Authentic Execution of Distributed Event-Driven Applications with a Small TCB

Job Noorman, Jan Tobias Mühlberg and Frank Piessens

imec-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium

Abstract. This paper 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 a Protected Module Architecture (PMA) to guarantee authenticity and integrity properties of applications, and to secure control of input and output devices used by these applications. More specifically, we want to guarantee that *if* an output is produced by the application, it was allowed to be produced by the application's source code. We present a prototype implementation as an extension of Sancus, a light-weight embedded PMA that extends the TI MSP430 CPU. Our evaluation of the security and performance aspects of our approach and the prototype show that PMAs together with our programming model form a basis for powerful security architectures for dependable systems in domains such as Industrial Control Systems, the Internet of Things or Wireless Sensor Networks.

1 Introduction

This paper studies the problem of securely executing distributed applications on a shared infrastructure with a small 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, because for this security property and class of applications, it is relatively easy to specify the exact security guarantees offered by our approach. But we believe our approach to be valuable for any kind of distributed application (event-driven or not). In particular, our prototype supports arbitrary C code for building distributed applications.

The approach discussed here has been experimented with in previous work, where a secure smart metering infrastructure [12] is built, which we build upon, generalize and partly formalize. 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. Let us elaborate this.

First, it is clear that authentic execution gives *no* availability guarantees: if the execution never produces any output, then it is vacuously secure. Extending our approach with availability guarantees is a challenging direction for future work.

Second, while our *implementation* does offer confidentiality, this is not the focus of this paper: We specify authentic execution *without* confidentiality guarantees, i.e., attackers can observe events in the system. Indeed, securing applications in domains such as safety-critical control systems requires authenticity while confidentiality is typically not desired so as to simplify system monitoring and forensics. Third, authentic execution *does* provide strong integrity guarantees: it rules out both spoofed events as well as tampering with the execution of the program. Informally, if the executing program produces an output event, it must also have produced that same event if no attacker was present. Any physical output event can be explained by means of the untampered code of the application, and the actual physical input events that have happened.

The main contributions of this paper are: *(1)* The design of an approach for authentic execution of event-driven programs under the assumption that the execution infrastructure offers specific security primitives – standard Protected Modules (PMs) [16] plus support for secure I/O (Sect. 3). *(2)* A novel technique for implementing such support for secure I/O by means of protected driver modules on small microprocessors such as the MSP430 (Sect. 4). *(3)* A prototype implementation of the approach for an MSP430 microprocessor where all security primitives are implemented in hardware, which results in a very small TCB (Sect. 4). *(4)* An evaluation of the performance and security aspects of that implementation (Sect. 5). Our complete implementation and all supplementary materials, including a formalization and proof sketch of our security guarantees, are available at `https://people.cs.kuleuven.be/~jantobias.muehlberg/stm17/`.

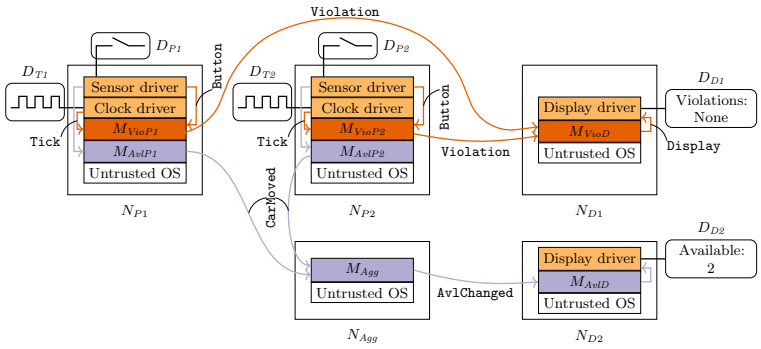

Fig. 1. Our running example with two applications, A_{Avl} (purple) and A_{Vio} (dark orange). Hardware (N_* and D_*) is trusted; the OS as well as the network are untrusted. E.g., the A_{Vio} deployer creates the three red PMs (cf. Fig. 2a) with a trusted compiler, attests the shared parking sensor-, clock- and display drivers and sets-up connections between the PMs. Remote attestation assures authentic execution of A_{Avl} and A_{Vio}.

2 Running Example, Infrastructure & Objectives

Fig. 1 (source code in Fig. 2) shows a running example for the kind of system we consider: a sensor network on a parking lot with two parking spots. The

infrastructure can be reused for multiple applications which can be provided by different stakeholders. Applications include parking guidance, parking lot utilization analysis, or detection of cars that violate parking rules. We show two of these applications: one (A_{Vio}) that detects and displays parking violations, and another (A_{Avl}) that displays the number of available parking spots.

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 MSP430 used in our prototype.

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

In our running example, there are 5 nodes. Two of these (N_{P1} and N_{P2}) are each attached to two input devices (a clock D_{Ti} and a car presence detector D_{Pi}), and are installed on parking spots. Two other nodes (N_{D1} and N_{D2}) are connected to display devices (D_{Di}) and show the output of the applications. One node (N_{Agg}) is not connected to any I/O device but performs general purpose computation, e.g., aggregating data from multiple sensor nodes.

```
module VioP1
on Button(pressed):
  if pressed: taken = 1
  else:
    taken = 0
    count = 0
    Violation(0)
on Tick():
  if taken: count = count + 1
  if count > MAX: Violation(1)
module VioP2
# Similar to VioP1
module VioD
on Violation1(violated)
  v1 = violated
  if v1: Display(1)
  if v2: Display(2)
on Violation2(violated):
  # Similar to Violation1
```

(a) A_{Vio}

```
module AvlP1
on Button(pressed):
  CarMoved(pressed)
module AvlP2
# Similar to AvlP1

module Agg
on CarMoved1(entered):
  p1 = entered
  num_avl = NUM_PARKINGS
  if (p1): num_avl = num_avl - 1
  if (p2): num_avl = num_avl - 1
  AvlChanged(num_avl)
on CarMoved2(entered):
  # Similar to CarMoved1

module AvlD
on AvlChanged(num_avl)
  Display(num_avl)
```

(b) A_{Avl}

Fig. 2. Source of the applications from Fig. 1. PMs are declared using the `module` keyword and span until the next `module` or the end of the file. `on` starts an event handler which can connect to an output of another PM, or to a physical I/O channel. Outputs are implicitly declared when invoked through a function call-like syntax.

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, an event produced on a physical output could turn on an LED. A key contribution of this paper is a way to securely connect modules to physical I/O channels (Sect. 4).

In our example applications (Fig. 1), A_{Vio} consists of three modules: two (M_{VioP1} and M_{VioP2}) are deployed on parking spots and detect single violations and one (M_{VioD}) aggregates and displays all violations. The two parking spot modules have two inputs that are connected to input devices provided by the infrastructure: one that produces events for cars entering and exiting the parking spot (D_{Pi}) and another that sends periodical timer events (D_{Ti}). As the source code (Fig. 2a) shows, these modules wait for a car enter event, then for a maximum number of timer events and then produce an output event to indicate a violation. These output events are connected to the inputs of M_{VioD} which in turn produces output events for all violations and sends them to the output display D_{D1}.

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, can modify traffic, or can mount man-in-the-middle attacks. With respect to the cryptographic capabilities of the attacker, we follow the Dolev-Yao model [5].

Attacks against the hardware are out of scope. We assume the attacker not to 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. While physical protection and side-channel resistance are important, they are orthogonal and complementary to the protection offered by our approach.

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, an actual trace of physical I/O events will happen, and the deployer can observe an actual sequence of physical output events. 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.

For instance for A_{Vio}, suppose we have physical events where a car arrives on parking 1, `MAX` clock ticks pass and then the display shows a 1. The trace of outputs is an authentic trace for A_{Vio}, because its source code allows for this

display event given the physical input events. A trace for the same sequence of physical events, but now ending with the display showing a 2, is *not* authentic.

Our objective is to design a deployment algorithm such that the deployer can efficiently check authenticity of traces. If the deployer observes a trace of physical output events, and the authenticity check of the deployer succeeds, then our approach guarantees that this trace of output events is 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. As discussed earlier, there are no availability guarantees – e.g., the attacker can suppress network communication. There are also no confidentiality guarantees: the attacker is not prevented from observing events. However, although this is not the focus of our design, our implementation *does* come with substantial protection of the confidentiality of the application's state as well as the information in events (Sect. 5).

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.

Underlying Architecture: PMAs. 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 PMA [16].

While details vary between PMAs, 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 also expect the availability of a compiler that targets PMs on 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 a PM suitable for isolation on the underlying architecture.

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

After enabling isolation, the PMA is capable of establishing a confidential, integrity protected and authenticated communication channel between a PM and its deployer. Although the details of how this works may differ from one PMA to

another, for simplicity we assume the PMA establishes a shared secret between a PM 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 a PM's identity and hence to attestation. Thus, the PMA ensures that if a deployer receives a message created with a module key, it can only have been created by the corresponding, correctly isolated, PM.

Mapping source modules to PMs. To map a source module to a PM, we use the following procedure. First, each of the source module's inputs and outputs is assigned a unique *connection identifier*. The format of this identifier is unimportant as long as it uniquely specifies a particular input or output.

A table (`KeyTable`) is added to the PM's variables that maps connection identifiers to symmetric keys such that every connection has one key associated with it. These keys will be initialized to all zeros by the architecture, which is interpreted as an unconnected input or output. For establishing a connection, an entry point is generated (`SetKey`). This entry point takes a connection identifier and a key – encrypted using the module key – as input and updates the corresponding mapping in `KeyTable` if it is not already set (Fig. 3).

```
def SetKey(payload):
    try:
      conn_id, key = Decrypt(payload)
      if KeyTable[conn_id] == 0:
        KeyTable[conn_id] = key
    except: pass
```

Fig. 3. Pseudocode of the `SetKey` entry point using a Python-like syntax. 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:
    key = KeyTable[conn_id]
  if key != 0:
    cb = CbTable[conn_id]
    nonce = NonceTable[conn_id]
    cb(Decrypt(nonce, payload, key))
    NonceTable[conn_id] += 1
  except: pass
```

Fig. 4. 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` it takes a key and the expected associated data as arguments.

```
def HandleOutput(conn_id, data):
  key = KeyTable[conn_id]
  if key != 0:
    nonce = NonceTable[conn_id]
    NonceTable[conn_id] += 1
    payload = Encrypt(nonce, data, key)
    HandleLocalEvent(conn_id, payload)
```

Fig. 5. Pseudocode of the generated output wrapper. Since the compiler generates calls to this function and it cannot be called from outside the module, the connection identifier is always valid and no error checking is necessary.

Since every connection needs to be protected from reordering and replay attacks, the compiler adds another table (`NonceTable`) to the PM's variables. This table maps connection identifiers to the current *nonce* for each connection.

All the module's event handlers are marked as non-entry functions. A callback table (`CbTable`) is added to the PM's constants that maps connection identifiers

of inputs to the corresponding event handlers. This table is used by the entry point `HandleInput`, which is called when an event is delivered to the PM. `HandleInput` takes two arguments: a plain-text connection identifier and an encrypted payload. If `KeyTable` has a key for the given identifier it is used to decrypt the payload (using the *expected* nonce as associated data), which is then passed to the callback function stored in `CbTable`. If any of these operations fails, the event is ignored (Fig. 4). From a programmer's perspective, an input callback is only called for events that were generated by entities with access to valid connection keys.

Each call to an output is replaced by a call to `HandleOutput`. This function takes a connection identifier and a payload, encrypts the payload together with the current connection nonce (which is incremented afterwards) using the corresponding connection key and publishes it to the event manager (via `HandleLocalEvent`), passing it the connection identifier. If the output is currently unconnected, the output event will be dropped (Fig. 5).

To conclude, the following PM definition will be given as input to the PMA compiler: *(1)* `SetKey` and `HandleInput` as entry points; *(2)* input event handlers and `HandleOutput` as non-entry functions; *(3)* `KeyTable`, `NonceTable`, and module-global variables; and *(4)* `CbTable` and module constants as constants. Fig. 6 shows the compiled memory layout of one of the example modules.

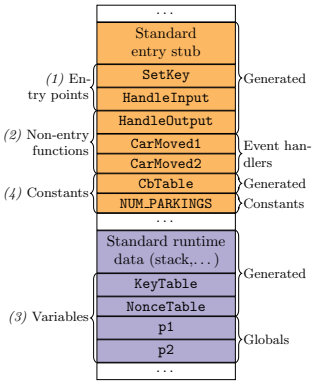

Fig. 6. Memory layout of the compiled version of M_{Agg} of A_{Avl} (Fig. 2b). The code- and data sections are shaded in light orange and purple, respectively. The numbers on the left labels correspond to the compiler inputs while the right labels indicate whether parts are implicitly generated by the compiler or correspond to source code.

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 the nodes, as outlined here.

Module Loader. The module loader is an untrusted software component running on every node. It listens for two types of remote requests: `LoadModule` and `CallEntry`. `LoadModule` takes a compiled PM as input, loads it into the PMA 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 is performed is specific to the used PMA. `CallEntry` takes a PM's identifier, the identifier of an entry point and potentially some arguments and calls the entry point with the given arguments.

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)* a PM 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.

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. 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 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. The infrastructure must also provide a way for the deployer of M_A to attest that it has exclusive access to M_D and that M_D also has exclusive access to 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.

Deployment. Deployment is the act of installing all application modules on their nodes and setting up the connections between outputs and inputs. All computations described in this section are run on the deployer's infrastructure and are therefore trusted. Communication is performed over an untrusted network.

In phase 1, the deployer starts by compiling each source module into a loadable image. Then, 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 PMA-specific method of attestation and 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.

To complete phase 1, the deployer sets up the connections between modules (not yet the connections to physical I/O channels). The deployer will generate a unique connection key and send it to both endpoints of the connection. Sending the key to a PM is done by first encrypting it together with the connection

identifier using the module key. This payload is then passed to the `SetKey` entry point using the module loader's `CallEntry` API.

Next (phase 2) the deployer sets up the connections to the physical I/O channels. The deployer first sets up the connection to physical outputs (phase 2a). This is the point in time from which we know that outputs will be authentic. Finally, all connections to physical inputs are set up (phase 2b).

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 2a of deployment (Sect. 3), and ending at a point where the deployer starts an attestation of a specific protected driver module for an output device D_O. If this attestation succeeds, and if the deployer has observed a specific sequence of physical output events on 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_{Vio} (Fig. 2a). If, after the application has been deployed, a "1" appears on the display, and if attestation confirms that the driver of the display is still in the expected state, then there must have been physical input events of a car arriving on parking spot 1 and `MAX` clock ticks.

Since output events can only be produced by the application's PMs; the assumption of a correct compiler then leads to the desired property. Because *(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)* a PM (M_O) has exclusive access to the driver; only M_O can initiate physical outputs on D_O. The successful attestation of the output driver module after the outputs have been observed ensures that exclusive access was maintained over the entire considered time frame. The construction of PMs ensures that a module can only be invoked through its two entry points. Of these, only `HandleInput` can result in output events (Figs. 3 and 4). Since `HandleInput` authenticates its input, output events are always the result of correct input events. Since our deployment scheme only allows for two types of correct input events, physical input events and outputs from other modules, our security property follows.

4 Implementation

We have created a fully functional prototype of our design based on the hardware-only PMA Sancus [14], which we briefly introduce here. We provide both the necessary compiler extensions to compile source modules to Sancus PMs and the runtime components to deploy applications on Contiki [6] based networks. A novelty of our work are *driver PMs* that facilitate secure I/O.

Sancus. Sancus [14] is an MSP430-based PMAs, designed for low-cost and low-power embedded applications. As described in Sect. 3, Sancus divides PMs in two sections called the *public code section* and the *private data section*, and enforces strict access rules. A PM'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.

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 owner of the node, the *infrastructure provider*. Every vendor who is to install PMs 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 a PMs *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 all authentication and attestation purposes. 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 PM, it is guaranteed that the *use* of a certain module key (e.g., by creating a MAC) functions as attestation of the module's identity.

This scheme works well for remote attestation but can be made more efficient for *local attestation*, i.e., the verification of a module's identity by another module running on the same node (called *secure linking* in Sancus terminology). Sancus provides an instruction that verifies the identity of a module wrt. a given MAC. A MAC, instead of a simple hash, is used to be able to store it in unprotected memory and is generated with the key of the module calling the attestation instruction. Unlike remote attestation, local attestation allows modules from different vendors to attest each other. Sancus assigns an integer *module identifier* to every PM, which is unique within a boot cycle. This can be used to speed-up local attestation by calling the attestation instruction only once.

Sancus defines secure communication as authenticated and integrity protected data exchange and provides modules with an instruction that uses the calling module's key to produce a MAC of some data. Modules can use this instruction to communicate securely with their vendor. Sancus' crypto engine uses HMAC with SPONGENT [4] as the underlying hash function to calculate MACs.

As our design requires the ability to create a *confidential* communication channel between a module and its vendor (Sect. 3), this engine did not suffice. We replaced the HMAC implementation with SpongeWrap [3] – an authenticated encryption with associated data mode. Our implementation can be configured to provide between 64 and 376 bits of security. 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.

Secure I/O on Sancus. 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 (Sect. 3). 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 MSP430 architecture, uses Memory-Mapped I/O (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 MMIO region. There is one difficulty, however, caused by the private section of Sancus modules being contiguous and the MSP430 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 in 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 and exclusive access to the device's MMIO registers. On failure, `mod-driver` aborts as well.

Sancus did not support caller authentication [14], 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 PM identity.

Secure Interrupts. On the MSP430, interrupt handlers are registered by writing their address to the interrupt vector, a specific memory location. Thus, handling interrupts inside PMs is done by registering a module's entry point as an interrupt handler. However, if the PM 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 PM.

Interfacing with Applications. One possibility to interface driver PMs with application PMs uses Sancus' secure linking feature (Sect. 4) for efficiency. The downside of our approach is that the application module has to be deployed on the same node as the driver. For simplicity, we discuss drivers for single physical events but the described techniques can easily be extended to drivers supporting multiple such events. Input drivers provide an entry point to register a callback function to be invoked when a physical event happens (`RegisterInputCb`). During the deployment phase, application modules call this entry point – using Sancus' secure linking feature – to register one of their entry points as a callback. The driver's identifier, which is the result of a successful secure linking step, is stored in the modules private data section. When a module's callback entry point is called, this identifier is compared with the result of the `CallerId` instruction to verify it was called by the expected driver.

Output drivers provide an entry point that allows modules to gain exclusive access (`AcquireOutput`). When called, this entry point checks if some module already has exclusive access and, if not, uses `CallerId` to store the identifier of the requesting module. It also offers an entry point for posting events which will check, again using `CallerId`, if the module posting the event has exclusive access. During deployment, an application module first attests the output driver, storing its module identifier, and then calls `AcquireOutput`, aborting on failure. For attestation, the application modules provides an entry point for the deployer that attests that the module has exclusive access. This is implemented by comparing the driver PM's current module identifier with that of the module located at the location where the driver module was loaded at deployment time.

The reason this attestation procedure is secure is as follows. When an application module (M_A) attests a driver module (M_D) during deployment, M_A checks the correctness of M_D's code. This includes, among others, that the code only allows a single module to have access to the driver, and that it does not release this access without the module having exclusive access asking for it. If M_A records the module identifier of M_D after having attested it, M_A can later check if M_D still exists by simply validating the identifier of the PM loaded at the location where M_D was loaded during deployment. This works because Sancus ensures module identifiers are unique within a boot cycle. If M_A calls `AcquireOutput` on M_D and it succeeds, and later verifies that M_D still exists, M_A can be sure it still has exclusive access to M_D. This procedure also ensures that M_D has exclusive access to its underlying device.

Compiler and Untrusted Runtime. Our compiler implementation is a literal translation of the design outlined in Sect. 3. 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 PMs). On top of the existing annotations provided by the Sancus compiler for specifying entry points (`SM_ENTRY`), internal functions (`SM_FUNC`) and private data (`SM_DATA`), we added two new annotations: `SM_INPUT` and `SM_OUTPUT` for specifying inputs and outputs. Fig. 7 shows an example module written in C using our annotations.

```
SM_OUTPUT(Violation);
SM_DATA int taken, count;
SM_INPUT(Button, data, len) {
 if (data[0]) {
   taken = 1;
 } else {
   taken = count = 0;
   char event = 0;
   Violation(&event, sizeof(event)); } }
SM_INPUT(Tick, data, len) {
 if (taken && ++count > MAX) {
   char event = 1;
   Violation(&event, sizeof(event)); } }
```

Fig. 7. A translation of module M_{VioP1} (Figs. 1 and 2a) to C using the annotations understood by our compiler.

`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: `void name(char* data, size_t len)`. This function can be called to produce an output event. For input handlers, `SM_INPUT` generates a functions 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. The untrusted runtime consists the module loader and the event manager (Sect. 3), both running as regular Contiki [6] processes.

5 Evaluation & Discussion

To assess the runtime overhead and the size of the software TCB we have implemented and deployed A_{Avl} (Figs. 1 and 2b). Each node is configured to provide 64 bits of security. We install Contiki, our module loader and event manager, PM drivers for I/O devices (a button driver for the car sensors and a serial LCD driver for the display), and the application modules.

Component	Src (LOC)	Bin (B)
Contiki	38386	14880
Event manager	598	1730
Module loader	906	1959
Buttons Driver	338	1016
LCD Driver	137	640
Parking Sensor	43	1383
Aggregator	84	1970
Display	31	1333
Deployment Descriptor	57	n/a

Table 1. Size ("Src": source code, "Bin": binary size) of the software for running the evaluation scenario. The shaded components are part of the run-time software TCB.

Tab. 1 shows the sizes of the different software components deployed on nodes. As can be seen, the majority of the code – about 40 kLOC – is untrusted. A total of 633 LOC comprising of drivers and the actual application code is compiled to PMs and needs to be trusted, together with 57 LOC

of the deployment descriptor. That is, only 1.7% of the deployed code base is part of the software TCB. When looking at the binary sizes of the these software components, the difference between infrastructure components (18.1 KiB) versus TCB (6.2 KiB, 25.5%) appears less prominent, which is mostly due to conditionally compiled code in Contiki and compiler generated entry points and stub code in PMs. Nevertheless, the reduction of the TCB when using our approach is substantial, leading to a considerably reduced attack surface on each node, and – importantly – the application owner does not need to trust *any* infrastructural software if he reviews the driver modules that his application uses.

We also performed a detailed performance analysis of this example application, the detailed results are given online in the supplements. For fast devices such as the button sensors, the overhead of our secure I/O approach can be quite large: the protected driver executes about 13 times slower than the unprotected one. However, not much effort was put in optimizing our implementation and we expect that significant performance gains are possible (e.g., the wrapper for encrypting output events uses `malloc` to create a buffer contributing about $30\mu s$ to the overhead). For slower devices such as the serial LCD, it is clear that the relative overhead drops significantly: the protected driver executes about 7% slower. Another type of overhead is due to the increased size of events. The sequence diagram shows that an event containing 2 bytes of useful data, will be 6 times as large. In general, the representation of events has a constant overhead of 10 bytes: 2 for the nonce and 8 for the MAC. Whether these overheads are acceptable will depend on the application. Yet, we feel that they are reasonable in the light of the security guarantees and TCB reduction provided by our approach.

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.

Hardware Attacks and Side-Channels. Although hardware attacks and side-channels are explicitly ruled out by our attacker model (Sect. 2), it is necessary to discuss the impact an attacker would have given access to such techniques.

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, the impact may be minor.

Given the kind of small microprocessors that we target, many side-channels such as cache timing attacks or page fault channels are not applicable. We leave an analysis of our implementation for side-channels for future work.

6 Related Work

A survey of hardware-based trusted computing architectures for isolation and attestation has been published in [8], and describes a number of platforms our approach could use. These range from Intel SGX over ARM's TrustZone to embedded architectures such as SMART and TyTAN. Notably, Sancus [14] is the only available embedded PMA, and the only open-source PMA overall. The secure compilation techniques for PMAs were proposed by Agten et al. [2].

Earlier techniques to establish a notion of trusted I/O paths are *BitE* [10], *Flicker* [9] and *Bumpy* [11]. Our approach improves over these by significantly reducing the size of the software TCB, from a full OS to less than 1 kLOC. By using Sancus as a PMA, we enable the integration of attestable software and I/O encryption directly into the input device. Techniques such as Flicker or SGX can be used to protect host software when communicating with Sancus nodes.

The VC3 system [15] is related to our work in the sense that they also provide strong security guarantees to the deployer of an application using SGX as a PMA, but they focus on correctness, confidentiality and completeness of Map-Reduce computations in a cloud-setting and hence do not need to deal with I/O.

The safe and secure deployment and use of devices in the domains of Wireless Sensor Networks (WSNs) and the IoT remains an open challenge. A number of schemes for distributed trust management for WSNs are surveyed in [7], which allow individual nodes to obtain trust values for neighboring nodes by observing these nodes' behavior. In [13], Sancus is used to securely inspect and assess the trustworthiness of unprotected software on WSN and IoT nodes directly. This kind of trust management is suitable to detect the systematic failure or misbehavior of nodes, but there are no inherent guarantees wrt. the authenticity of distributed computations being provided. We address this shortcoming by protecting all components of a distributed application throughout their life-cycle.

7 Conclusions

We have extended Sancus, a light-weight embedded protected module architecture with support for secure I/O, 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 foresee compelling use cases in IoT and control systems.

ACKs. This research is partially funded by the Research Fund KU Leuven.

References

1. Abadi, M. Protection in programming-language translations. In *Secure Internet Prog., Security Issues for Mobile and Distributed Objects*, pp. 19–34. Springer, 1999.
2. Agten, P., Strackx, R., Jacobs, B., and Piessens, F. Secure compilation to modern processors. In *CSF*, pp. 171–185. IEEE, 2012.
3. Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. Duplexing the Sponge: Single-pass authenticated encryption and other applications. In *SAC*, pp. 320–337. Springer, 2011.
4. Bogdanov, A., Knezevic, M., Leander, G., Toz, D., Varici, K., and Verbauwhede, I. SPONGENT: The design space of lightweight cryptographic hashing. *IEEE Transactions on Computers*, 99(PrePrints):1, 2012.
5. Dolev, D. and Yao, A. C. On the security of public key protocols. In *SFCS*, pp. 350–357. IEEE, 1981.
6. Dunkels, A., Gronvall, B., and Voigt, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Comp. Net.*, pp. 455–462. IEEE, 2004.
7. Fernandez-Gago, M., Roman, R., and Lopez, J. A survey on the applicability of trust management systems for wireless sensor networks. In *SECPerU*, pp. 25–30, 2007.
8. Maene, P., Götzfried, J., de Clercq, R., Müller, T., Freiling, F., and Verbauwhede, I. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, (99), 2017.
9. McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An execution infrastructure for TCB minimization. In *Eurosys*, pp. 315–328. ACM, 2008.
10. McCune, J. M., Perrig, A., and Reiter, M. K. Bump in the ether: A framework for securing sensitive user input. In *ATEC*. USENIX, 2006.
11. McCune, J. M., Perrig, A., and Reiter, M. K. Safe passage for passwords and other sensitive data. In *NDSS*, 2009.
12. Mühlberg, J. T., Cleemput, S., Mustafa, M. A., Bulck, J. V., Preneel, B., and Piessens, F. An implementation of a high assurance smart meter using protected module architectures. In *WISTP '16*, vol. 9895 of *LNCS*, pp. 53–69. Springer, 2016.
13. Mühlberg, J. T., Noorman, J., and Piessens, F. Lightweight and flexible trust assessment modules for the Internet of Things. In *ESORICS*, vol. 9326 of *LNCS*, pp. 503–520. Springer, 2015.
14. Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Sec. Symp.*, pp. 479–494. USENIX, 2013.
15. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. VC3: trustworthy data analytics in the cloud using SGX. In *Symp. S&P*, pp. 38–54. IEEE, 2015.
16. Strackx, R., Noorman, J., Verbauwhede, I., Preneel, B., and Piessens, F. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pp. 241–251. Springer, 2013.