART streams to TCP/IP packets and vice versa without being able to decrypt or manipulate the content of events.

[6] https://www.statista.com/topics/2430/smart-homes.
[7] https://www.home-assistant.io/.
[8] https://github.com/AuthenticExecution/examples/tree/main/smart-home.

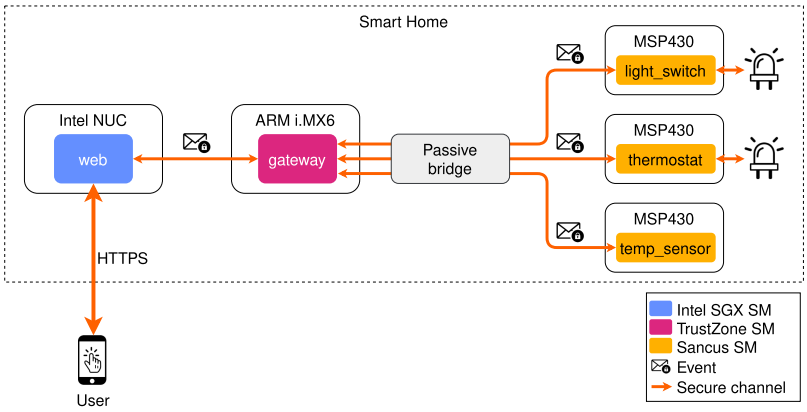


Fig. 13. Setup of our *smart home* application. Each SM is deployed on a different node, either running Intel SGX, TrustZone, or Sancus. All communication between modules is encrypted and authenticated by our framework, while the communication between *web* and users relies on HTTPS and mutually authenticated. Sancus nodes are connected via UART to a passive bridge, which converts between UART streams and TCP/IP.

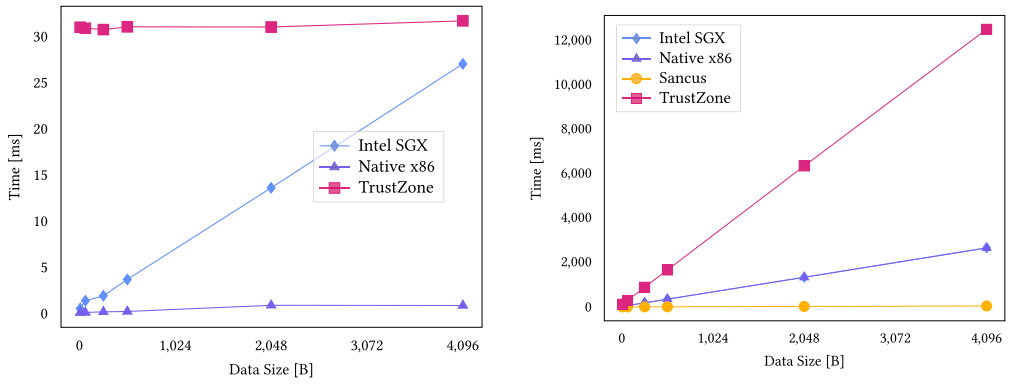
All communication is end-to-end protected from the user to the IoT devices. As shown in the figure, communication between SMs is carried over encrypted and authenticated *events*, leveraging our framework (Section 3). Instead, the communication channel between *web* and the user is encrypted using HTTPS. User and *web* mutually authenticate each other: *web* is authenticated by the user during the TLS handshake, during which *web* presents an ephemeral X.509 certificate generated after successful attestation; the user, instead, authenticates by providing a *secret token*, similarly to session cookies or JSON Web Tokens (JWTs).

5.3 Performance Benchmarks

We analyzed the performance of the Smart Home application introduced above. Cryptographic overhead was assessed per node and an end-to-end evaluation was conducted by measuring the round-trip time of an event sent from *web* to *light_switch* and return. The impact of a module update was evaluated by measuring application downtime during the update.

5.3.1 Cryptographic Operations. Figure 14 shows the average time calculated on each TEEs to perform crypto operations using either AES-GCM or SPONGENT as authenticated encryption library, with 128 bits of security. We used different sizes for the data to encrypt, ranging from 8 to 4096 bytes. To provide a reference, we also carried out the same experiments on a simple Linux x86 process without TEE protection, running on the Intel SGX node, whose results are depicted in the third column (*Native x86*). The plots show average timings computed over 110 iterations, except that Sancus values have been extrapolated from previous experiments [52].

Modern x86 and ARM processors include AES instructions in their instruction set, allowing them to perform cryptographic operations in hardware for improved performance and security. As shown in Figure 14(a), a single AES encryption is extremely fast natively, taking around 878 μs to encrypt 4,096 bytes of data. The overhead caused by the enclaved execution in Intel SGX significantly slows down the execution of these functions with a factor that increases linearly with the size of the data: encrypting 4,096 bytes takes up to 31 times more than natively. Regarding TrustZone, instead, it can be observed from Figure 14(a) that the payload size had little impact on the encryption time, with values between 30 and 31 ms and a standard deviation of 0.3 ms. We noticed that TrustZone has a fixed overhead due to some system calls that need to be called to initialize



(a) AES-128 (b) SPONGENT-128

Fig. 14. Average timings to perform encryption using AES-128 and SPONGENT-128 on different TEEs. As a reference, we also measured the average time spent on a Linux x86 process without TEE protection (i.e., the *Native x86* line). All measurements are in milliseconds.

each cryptographic operation. In our case, the TA has to call `TEE_ResetOperation`, `TEE_AEInit` and `TEE_AEUpdateAAD`.[9] Instead, we could not provide any data for Sancus, as it does not include an AES module in its architecture.

The lack of an AES engine in Sancus is a serious issue for our framework, as it prevents Sancus modules to communicate securely with modules of other TEEs and the deployer. To circumvent this problem, a software implementation of the SPONGENT crypto library was implemented in previous work, in both C/C++ and Rust. Thus, both Intel SGX and TrustZone modules can leverage such library to exchange protected events with Sancus modules. However, the overhead for performing such operations in software is significant: as shown in Figure 14(b), while for small amount of data the difference is less prominent, the performance heavily degrades for bigger data. Compared to AES and for payloads ranging between 8 and 4,096 bytes, SPONGENT is up to 3,026 times slower on native x86, up to 99 times slower in Intel SGX, and up to 395 times slower in TrustZone. Instead, experiments show that SPONGENT is significantly faster in hardware.[10] Unfortunately, this poor performance may completely compromise the real-time requirements of certain use cases, and replacing SPONGENT with AES might be appropriate, depending on the acceptable processing time and power consumption on Sancus nodes for a specific use case.

5.3.2 End-to-end Measurements and RTT. We performed end-to-end experiments on our Smart Home application to measure execution times and **round-trip time** (**RTT**). In particular, we evaluated the scenario where an external user manually enables the lights by sending an HTTP request to *web*. This event triggers the following flow:

(1) *web* sends an event to *gateway* containing 2 bytes of payload that encodes the desired action (i.e., enable the lights);
(2) *gateway* decrypts the event and generates a new one for *light_switch*, using the same payload;

[9] https://optee.readthedocs.io/en/latest/architecture/crypto.html.
[10] The SPONGENT [10] family of light-weight hash functions are optimized for hardware implementation. We have confirmed in independent experiments that the implementation in Sancus indeed outperforms a software implementation on other architectures by several orders of magnitude.

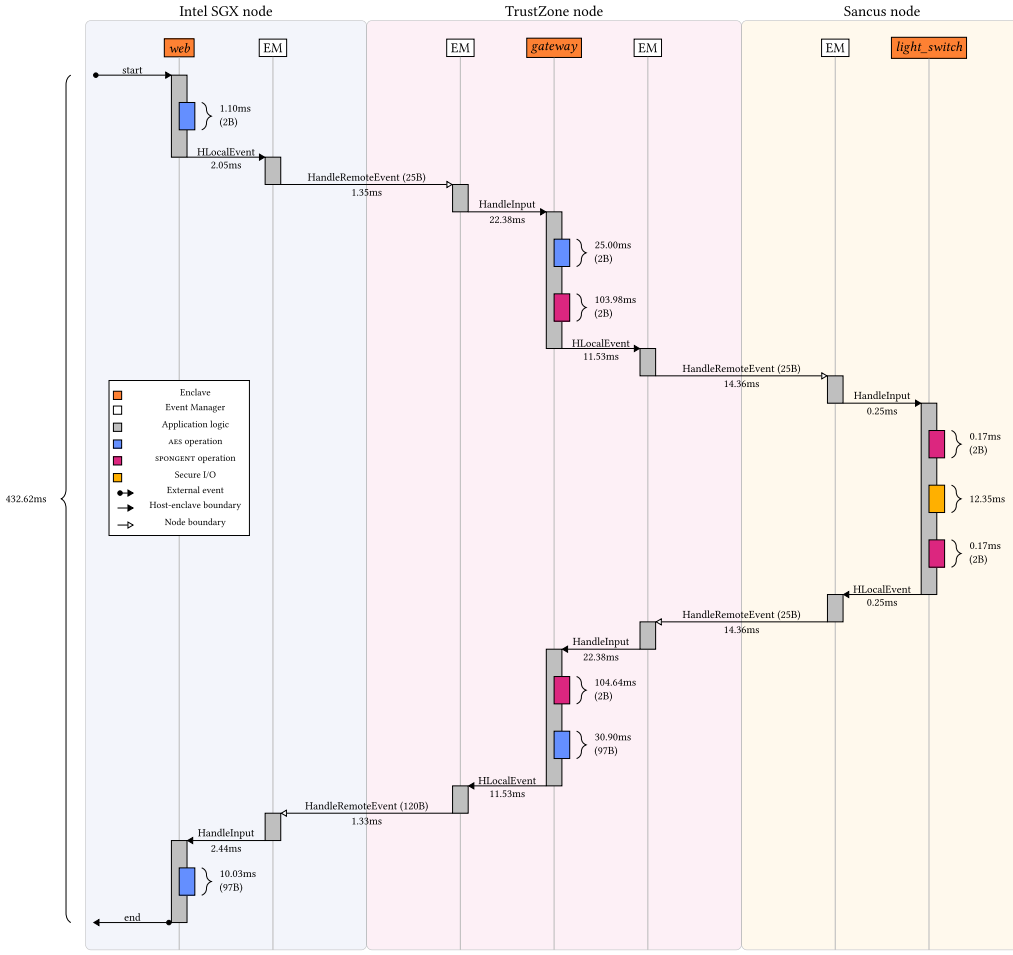


Fig. 15. Sequence diagram showing the control flow and timings of the Smart Home application in Figure 13. The diagram illustrates a scenario where the user manually switches the lights; bar lengths are *not* to scale.

(3) *light_switch* receives and decrypts the event, then dispatches an internal event to turn the LED on. After that, it sends an event back to *gateway* as a notification that the lights have been turned on, using the same 2 bytes of payload as before;
(4) *gateway* receives the event and updates its internal state. Then, it generates an event for *web* containing the current status of the smart home in JSON format, whose size is 97 bytes;
(5) Finally, *web* receives the updated status. Starting from this moment, the user will be able to see in the web application that the lights have been successfully turned on.

Figure 15 shows the sequence diagram of the flow just described. Time was recorded to measure the performance impact of the most important steps: encryptions/decryptions, host-enclave boundary crosses, Secure I/O, and transmission times. We also measured the end-to-end round-trip time (RTT), consisting of the difference in time between the *start* and *end* points in *web*. The sequence diagram shows average timings over 110 iterations. The average RTT was 432.62 milliseconds, which includes 8 encryptions/decryptions (4 using AES and 4 using SPONGENT), 8 host-enclave boundary crosses, and 4 network transmissions between nodes. To better understand the performance impact of each operation, timing values have been aggregated and shown in Table 2.

Table 2. End-to-end Measurements of the Smart Home Application, Aggregating the Average Time Spent for Performing Specific Tasks

Operation	Time (ms)	% of RTT
AES instructions	67.03	15.49
SPONGENT HW instructions	0.34	0.08
SPONGENT SW instructions	208.62	48.22
Host-enclave boundary	72.81	16.83
Secure I/O	12.35	2.86
Network delay	31.40	7.26
Other	40.07	9.26
RTT	**432.62**	**100.00**

The last row shows the round-trip time (RTT), which consists of the sum of all the intermediate times. In the third column, we show the impact of each task on the RTT.

Around 64% of the RTT was spent on encrypting/decrypting events: On the one hand, cryptographic operations performed in hardware were generally fast, taking a total of 67.03 ms for AES and 336 μs for SPONGENT. The former, however, was heavily affected by the fixed TrustZone overhead for initializing cryptographic operations (Section 5.3.1), which may be optimized in future releases of OP-TEE. On the other hand, SPONGENT operations performed in software caused huge performance penalties on the application, taking a total of 208.62 ms (~48.2% of the RTT).

Crossing the boundary between host and enclave accounted for the 16.83% of the RTT (72.81 ms); in particular, we observed that the slowest transitions between untrusted and trusted domains were registered in the TrustZone node, with an average of 17 ms for each transition. Domain transitions in Sancus proved to be extremely fast, each of them requiring only around 250 μs. In Intel SGX, instead, it took roughly 2.2 ms to enter/exit the enclave.

Poor network performance in Sancus had a non-negligible impact on our measurements. In fact, the average total transmission time was 31.40 ms, 91% of which spent on the transmission of events to and from Sancus microcontrollers. As described in Section 5.2, this is due to the low baud rate (57,600 bps) of the UART interface and the overhead caused by the passive device used to communicate with the Sancus microcontrollers.

From our experiments, we observed that the Secure I/O overhead was on average 12.35 ms which, according to Section 4.1.5, includes: *(1)* sending an event from *light_switch* to the LED driver module (`mod-driver`); *(2)* sending an event from the LED driver module to the LED MMIO module (`mod-mmio`); *(3)* executing the logic to enable the LED. We did not measure such steps independently in this evaluation, since the main objective of our evaluation was to assess and evaluate the inter-operation between different TEEs and measure the communication overhead across different implementations. However, we extensively evaluated Sancus in our previous work [51].

Finally, an average of 40.07 ms (~9.26% of the RTT) was due to other application logic that was not measured during the experiments. In the sequence diagram, this consists of the grey rectangles minus the time spent for performing cryptographic operations and Secure I/O. For example, this logic includes the time spent for the look up of a `Connection` structure given a specific connection ID, or for finding the correct SM to which deliver an event. The performance of such operations is implementation-specific, and may be improved by using more efficient logic and data structures.

In summary, our experiments demonstrate that the overhead of our framework does not affect user experience for near real-time applications such as Smart Home. Considering our setup, after

Table 3. Average Time to Perform a Software Update on Different TEEs, All Measurements are in Seconds

Module	TEE	Build	Deploy	Attest	Connect	**Total**
web	Intel SGX	1.199	0.247	1.794	0.775	**4.027**
temp_sensor	Sancus	0.443	2.510	2.576	0.251	**5.833**
gateway	TrustZone	0.976	0.438	0.329	1.277	**3.063**

We carried out the experiments on the Smart Home application, using the same setup described in Figure 13. Build, deployment, and attestation times are highly application-specific; in our case, all modules were re-deployed without code changes and no state transfer between old and new instances was implemented.

sending out a command, the user receives a feedback in less than 500 milliseconds. Our micro-benchmarks also show that the RTT can be further improved by adopting different deployment strategies, e.g., by merging *web* and *gateway* together in a single SM. For real-time applications, however, this may not be sufficient and additional capabilities might be needed, such as hardware acceleration for all cryptographic operations. We also note that all experiments were performed on a prototype that is not fully optimized, and all modules were built in debug mode; Thus, we expect that the overall performance would improve in a production environment.

5.3.3 Module Update. Real-world applications require changes at runtime, e.g., software updates or module migration from one node to another. This process impacts on the availability of the whole application, because updating the application unavoidably results in a temporary loss of connectivity. Therefore, we evaluated the overhead of a software update in our framework, for all our supported TEEs. Again, we carried out our experiments on the Smart Home application (Figure 13); results are shown in Table 3, each value representing the average over 10 iterations.

In our experiments, we re-deployed the original SMs without any code changes. As shown in Table 3, the build time was rather small for all TEEs, ranging from nearly 450 ms for Sancus to around 1.2 seconds for Intel SGX. However, building times highly depend on the amount of changes made to the original code, and whether the compiler's cache was used or not. Hence, these measurements are not really interesting because highly variable. Similarly, deploying the new SM on the infrastructure depends on the size of the SM itself and the bandwidth of the communication medium. The table shows that the deployment of a Sancus module is a slow process, taking an average of 2.510 seconds; this can be explained by the slow communication channel between the Sancus node and the deployer, as explained in Section 5.3.2. The deployment process for Intel SGX and TrustZone takes only 247 ms and 438 ms, respectively; Binary sizes are shown in Table 4.

Our experiments show that the attestation time greatly varies according to the TEE used. Attesting an Intel SGX enclave took an average of 1.794 s; in fact, the modified sigma protocol used in our framework consists of several steps, including the involvement of the **Intel Attestation Server** (**IAS**) to decrypt the quote (Section 4.2.4). For TrustZone, instead, we use a simple challenge-response protocol, as described in Section 4.3.4. Thus, attesting a TrustZone module is very fast and takes an average of 329 ms. Sancus uses a similar mechanism, though its attestation takes much longer, i.e., 2.576 s on average. However, we observed that the majority of this time was actually spent on the deployer's side to compute the module key from the module's binary; the actual challenge-response protocol only took less than 100 milliseconds.

Finally, after the new module is ready (i.e., deployed and attested), all connections need to be redirected from the old to the new instance. Following the update strategy illustrated in Section 3.7, this is the time in which the application would suffer from temporary connectivity loss. However, our experiments show that this process only takes a very short amount of time, ranging from

Table 4. Size ("Src": Source Code, "Bin": Binary Size) of the Software for Running the Evaluation Scenario

Component	Src (LOC)	Bin (B)
RIOT OS	9713	
Event Manager	1312	
Total untrusted	**11025**	**35638**
Stub code and libraries	797	
thermostat	10	9001
temp_sensor	21	9423
light_switch	10	9079
Total trusted	**838**	**27503**
TCB reduction (%)	**92.9**	**56.4**
Dev. effort (%)	**4.9**	

(a) Sancus

Component	Src (LOC)	Bin (B)
Host OS[a]	>700M	314.3M
Event Manager	623	1.2M
Total untrusted	**>700M**	**315.5M**
Fortanix EDP	>20K	
Stub code	3412	
3rd party libraries	~20K	
web	372	4.2M
Total trusted	**>43K**	**4.2M**
TCB reduction (%)	**99.9**	**98.7**
Dev. effort (%)	**9.8**	

(b) Intel SGX

Component	Src (LOC)	Bin (B)
Host OS[a]	>700M	314.3M
Event Manager	1448	17.3K
Total untrusted	**>700M**	**314.3M**
OP-TEE OS	286K	244K[b]
Stub code	1432	
gateway	101	111.3K
Total trusted	**287K**	**355.3K**
TCB reduction (%)	**99.9**	**99.9**
Dev. effort (%)	**6.6**	

(c) TrustZone

Shaded components are part of the run-time software TCB. For a fair comparison, we only consider source code (e.g., C/C++, Rust files), and not build scripts or other similar files. Besides, our evaluation does not include compilers, standard libraries, and other software layers such as hypervisors. TCB reduction is calculated as untrusted code over total code, while developer's effort as application logic (e.g., *gateway*) over total trusted code, which includes stub code but excludes third party libraries. Module binaries have been built in debug mode.
[a]: We use Debian 11 (bullseye) as reference [17];
[b]: As of 2016, according to the official documentation.

251 ms for Sancus to 1277 ms for TrustZone. It has to be noted, though, that these values only represent the time to *deliver* all the events to the SMs and event managers (`SetKey` and `AddConnection` events, respectively), and do not include the processing time in the nodes. Hence, the effective time would be a bit higher. Besides, this time is linearly dependent on the number of connections to re-establish; in our case, we had five connections for *web*, nine for *gateway*, and two for *temp_sensor*.

In conclusion, the overall time for a software update is highly variable and application-dependent. Nevertheless, the connectivity loss is only perceived during the Connect phase, which roughly takes around 150 milliseconds for each connection to re-establish.

5.4 Adapting Burden

We performed a code evaluation of our Smart Home application, results of which are shown in Table 4. Our analysis focuses on two main aspects: first, the TCB size in relation to the whole software stack; second, an estimation of the developer's effort. Concerning the TCB, we calculated both lines of code and binary sizes, while for the developer's effort we only focused on the code. We note that application modules were built in debug mode, which may have caused slightly bigger binaries.

One of the most important benefits of process-based TEEs is a substantial reduction of the TCB, leading to a considerably reduced attack surface on each node. Table 4 shows that the reduction in Sancus is 92.9% in **lines of code** (**LOCs**) and 56.4% in binary size, while on SGX this reduction is more prominent with 99.9% reduction in code and 98.7% in binary size. Interestingly, TrustZone achieves a 99.9% reduction in both, which means that OP-TEE is only a thin software layer compared to a classic operating system. It has to be noted that the code reduction in SGX and TrustZone is enhanced by the fact that our reference host OS (Debian 11) consists of more than 700 million C/C++ lines of code (LOCs), and more than one billion LOCs overall [17].

Our framework strives to simplify the development of distributed enclave applications. As shown in Sections 3 and 4, our API allows a developer to only specify the core logic of a software module, while the rest of the code is added at compile time by the framework. Our evaluation of the developer's effort consists of the ratio between the LOCs written by the developer and the total LOCs of a module, including stub code but excluding third-party libraries. Results show that the

amount of code written by the developer is minimal: only 4.9% for Sancus, 9.8% for SGX, and 6.6% for TrustZone. However, these numbers may greatly vary: our evaluation was carried on a simple prototype with a few modules; More complex applications would require higher effort.

The above evaluation can only provide rough estimates of the benefits of our framework: Performing a precise code evaluation is not a trivial task. Kernels and operating systems typically make extensive use of conditional compilation to enable/disable specific features or to target different hardware architectures. Thus, even though an operating system may have millions or more LOCs, most of them are not used during compilation. Besides, the number of LOCs is often magnified by auxiliary files such as build scripts, Makefiles, and so on. It is debatable whether these files should be included in the calculation or not: they are not part of the core logic of a software, but they can nevertheless introduce vulnerabilities at compile time. Applications, instead, may include several third-party libraries which size might not be easy to calculate: examples are closed-source and dynamic libraries. Finally, our evaluation did not consider software components such as bootloaders, firmware/BIOS, hypervisors, compilers, standard libraries, and so on. While some of these are untrusted, others are part of the TCB and must be taken into account for a complete evaluation.

6 DISCUSSION

6.1 Integrity versus Confidentiality

We have focused our security objective on integrity and authenticity, and an interesting question is to what extent we can also provide confidentiality guarantees. It is clear that, thanks to the isolation properties of protected modules and to the confidentiality properties of authenticated encryption, our prototype already provides substantial protection of the confidentiality of both the state of the application as well as the information contained in events. However, providing a formal statement of the confidentiality guarantees offered by our approach is non-trivial: some information leaks to the attacker, such as for instance when (and how often) modules send events to each other. This in turn can leak information about the internal state of modules or about the content of events. The ultimate goal would be to make compilation and deployment fully abstract [1] (indicating roughly that the compiled system leaks no more information than can be understood from the source code), but our current approach is clearly not fully abstract yet. Hence, we decided to focus on strong integrity first, and leave confidentiality guarantees for future work.

There is also a wealth of orthogonal research aiming at protecting network information flows from being used by attackers to learn the system state by means of, e.g., covert channels or routing protocols for mix networks (cf. [60] for an overview). The applicability of these approaches depends on application configuration and available system resources and may heavily impact availability guarantees. On light-weight TEEs such as Sancus, and specifically in the presence of constraints on system resources and power consumption, these additional protections are not applicable.

6.2 Availability

Our authentic execution framework focuses on the notion that if a valid input to a module is received, we can deduce that it must have originated from an authentic event at the source. By definition, this notion does *not* give any availability guarantees: only if we receive such a valid input can we make claims about the authentic event, whereas nothing can be claimed otherwise.

However, there has been recent work on enforcing availability guarantees for TEEs, specifically for Sancus [4]. At its core, this related work enables strict availability guarantees to enclaves on Sancus, such as that an enclave will be executed periodically or be enabled to serve an interrupt within a predictable time frame. This allows to make claims about how specific enclaves will behave

and *when* these enclaves will react upon events they receive. At the same time, it is still not possible to make any availability claims about events that exceed the device boundary, i.e., events that cross beyond the device itself and that are communicated over a shared untrusted medium such as a network. Overall, it may be possible to make strong claims about when and how fast a received event will be processed by an enclave, but it is not possible to guarantee that an event will necessarily arrive over the untrusted network.

Thus, we see availability as a complementary guarantee for authentic execution: If an event arrives at the device boundary, earlier work such as Aion [4] can guarantee that this event can be handled within a strict time frame, depending on the scenario's needs. Furthermore, the same work can guarantee that if *no* event is received within the expected time, the enclave will also be able to react within a guaranteed time window. This may be useful for a specific set of scenarios where disrupted availability could lead to unwanted or dangerous outcomes. By employing availability as a safeguard, the absence of events could, e.g., be used to trigger mode change and activate a local control loop to emergency-shutoff equipment until an authentic event is received.

While we do believe that this complementary notion of availability can have a benefit to certain scenarios that utilize authentic execution, we leave the combination of these two orthogonal research directions for future work.

6.3 Hardware Attacks and Side-Channels

Although hardware attacks and side-channels are explicitly ruled out by our attacker model (Section 2.3), it is necessary to discuss the impact in case an attacker would have given access to such techniques. We leave an analysis of our implementation for side-channels for future work.

An attacker that successfully circumvents the hardware protections on a node would be able to manipulate and impersonate all modules running on *that node*. That is, the attacker would be able to inject events into an application but only for those connections that originate from the compromised node. The impact on the application obviously depends on the kind of modules that run on the node. If it is an output module, the application is completely compromised since the attacker can now produce any output they want. If, on the other hand, it is one among many sensor nodes, that get aggregated on another node, the impact may be minor.

Depending on the node type different attack vectors apply. A wealth of literature targets Intel processors and SGX enclaves [11, 24, 64, 67, 70] that enable the extraction of cryptographic keys or other application secrets. On embedded TEE processors such as Sancus, many side-channels such as cache timing attacks [25, 40] or page fault channels [75] are not applicable. Interrupt-based side channels [66] do pose a challenge and have been mitigated in orthogonal research in hardware [12] and in software [71].

7 RELATED WORK

7.1 TEE Frameworks

In recent years, TEEs have gained more and more popularity in both research and industry, due to their properties and the security guarantees they provide. However, developing TEE applications is not trivial as software developers are normally required to have some expertise in the field; moreover, more and more TEEs are emerging nowadays, which makes it more difficult to write heterogeneous applications or simply to port an application from one TEE to another.

To this end, Keystone [32] is an open-source framework for building customized TEEs; it provides generic primitives such as isolation and attestation, and hardware manufacturers and programmers can tailor the TEE design based on what they actually need. More software-based frameworks such as OpenEnclave [54] and Google Asylo [23] aim at providing a general API for writing

enclave applications, allowing developers to write code that is not tied to a specific TEE and can therefore be easily ported to multiple platforms. Our framework uses a different approach, which consists of abstracting the TEE layer away from developers in order to write enclave applications just like normal ones. Other projects such as Enarx [20], follow the same principles as ours, although they mainly focus on high-end TEEs such as Intel SGX, TDX, and AMD SEV. To the best of our knowledge, our framework is the only one that support heterogeneous architectures, combining high-end systems, IoT devices, and small embedded microcontrollers.

Process-based TEEs such as Intel SGX strive in minimizing the TCB, as only the critical part of an application can be moved to the enclave. However, this comes with significant performance penalties, especially when crossing the boundary between the enclave and the untrusted domain. Instead, VM-based TEEs such as AMD SEV [30], Intel TDX [26], and ARM CCA [36] are substantially better in terms of performance, but have a much bigger attack surface as the entire virtual machine is part of the TCB. As future work, we might investigate the use of VM-based TEEs combined with lightweight operating systems such as the formally-verified seL4 microkernel [31].

7.2 Mutually Authenticated Enclaves

At deployment time, our framework leverages symmetric connection keys to establish secure and mutually-authenticated channels between modules. Modules do not need to explicitly attest each other, but rather they rely on the fact that connection keys are securely distributed to modules *only* after successful attestation. The implicit assumption here is that such keys are known only by the modules and their deployer, and are always kept in secure memory and transmitted over encrypted channels. A similar idea was applied to the automotive scenario by creating a mesh of mutually-authenticated Sancus enclaves to provide integrity and authenticity of CAN messages exchanged between the ECUs of a vehicle [65].

In Reference [76], multiple Intel SGX enclaves of an application establish mutual trust during the initialization of TLS sessions, using certificates previously generated by central servers upon successful attestation. Marblerun [42] uses the same principle to create a service mesh between enclaves in a Kubernetes cluster. Our framework, instead, relies on symmetric encryption for the establishment of secure channels, as public-key cryptography is not supported on small microcontrollers like Sancus. Although asymmetric encryption and client certificates seem a better choice in terms of key management and authentication, we decided to leverage symmetric connection keys even on Intel SGX and TrustZone in order to allow communication with Sancus modules.

7.3 Secure I/O

There are several intensive previous solutions aiming at providing a trusted path between an authorized trusted application and I/O devices that guarantees the integrity and authenticity of I/O data. McCune et al. [44] have proposed a BitE framework that leverages a trusted mobile device to establish an encrypted and authenticated input channel between I/O devices and an application running on a TPM-equipped untrusted platform. However, data inside the host platform is not isolated, hence the OS kernel must be part of the TCB. Bumpy [45], a succeeding work from the same research group, addresses this limitation by relying on the Flicker TEE [43] and using the encryption-capable keyboard to provide a safe pathway from input devices through untrusted environments. Bumpy uses dedicated hardware and hence cannot generalize to arbitrary I/O devices. Our approach to trusted I/O improves over Bumpy by greatly reducing the size of the software TCB, from a full OS to less than 1 kLOC. By using Sancus as a TEE, we enable the integration of attestable software and I/O encryption directly into the input device. When communicating with a host that interacts with Sancus nodes, technologies such as Flicker, TrustZone or Intel SGX can be used to protect host services and to further reduce the TCB.

Several studies investigated a hypervisor-based approach. The techniques offered by Zhou et al. [78, 79] aim at establishing a trusted path between an input peripheral and an application using a hypervisor to run the untrusted OS and application endpoints providing the necessary device drivers in separate VMs. DriverGuard [13] is a hypervisor-based mechanism that leverages a combination of cryptographic and virtualization techniques to protect I/O device control, I/O data flows against attacks from a malicious guest kernel. These systems rely on hardware interfaces, peripherals, and the USB bus to behave according to specification. Rogue devices on the USB bus, e.g., hardware keyloggers, can intercept messages between an I/O device and the host. Although our design focuses on integrity guarantees, our prototype protects communication channels using authenticated encryption. A more recent work, SGXIO [69], provides support for generic, trusted I/O paths for enclaves in Intel SGX by a composite of TPM, Intel SGX, and hypervisor techniques. Thus, SGXIO introduces a formally verified hypervisor (seL4 is recommended) to establish a trusted path while Intel SGX is in charge of protecting user applications from an untrusted OS. This approach improves upon existing generic trusted paths for x86 systems using Intel SGX's easy programming model. Similarly, Aurora [37] utilizes a hypervisor running in the **Secure Management Mode** (**SMM**) of Intel x86 processors as well drivers in the Driver eXecution Environment of open-source UEFI firmware to implement a concept not unlike the protected driver modules from our work. A TPM is used to store Intel Boot Guard measurements of the firmware and hypervisor, enabling mutual attestation between the SMM side and application modules in SGX enclaves. Omitting a hypervisor layer, BASTION-SGX [55] introduces a trusted Bluetooth controller and application support to establish a trusted channel between SGX enclaves and Bluetooth devices. SGX-USB [72], instead, places a proxy device between the host and an I/O peripheral connected through USB, to establish a secure path to an Intel SGX enclave. Unlike the other SGX-based solutions, SGX-USB does not rely on software modifications or additional trusted hardware in the host platform.

Fidelius [21] leverages only hardware settings with no reliance on hypervisor security assumptions to assist the establishment of a trusted path from input and output peripherals to a hardware enclave. More specifically, Fidelius utilizes a trusted dongle (e.g., Raspberry Pi) to protect the entire I/O path between the keyboard/screen and a small JavaScript interpreter that runs inside an Intel SGX enclave during web browsing sessions under a compromised OS and browser. The trusted dongle captures user input and sends a stream of encrypted keystrokes to an attested local enclave. Then, the web enclave decrypts and updates the state of the relevant trusted input field and eventually forwards it to the remote server. In the opposite direction, a series of encrypted overlays are sent from the enclave to the display. The trusted path between a dongle and the web enclave is established through pre-shared symmetric keys. Conversely, the web enclave and remote server attest to each other and establish a secure channel. Fidelius uses an LED indicator on each dongle and a status bar on the screen to ensure the establishment of the trusted paths; however, this leads to a high cognitive load on the users as they must monitor continuously different security indicators. Fidelius leverages a trusted external device to guarantee the integrity and authenticity of the transferred I/O data, while our solution for secure I/O relies only on the isolation and attestation properties in Sancus and does not suffer from micro-architectural attacks and relay attacks.

ARM TrustZone provides isolated execution combined with generic trusted I/O paths. In TrustZone, different peripherals, including parts of RAM, IO devices, and interrupts, can be dynamically configured by a set of hardware controllers to be accessible only in the secure mode. This is achieved by the reflection of the NS bit into the respective peripheral. In contrast to Intel SGX, TrustZone assumes that all secure application processes are trusted and does not isolate them in hardware. However, several prior works take the advantage of TrustZone to establish a trusted I/O path with much focus on balancing the size of TCB against functionality. For example,

TrustUI [35] builds trusted paths by splitting device drivers into an untrusted backend and a trusted front-end. The two parts communicate through shared memory by leveraging proxy modules running in both worlds. Similarly, TruZ-Droid [73] runs only half part of the HTTP and SSL protocols in the secure world to reduce the TCB size. VeriUI [39] proposes a login protection mechanism by moving all layers of the communication stack and UI library into TrustZone. TrustOTP [63] relies on tiny drivers running in the secure world to protect the one-time password display. Secure I/O in TrustZone is a well-researched topic that has got a great deal of attention in the past years. Apart from the projects described above, many others exist in the literature [33, 34, 74, 77]. More recently, a notion of trusted I/O was also implemented in Android smartphones to protect user transactions [15]. In our work, we did not implement a secure I/O solution for TrustZone, but rather we interfaced Sancus-enabled I/O devices with Intel SGX and TrustZone enclaves, providing exclusive and confidential access to isolated host drivers without trusting the underlying communication channels.

Efforts have recently been made to enhance TEEs by incorporating hardware acceleration, particularly GPUs. This enables software modules to exclusively access external devices for computationally intensive tasks, such as machine learning model training. Several academic projects propose hardware extensions to GPUs to enable isolation and strict access control, including Graviton [68], HIX [28], and HETEE [80]. StrongBox [18] utilizes TrustZone extensions already present in ARM devices to provide a solution without requiring hardware changes. This trend has also influenced commercial products, such as NVIDIA's new H100 Tensor Core GPU that supports confidential computing [53], and Intel TDX Connect, which offers extensions to the TDX architecture to securely assign I/O devices to trusted domains [27]. Our architecture can utilize hardware acceleration to enhance the use cases we are considering (Sections 2 and 5.1). For instance, it can be used to process sensor data for computing statistics or training machine learning models.

8 SUMMARY & CONCLUSIONS

In this article, we present a software architecture to securely execute distributed applications on shared, heterogeneous TEE-infrastructures. We have extended Sancus, a light-weight embedded TEE with support for secure I/O, interfaced it with the commercial of-the-shelf TEEs ARM TrustZone and Intel SGX, and implemented and evaluated a software development framework, which enables the execution of reactive (event-driven) distributed applications on a shared infrastructure with strong authenticity guarantees and in the presence of capable attackers, while relying on a very small TCB. We have implemented (and foresee many more) compelling use cases in IoT and control systems. While much of our work is focused on small embedded microprocessors, we demonstrate that our solution to be generally applicable to, e.g., build secure distributed applications that integrate IoT sensing and actuation with cloud-based data processing or aggregation. Yet, implementing secure I/O is more challenging on non-embedded architectures. Extensions to our work would be to strengthen security guarantees while maintaining a small TCB, fully integrating a notion of availability based on [4], and to formalize the security argument using recently proposed logics (e.g., [8]) for reasoning about TEEs.

APPENDIX

A LIST OF ASSUMPTIONS AND REQUIREMENTS

This work relies on a number of assumptions on hardware, software, and the environment. At the same time, we put specific requirements on the realization of concepts, entities, and mechanisms that we abstract from here, e.g., secure I/O, the infrastructure provider, and the deployment process. Below we list and justify these assumptions and requirements for easier reference.

A.1 Assumptions

Table A.1.1. Assumptions on Hardware

Ref.	Reproduction	Comment
1	*The execution infrastructure offers specific security primitives – standard enclaves plus support for secure I/O.*	These hardware technologies are the base for authentic execution.
3.1	*We assume the underlying architecture is a TEE.*	TEEs allow us to isolate source modules from other code running on a node and to minimize the TCB.
3.1	*The TEE provides an authenticated encryption primitive.*	This cryptographic scheme allows for integrity-protected and confidential communication between a module and other modules or the deployer.
3.4	*The infrastructure offers physical input and output channels using protected driver modules that translate application events into physical events and vice versa.*	Without a secure I/O mechanism, authentic execution is impossible. Here we assume hardware support for some form of protected driver modules.
4.3.4	*Our scheme assumes that each TrustZone device is equipped with a Hardware Unique Key (HUK).*	A trust anchor is needed for any remote attestation scheme. For our TrustZone implementation of remote attestation we assume a HUK.

Table A.1.2. Assumptions on Attacker

Ref.	Reproduction	Comment
2.3	*Attackers can manipulate all the software on the nodes, including OS, and can deploy their own applications on the infrastructure.*	We assume a strong attacker that can mess with all software except what is protected by the hardware, e.g., TEEs and driver modules.
2.3	*Attackers can also control the communication network that nodes use to communicate with each other, can sniff the network, modify traffic, or mount man-in-the-middle attacks. With respect to the cryptographic capabilities of the attacker, we follow the Dolev-Yao model*	This is a common assumption for a network-based attacker or an attacker that has complete control over at least one of the networking stacks of a connection.
2.3	*The attacker does not have physical access to the nodes.*	Otherwise this would allow, e.g., for physical side channel and bit faulting attacks, which are out of scope here.
2.3	*Neither can they physically tamper with I/O devices.*	Doing so would allow an attacker to mess with physical inputs or the connection of devices to I/O ports, undermining the requirements on secure I/O and the authentic execution guarantees.
2.3	*We also do not consider side-channel attacks against our implementation.*	This could leak secrets such as a module key which would allow to bypass the authentic execution mechanism. Countermeasures against such attacks exist but are highly hardware-dependent.

Table A.1.3. Assumptions on Software

Ref.	Reproduction	Comment
3.1	*Both a module's code and data are located in contiguous memory areas called, respectively, its code section and its data section.*	This is mainly a simplification to support TEEs that do not allow complex memory management, e.g., Sancus.
3.1	*We assume a correct compiler.*	Without this assumption one could not deduce that the measured binaries of secure modules implement the expected source code functionality.

Table A.1.4. Assumptions on Infrastructure and Credentials

Ref.	Reproduction	Comment
2.4	*The deployer's computing infrastructure is assumed to be trusted.*	While this can be alleviated by using TEEs, eventually a trusted system is needed to verify attestations on behalf of the deployer.
7.2	*Connection keys are known only by the modules and their deployer.*	Leaking Connection keys might allow an attacker to forge messages and bypass the authentic execution mechanism.
3.4	*Driver modules are part of the trusted infrastructure*	As they take exclusive access of I/O devices to process physical inputs and outputs, they are controlled by the trusted infrastructure provider.
3.4	*Driver module keys are only known to the infrastructure provider.*	Otherwise, attackers could take control of driver modules on their own.

A.2 Implementation Requirements

Table A.2.1. Requirements on Secure I/O and Driver Modules

Ref.	Reproduction	Comment
IO1	*The infrastructure provider configures the physical I/O devices as expected, i.e., that the desired peripherals are connected to the right pins and thus mapped to the correct Memory-Mapped I/O (MMIO) addresses in the node.*	If this was not the case, the inputs registered by the drivers would not correspond to the actual physical inputs, and vice versa for physical outputs.
IO2	*The infrastructure must provide a way for the deployer of M_A to attest that it has exclusive access to the driver module M_D and that M_D also has exclusive access to its I/O device D.*	Without these attestation and exclusive access capabilities, it could not be guaranteed that inputs registered and outputs produced by the drivers are genuine.
IO3	*As soon as a microcontroller is turned on, driver modules take exclusive access to their I/O devices and never release it.*	This prevents an attacker from taking control of I/O devices before the infrastructure provider after a node reset.
IO4	*No outputs are produced by output driver modules unless requested by the application module.*	Otherwise outputs could not be attributed to corresponding inputs up the chain.
IO4	*Input driver modules do not generate outputs to application modules that do not correspond to physical inputs.*	Otherwise physical outputs may be generated that are not justified by any physical inputs. There is a caveat on automatic initialization messages and other meta-data outputs of the drivers.

Table A.2.2. Requirements on Infrastructure and Deployer

Ref.	Reproduction	Comment
D1	*The channel between deployer and infrastructure provider is assumed to be secure.*	Without a secure channel between these entities, attackers could launch man in the middle attacks between drivers and the deployer's secure modules.
D2	*Before granting such access, the infrastructure provider needs to ensure the authenticity of the driver module controlling the I/O device, e.g., via attestation*	As driver modules are part of the TCB, it has to be ensured that the correct, trusted driver module is running on the node where exclusive access to a device driver has been requested.
D3	*It is assumed that the provider does not leak the driver module keys and that exclusive access to a device is reserved for the deployer until they request to release it or an agreed-upon time T has passed.*	If the keys are leaked, an attacker may take control of said device driver at will. Releasing exclusive access prematurely is more of an availability problem as connection keys distributed to the driver module are never leaked, i.e., attackers could in that case take over a device driver but not insert themselves into a previous communication session with that driver.
D4	*The infrastructure must provide replay protection for messages exchanged with the driver module in the protocol to establish exclusive driver access.*	Otherwise an attacker may repeatedly obtain exclusive access to a driver module without involving the infrastructure provider.

ACKNOWLEDGMENTS

We thank Danny Hughes and his group, specifically Tom Van Eyck, for sharing hardware and experience to extend the evaluation.

REFERENCES

[1] Martín Abadi. 1999. Protection in programming-language translations. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. Jan Vitek and Christian D. Jensen (Eds.). Springer, Chapter 2, 19–34.

[2] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *Proceedings of the 2019 IEEE 32nd Computer Security Foundations Symposium*. IEEE, 256–25615.

[3] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N. Asokan. 2018. Migrating SGX enclaves with persistent state. In *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 195–206.

[4] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. 2021. Aion: Enabling open systems through strong availability guarantees for enclaves. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Seoul, 1357–1372.

[5] Tiago Alves and Don Felton. 2004. TrustZone: Integrated hardware and software security. *Information Quarterly* 3, 4 (2004), 18–24.

[6] ARM-software. 2023. *Trusted Board Boot*. Retrieved from https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/design/trusted-board-boot.rst. Accessed 05-02-2023.

[7] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C. Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *Proceedings of the 2013 IEEE Conference on Computer Communications Workshops*. IEEE, 79–80.

[8] Manuel Barbosa, Bernardo Portela, Bogdan Warinschi, and Guillaume Scerri. 2016. Foundations of hardware-based attested computation and application to SGX. In *Proceedings of the Euro S&P*. IEEE.

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2011. Duplexing the Sponge: Single-pass authenticated encryption and other applications. In *Proceedings of the SAC*. Springer, 320–337.

[10] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. 2012. SPONGENT: The design space of lightweight cryptographic hashing. *IEEE Transactions on Computers* 62, 10 (2012), 2041–2053.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *Proceedings of the 11th USENIX WS on Offensive Technologies*.

[12] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. 2021. Securing interruptible enclaved execution on small microprocessors. *Transactions on Programming Languages and Systems* 43, 3 (2021), 1–77. DOI: https://doi.org/10.1145/3470534

[13] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. 2011. DriverGuard: A fine-grained protection on I/O flows. In *Proceedings of the ESORICS*. LNCS, Vol. 6879. Springer.

[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *IACR Cryptol. ePrint Arch.* (2016), 86. Retrieved from http://eprint.iacr.org/2016/086. Accessed 05-02-2023.

[15] Janis Danisevskis. 2023. *Android Protected Confirmation: Taking Transaction Security to the Next Level*. Retrieved from https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html. Accessed 05-02-2023.

[16] Brittany D. Davis, Janelle C. Mason, and Mohd Anwar. 2020. Vulnerability studies and security postures of IoT devices: A smart home case study. *IEEE Internet of Things Journal* 7, 10 (2020), 10102–10110.

[17] Debian. 2023. *Debian 11 (Bullseye) Statistics*. Retrieved from https://sources.debian.org/stats/bullseye/. Accessed 05-02-2023.

[18] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, et al. 2022. StrongBox: A GPU TEE on arm endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 769–783.

[19] D. Dolev and A. C. Yao. 1981. On the security of public key protocols. In *Proceedings of the SFCS*. IEEE, 350–357.

[20] Enarx. 2023. *Enarx: Confidential Computing with WebAssembly*. Retrieved from https://github.com/enarx/enarx/. Accessed 05-02-2023.

[21] Saba Eskandarian, Jonathan Cogan, Sawyer Birnbaum, Peh Chang Wei Brandon, Dillon Franke, Forest Fraser, Gaspar Garcia, Eric Gong, Hung T. Nguyen, Taresh K. Sethi, Vishal Subbiah, Michael Backes, Giancarlo Pellegrino, and Dan Boneh. 2019. Fidelius: Protecting user secrets from compromised browsers. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. 264–280. DOI: https://doi.org/10.1109/SP.2019.00036

[22] FortanixEDP. 2023. *Fortanix Enclave Development Platform*. Retrieved from https://edp.fortanix.com/. Accessed 05-02-2023.
[23] Google. 2023. Google Asylo. Retrieved from https://asylo.dev/. Accessed 05-02-2023.
[24] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.
[25] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the Symp. S&P*. IEEE.
[26] Intel. 2023. *Intel Trust Domain Extensions*. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. Accessed 05-02-2023.
[27] Intel. 2023. Intel TDX Connect Architecture Specification. Retrieved from https://www.intel.com/content/www/us/en/content-details/773614/intel-tdx-connect-architecture-specification.html. Accessed 05-02-2023.
[28] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 455–468.
[29] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel software guard extensions: EPID provisioning and attestation services. *White Paper* 1, 1–10 (2016), 119.
[30] David Kaplan, Jeremy Powell, and Tom Woller. 2016. *AMD Memory Encryption*. Technical Report. https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf. Accessed 05-02-2023.
[31] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 207–220.
[32] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the 15th European Conference on Computer Systems*. 1–16.
[33] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. 2015. Adattester: Secure online mobile advertisement attestation using trustzone. In *Proceedings of the 13th Annual International Conf. on Mobile Systems, Applications, and Services*. 75–88.
[34] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Vbutton: Practical attestation of user-driven operations in mobile apps. In *Proceedings of the 16th Annual International Conf. on Mobile Systems, Applications, and Services*. 28–40.
[35] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. 2014. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific WS on Systems*. 1–7. DOI: https://doi.org/10.1145/2637166.2637225
[36] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and verification of the arm confidential compute architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. 465–484.
[37] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. 2020. Establishing trusted I/O paths for SGX client systems with aurora. *IEEE Transactions on Information Forensics and Security* 15 (2020), 1589–1600. https://ieeexplore.ieee.org/document/8859293.
[38] Linaro. 2023. *OP-TEE Documentation*. Retrieved from https://optee.readthedocs.io/. Accessed 05-02-2023.
[39] Dongtao Liu and Landon P. Cox. 2014. Veriui: Attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. 1–6.
[40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of the Symp. S&P*. IEEE, 605–622.
[41] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix C. Freiling, and Ingrid M. R. Verbauwhede. 2018. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers* 67, 9 (2018), 361–374.
[42] Edgeless Systems. 2023. Marblerun: The control plane for confidential computing. https://marblerun.sh/. Accessed 05-02-2023.
[43] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the Eurosys*. ACM, 315–328.
[44] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. 2006. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of the ATEC*. USENIX.
[45] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. 2009. Safe passage for passwords and other sensitive data. In *Proceedings of the NDSS*.
[46] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the HASP*. ACM, 10:1–10:1.

[47] Jan Tobias Mühlberg, Sara Cleemput, A. Mustafa Mustafa, Jo Van Bulck, Bart Preneel, and Frank Piessens. 2016. An implementation of a high assurance smart meter using protected module architectures. In *Proceedings of the WISTP'16*. Springer, 53–69.

[48] Jan Tobias Mühlberg and Jo Van Bulck. 2018. Tutorial: Building distributed enclave applications with sancus and SGX. In *Proceedings of the DSN'18*. IEEE.

[49] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. TrustZone explained: Architectural features and use cases. *2016 IEEE 2nd International Conference on Collaboration and Internet Computing* (2016), 445–451. https://ieeexplore.ieee.org/document/7809736.

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

[51] Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2017. Authentic execution of distributed event-driven applications with a small TCB. In *Proceedings of the STM'17*. Springer.

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

[53] NVIDIA. 2023. *NVIDIA H100 Tensor Core GPU Architecture*. Technical Report. Retrieved from https://resources.nvidia.com/en-us-tensor-core. Accessed 05-02-2023.

[54] Open Enclave. 2023. Open Enclave SDK. Retrieved from https://openenclave.io/sdk/.

[55] Travis Peters, Reshma Lal, Srikanth Varadarajan, Pradeep Pappachan, and David Kotz. 2018. BASTION-SGX: Bluetooth and architectural support for trusted I/O on SGX. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. Jakub Szefer, Weidong Shi, and Ruby B. Lee (Eds.), ACM, 3:1–3:9. DOI: https://doi.org/10.1145/3214292.3214295

[56] Sandro Pinto and Nuno Santos. 2019. Demystifying arm TrustZone: A comprehensive survey. *ACM Computing Surveys* 51, 6 (2019), 1–36.

[57] Barath Raghavan, Bonnie Nardi, Sarah T. Lovell, Juliet Norton, Bill Tomlinson, and Donald J. Patterson. 2016. Computational agroecology: Sustainable food ecosystem design. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, San Jose, CA, 423–435. DOI: https://doi.org/10.1145/2851581.2892577

[58] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Löser, Dennis Mattoon, Magnus Nyström, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. 2016. fTPM: A software-only implementation of a TPM chip. In *Proceedings of the USENIX Security Symposium*.

[59] Gianluca Scopelliti. 2020. *Securing Smart Environments with Authentic Execution*. Master's Thesis. Politecnico Di Torino. Retrieved from https://distrinet.cs.kuleuven.be/software/sancus/publications/scopelliti2020.pdf. Accessed 05-02-2023.

[60] Fatemeh Shirazi, Milivoj Simeonovski, Muhammad Rizwan Asghar, Michael Backes, and Claudia Diaz. 2018. A survey on routing in anonymous communication protocols. *ACM Computing Surveys* 51, 3 (2018), 39 pages.

[61] Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, and Frank Piessens. 2013. Protected software module architectures. In *Proceedings of the ISSE 2013 Securing Electronic Business Processes*. Springer, 241–251.

[62] Adam Streed, Michael Kantar, Bill Tomlinson, and Barath Raghavan. 2021. How Sustainable is the smart farm? *Workshop on Computing within Limits* (2021). DOI: https://doi.org/10.21428/bf6fb269.f2d0adaf

[63] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. TrustOTP: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 976–988.

[64] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *Proceedings of the 27th USENIX Security Symposium*. 991–1008.

[65] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. 2017. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the ACSAC'17*. ACM, New York, 225–237.

[66] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 178–195.

[67] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX Fails in Practice. Retrieved from https://sgaxe.com/files/SGAxe.pdf. Accessed 05-02-2023.

[68] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted execution environments on GPUs. In *Proceedings of the OSDI*. 681–696.

[69] Samuel Weiser and Mario Werner. 2017. SGXIO: Generic trusted I/O path for intel SGX. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. Association for Computing Machinery, 261–268. https://doi.org/10.1145/3029806.3029822

[70] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. Technical Report. https://foreshadowattack.eu/foreshadow-NG.pdf. Accessed 05-02-2023.

[71] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. 2021. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. In *Proceedings of the EuroS&P'21*. IEEE, Washington, DC, 667–682. DOI: https://doi.org/10.1109/EuroSP51992.2021.00050

[72] Jang Yeong Jin. 2017. *Building Trust in the User I/O in Computer Systems*. Ph. D. Dissertation. Georgia Institute of Technology.

[73] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. 2018. Truz-droid: Integrating trustzone with mobile operating system. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 14–27.

[74] Kailiang Ying, Priyank Thavai, and Wenliang Du. 2019. Truz-view: Developing trustzone user interface for mobile os using delegation integration model. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy*. 1–12.

[75] Marcus Peinado Yuanzhong Xu, Weidong Cui. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the Symp. S&P*. IEEE.

[76] Haofan Zheng and Owen Arden. 2021. Secure distributed applications the decent way. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*. 29–42.

[77] Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, and Dan Meng. 2016. TrustPAY: Trusted mobile payment on security enhanced ARM TrustZone platforms. In *Proceedings of the 2016 IEEE Symposium on Computers and Communications*. IEEE, 456–462.

[78] Zongwei Zhou, Virgil Gligor, James Newsome, and Jonathan McCune. 2012. Building verifiable trusted path on commodity x86 computers. *Proceedings of the IEEE Symposium on Security and Privacy*. 616–630. DOI: https://doi.org/10.1109/SP.2012.42

[79] Zongwei Zhou, Miao Yu, and Virgil Gligor. 2015. Dancing with giants: Wimpy kernels for on-demand I/O isolation. *Proceedings of the IEEE Symposium on Security and Privacy* 13, 03 (2015), 38–46. DOI: https://doi.org/10.1109/MSP.2015.26

[80] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, et al. 2020. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*. IEEE, 1450–1465.

Received 2 June 2022; revised 10 February 2023; accepted 3 April 2023