— **Supplementary Materials** —

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

Formal definition and proof of security properties, detailed performance evaluation and detailed related work

Job Noorman, Jan Tobias Mühlberg and Frank Piessens

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

In "Authentic Execution of Distributed Event-Driven Applications with a Small TCB" [15] we 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.

In this addendum to the paper we focus on defining a simple reactive programming language for our architecture, the formal semantics of that programming language, and the formal definition and proof of the security properties of our approach (Sects. 1 to 3). We further provide background information on our performance evaluation (Sect. 4) and a more complete discussion of the related work (Sect. 5), as the version in the paper is rather brief.

1 Formal Definition of our Security Properties

1.1 Programs

For the purpose of this paper, we consider event-driven programs. To precisely define our security property, we define a simple formal reactive programming language, the syntax of which shown in Figure 1. We emphasize that our implementation supports the full C language for writing event handlers.

$$
\begin{array}{rl}
m ::= & \text{module } id(id*; id*); h* \\
h ::= & \text{on } id(x)\ \{c\} \\
c ::= & \text{skip} \\
 & \mid c; c \\
 & \mid g := e \\
 & \mid \text{if } e \text{ then } \{c_1\} \text{ else } \{c_2\} \\
 & \mid \text{while } e\ \{c\} \\
 & \mid id(e)
\end{array}
\qquad
\begin{array}{rl}
e ::= & x \mid n \mid g \mid e \odot e \\
\odot ::= & + \mid - \mid = \mid < \\
cn ::= & id \to id \\
p ::= & m*; cn*; \\
o ::= & !id(n) \mid \cdot \\
i ::= & ?id(n) \\
\alpha ::= & o \mid i
\end{array}
$$

Fig. 1. Syntax of our event-driven language.

A *source module* m declares a name, zero or more identifiers for input channels, and zero or more identifiers for output channels. For each declared input channel, it then defines exactly one event handler (h) which handles events arriving on that input channel. Events carry data (in our simple language only integers), and on arrival of the event, the received integer is bound to the formal parameter x. The handler then executes its body, a command c. Commands are standard and can use assignment to module global variables g to share information between the handlers in the same module. Module global variables do not need to be declared and are initialized to 0.

A program (p) consists of zero or more modules, and zero or more connections (cn), where a connection $id_1 \to id_2$ specifies that the outputs sent on output channel id_1 should be delivered to input channel id_2. For simplicity, we assume that all input and output channels have unique names in the entire program (in practice, one would want to have module-scoped namespaces). Each output channel is connected to at most one input channel, and each input channel is connected to at most one output channel. The input and output channels that are not connected to anything are the *primitive* input and output channels of the application. We write $p(\overline{in}; \overline{out})$ to say that p is a program with primitive input channels $\overline{in}$ and primitive output channels $\overline{out}$.

Fig. 2 shows (parts of) the code of the A_{Vio} application described in [15]. The code implementing the event handler for the parking sensor on N_{P1} stores in the global variable `taken` whether there is currently a car on the parking spot. If so, the event handler for a clock tick on N_{P1} will increase a counter. If that counter exceeds MAX, it sends a violation event. To define the input and output channels of the A_{Vio} program, we write $A_{Vio}(P1, T1, P2, T2; Display)$.

The semantics of programs is as expected: execution is triggered by an event on a primitive input channel. We write such input events (i) as $?id(n)$. This leads to the execution of the corresponding handler for that event. Execution of this

```
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 code of a simple example system with two applications. The system is described in the paper.

handler generates output events o. These can be internal (unobservable, silent) events that we write as $\cdot$, or primitive output events that we write as $!id(n)$.

Below we specify the formal semantics for programs as a labelled transition system, where labels are events α, thus defining exactly what traces $\overline{\alpha}$ of events a program allows. The following is a trace of the A_{Vio} program (with `MAX == 2`, showing only primitive events):

$$?P1(1), ?T1(0), ?T1(0), ?T1(0), !Display(1)$$

After the third input event on $clock_{NP1}$, the M_{VioP1} module will send a $Violation(1)$ output that generates a $Violation1(1)$ input event on M_{VioD}, which then displays that parking spot 1 has been taken for too long. Note that the events on $Violation$ and $Violation1$ are internal events and do not show up in the trace. Note that in general, execution of programs is non-deterministic – when one handler generates output events connected to different input channels, the semantics does not impose an order in which the resulting input events must be handled.

We introduce the following notation for traces: given a trace $\overline{\alpha}$, we write $\alpha \downarrow_{\overline{id}}$ for the trace obtained from α but keeping only the events $?id_i(n)$ and $!id_i(n)$ with $id_i \in \overline{id}$.

1.2 Security Properties of Modules

The compilation and deployment scheme for our event-driven distributed applications is described in detail in [15]. We now discuss the security guarantees that this compilation and deployment of modules gives us. Consider a source module $m(\overline{in}, \overline{out})$, and the corresponding Protected Module (PM) $\lfloor m \rfloor$. The operational semantics of m defines what traces of source level events of the form $?id(n)$ and $!id(n)$ this module has. At run-time, $\lfloor m \rfloor$ will only perform the following two kinds of events:

- Output events, where the module hands a new output to the event manager. We use the notation $!(id, nc, n)$ for an event where the module calls `HandleOutput` on the connection id with nonce nc and payload n.
- Input events, where the event handler hands an event to `HandleInput` and this event passes the authenticity check. We use the notation $?(id, nc, n)$ for an event where `HandleInput` successfully accepts an event on connection id with nonce nc and payload n.

We use the notation $\lceil\rceil$ to relate run-time events to corresponding source-level events, i.e. $\lceil !(id, nc, n) \rceil = !id(n)$ and $\lceil ?(id, nc, n) \rceil = ?id(n)$.

The security guarantee that we obtain is the following. Assume that m has been correctly compiled, deployed and initialized (i.e. SetKey has been called by the deployer for every connection), resulting in $\lfloor m \rfloor$.

Lemma 1. *If $\lfloor m \rfloor$ has at run-time a trace ρ of run-time events, then m has trace $\lceil \rho \rceil$ of source-level events.*

The lemma is proven in the Section 3, but the intuition is the following. After initialization is completed, the isolation properties of the PMA ensure that the only interactions that are possible with $\lfloor m \rfloor$ are in-calls to its entry points (`HandleInput` and `SetKey`), and out-calls from within the module itself. From the construction of $\lfloor m \rfloor$ as described above, it follows that both entry points drop anything that does not pass the authenticity check, and that the only out-call the module ever performs is to `HandleLocalEvent` with a correctly constructed output message. So $\lfloor m \rfloor$ has no other interactions with its environment than the two types of events defined above. That these events correspond to a trace of source-level events allowed by the operational semantics of the module then follows from our assumption that the $\lfloor m \rfloor$ is created from the m using a correct compiler.

1.3 Security Properties of Programs

Now, consider a program $p(\overline{in}, \overline{out})$, and the deployed program $\lfloor p \rfloor$ at the end of phase 1 of deployment. Any further execution of $\lfloor p \rfloor$ (independent of whether deployment proceeds with phase 2) satisfies the following.

Lemma 2. *If $\lfloor p \rfloor$ has a run-time trace ρ, then p has a trace $\lceil \rho \rceil$.*

The proof is again in Section 3, but the intuition is the following: according to Lemma 1 each individual module behaves as its source counterpart. For the connections between modules, we know that there is a unique key unknown to the attacker per connection, and the use of nonces (sequence numbers) and authenticated encryption with this key is sufficient to turn the untrusted network in a queue: only the module that outputs on the connection can put the appropriate cryptographic messages on the network, and the module reading from the connection is assured of authenticity and correct order of receipt.

Let $p(\overline{in}, \overline{out})$ be a program with primitive inputs $\overline{in}$ and primitive outputs $\overline{out}$. Deployment connects in_i to physical input channels pi_i and out_j to physical output channels po_j. We use the notation $?pi(n)$ for the occurrence of a physical input event on pi, and $!po(n)$ for the occurrence of a physical output event on po. These physical events are the events we care about, and that can be observed or that will have effects in the real world at run-time. We extend the notation $\lceil\rceil$ in the obvious way (e.g. $\lceil ?pi(n) \rceil = ?id(n)$ when deployment connects id to pi).

Consider the time frame starting at the end of phase 2a of deployment, and ending at a point where the deployer starts a new attestation of the protected driver module for po_j. Suppose that this attestation eventually succeeds, and that the deployer has observed a sequence ρ_{out_j} of outputs on po_j. Then the following holds.

Theorem 1. *p has a trace $\overline{\alpha}$ such that (1) $\overline{\alpha}\downarrow_{out_j} = \lceil \rho_{out_j} \rceil$, and (2) for each primitive input channel $in_i \in \overline{in}$, there has been a contiguous sequence of inputs ρ_{in_i} on pi_i with $\overline{\alpha}\downarrow_{in_i} = \lceil \rho_{in_i} \rceil$.*

The proof is given in the Section 3, but the intuition is: since ρ_{out_j} is observed on an output device that was attested to only accept authenticated output events from p, there must have been an execution and hence a trace ρ of p with these output events. Lemma 2 allows us lift this to a trace of the source program. Since $\lfloor p \rfloor$ will only respond to authenticated sequences of inputs from the pi_i, there must have been such contiguous sequences of inputs.

2 Formal Semantics for our Model Language

In Section 1.1, we introduced a simple reactive programming, the syntax of which is shown in Figure 1. This section defines the operational semantics of this language using standard techniques.

2.1 Modules

We first define how modules are executed. A module configuration (μ, c) consists of a (module-private) store μ, and the currently executing command c. We use the following semantic judgements:

- $\mu \vdash e \Downarrow n$ says that the ground expression e evaluates under the variable store μ to the integer n. A *ground* expression does not contain any parameters x, but it can contain global variables g. A store μ is a mapping from global variable names g to integers n. The definition of this judgement is completely standard and hence omitted.
- $(\mu, c) \overset{\alpha}{\rightarrow} (\mu', c')$ says that module configuration (μ, c) can step to (μ', c') by performing transition α. A transition α can be an output $!id(n)$, or it can be an input $?id(n)$, i.e. the occurrence of an event on input channel id. A

transition α can also be an unobservable (silent, module-internal) action denoted by $\cdot$. For technical reasons, we use the metavariable o for outputs and silent transitions, and i for inputs.
The definition of this judgment for a specified module m is given in Figure 3, where the function lookup(m, id) looks up the body of the handler for id in the module m. We use the notation $\mu[g \mapsto n]$ for an update of the store μ that sets global variable g to n, and the notation $c[x \leftarrow n]$ for the substitution of integer n for formal parameter x in command c.

$$\frac{}{(\mu, \text{skip}; c) \xrightarrow{\cdot} (\mu, c)} \tag{1}$$

$$\frac{(\mu, c_1) \xrightarrow{o} (\mu', c_1')}{(\mu, c_1; c_2) \xrightarrow{o} (\mu', c_1'; c_2)} \tag{2}$$

$$\frac{\mu \vdash e \Downarrow n}{(\mu, g := e) \xrightarrow{\cdot} (\mu[g \mapsto n], \text{skip})} \tag{3}$$

$$\frac{c = \text{if } e \text{ then } \{c_1\} \text{ else } \{c_2\} \quad \mu \vdash e \Downarrow n \quad n \neq 0}{(\mu, c) \xrightarrow{\cdot} (\mu, c_1)} \tag{4}$$

$$\frac{c = \text{if } e \text{ then } \{c_1\} \text{ else } \{c_2\} \quad \mu \vdash e \Downarrow 0}{(\mu, c) \xrightarrow{\cdot} (\mu, c_2)} \tag{5}$$

$$\frac{c = \text{while } e\, \{c_{\text{loop}}\} \quad \mu \vdash e \Downarrow 0}{(\mu, c) \xrightarrow{\cdot} (\mu, \text{skip})} \tag{6}$$

$$\frac{c = \text{while } e\, \{c_{\text{loop}}\} \quad \mu \vdash e \Downarrow n \quad n \neq 0}{(\mu, c) \xrightarrow{\cdot} (\mu, c_{\text{loop}}; c)} \tag{7}$$

$$\frac{\mu \vdash e \Downarrow n}{(\mu, id(e)) \xrightarrow{!id(n)} (\mu, \text{skip})} \tag{8}$$

$$\frac{\text{lookup}(m, id) = c}{(\mu, \text{skip}) \xrightarrow{?id(n)} (\mu, c[x \leftarrow n])} \tag{9}$$

Fig. 3. Small-step semantics for modules.

The *initial state* of a module is (μ_0, skip) where μ_0 maps every global variable to the integer 0. We say a module m has a trace $\overline{\alpha}$ if, starting from its initial state, it can do all the transitions in $\overline{\alpha}$.

2.2 Programs

We can now turn to the definition of the semantics of programs. A program configuration $\overline{(\mu, c)}; \overline{in : q}$ consists of a sequence of module configurations $\overline{(\mu, c)}$ (discussed above), and a sequence of connection queues $\overline{in : q}$ that associate a queue of natural numbers with each input channel (this queue contains the messages that have already been sent to this input channel but not yet received from that channel).

The initial state of a program is the state: $\overline{(\mu_0, \text{skip})}, \overline{in : []}$ (i.e. an initial module state for each module in the program, and all input channels start with the empty queue). The rules in Figure 4 define the semantics of programs.

$$\frac{(\mu_i, c_i) \overset{?in_j(n)}{\rightarrow} (\mu_i', c_i')}{\overline{(\mu, c)}, (\mu_i, c_i); \overline{in : q}, in_j : q_j :: n \overset{\cdot}{\rightarrow} \overline{(\mu, c)}, (\mu_i', c_i'); \overline{in : q}, in_j : q_j} \quad \text{(E-RECV)}$$

$$\frac{(\mu_i, c_i) \overset{\cdot}{\rightarrow} (\mu_i', c_i')}{\overline{(\mu, c)}, (\mu_i, c_i); \overline{in : q} \overset{\cdot}{\rightarrow} \overline{(\mu, c)}, (\mu_i', c_i'); \overline{in : q}} \quad \text{(E-SILENT)}$$

$$\frac{(\mu_i, c_i) \overset{!id(n)}{\rightarrow} (\mu_i', c_i') \quad id \rightarrow in_j}{\overline{(\mu, c)}, (\mu_i, c_i); \overline{in : q}, in_j : q_j \overset{\cdot}{\rightarrow} \overline{(\mu, c)}, (\mu_i', c_i'); \overline{in : q}, in_j : n :: q_j} \quad \text{(E-SEND)}$$

$$\frac{\not\rightarrow in_i}{\overline{(\mu, c)}; \overline{in : q}, in_i : q_i \overset{?in_i(n)}{\rightarrow} \overline{(\mu, c)}; \overline{in : q}, in_i : n :: q_i} \quad \text{(E-INPUT)}$$

$$\frac{(\mu_i, c_i) \overset{!id(n)}{\rightarrow} (\mu_i', c_i') \qquad id \not\rightarrow}{\overline{(\mu, c)}, (\mu_i, c_i); \overline{in : q} \overset{!id(n)}{\rightarrow} \overline{(\mu, c)}, (\mu_i', c_i'); \overline{in : q}} \quad \text{(E-OUTPUT)}$$

Fig. 4. Small-step semantics for programs.

Rule (E-RECV) describes the case where one of the modules consumes an event from its (non-empty) input queue. Rule (E-SILENT) describes the case where one of the modules does an internal computation step. Rule (E-SEND) describes the case where one of the modules performs an output, and the output channel is connected to input channel in_j. Hence, the output value is put in the queue corresponding to in_i. Rule (E-INPUT) describes an input on a primitive input channel in_i (i.e. in_i is not connected to any output channel of the program). Such an input is just buffered in the appropriate queue, and no modules change state. The input will at a later point in time be removed from the queue by rule (E-RECV). Finally, rule (E-OUTPUT) describes the occurrence of a primitive output event.

We say a program p has a trace $\overline{\alpha}$ if, starting from its initial state, it can do all the transitions in $\overline{\alpha}$.

3 Proofs of Lemmas and Theorem 1.

We model PMs at run-time as states in a labeled transition system. We call the set of states S, and we denote the transition relation as $s_1 \overset{\beta}{\rightarrow} s_2$, with β a run-time event. Given the isolation properties of PMs, the only run-time interactions that are possible with a PM are (1) calls into an entry-point, and (2) out-calls from within the module to code outside.

For the PMs that we construct in the paper, there are only two entry points: `SetKey` and `HandleInput`. `SetKey` is only called once by the deployer during initialization, and has no effect on later calls. Hence, for the PMs, it suffices to consider the following run-time events:

– Output events, where the PM hands a new output to the event manager. We use the notation $!(id, nc, n)$ for an event where the module calls `HandleOutput` on the connection id with nonce nc and payload n.
– Input events, where the event handler hands an event to `HandleInput` and where this event passes the authenticity check. We use the notation $?(id, nc, n)$ for an event where `HandleInput` successfully accepts an event on connection id with nonce nc and payload n.
– Internal computation within the PM. We use the notation $\cdot$ for these silent steps.

Let $\lfloor m \rfloor$ be the initial state of the PM after compilation and deployment (including the appropriate calls to `SetKey`) of source module m. We relate run-time events to corresponding source-level events using the notation $\lceil !(id, nc, n) \rceil = !id(n)$ and $\lceil ?(id, nc, n) \rceil = ?id(n)$, and we use the same notation on traces $\lceil \overline{\beta} \rceil = \overline{\alpha}$ which maps the $\lceil\rceil$ function over the elements of the trace.

Lemma 1. *If $\lfloor m \rfloor$ has at run-time a trace $\overline{\beta}$ of run-time events, then m has trace $\lceil \overline{\beta} \rceil$ of source-level events.*

Proof: We model the fact that $\lfloor m \rfloor$ is created by a functionally correct and trusted compiler, by assuming that there exists a function (also denoted $\lceil\rceil$), that relates a run-time module state s to a source module configuration $\lceil s \rceil = (\mu, c)$, and by assuming that this function is a backward simulation. I.e. we assume that (1) $\lceil \lfloor m \rfloor \rceil = (\mu_0, \text{skip})$ (initial state of the compiled module maps to an initial source module configuration), and (2) if $s \overset{\overline{\beta}}{\rightarrow} s'$, then$\lceil s \rceil \overset{\overline{\alpha}}{\rightarrow} \lceil s' \rceil$ with $\lceil \overline{\beta} \rceil \downarrow_{\overline{in},\overline{out}} = \overline{\alpha} \downarrow_{\overline{in},\overline{out}}$ (if a run-time state s can make a number of steps to state s', then the corresponding source module configuration can also make a number of steps with corresponding observable transitions, and ending in a source module configuration corresponding to s'). From this formalization of the assumption of a correct module compiler, and from the observation that the state of PMs can not be affected in any other way than by the run-time events β (thanks to the PM), the lemma follows directly. $\square$

Next let $p(\overline{in}, \overline{out})$ be a source program. We assume all input and output channels used in the program have different identifiers, and use $mod(id)$ to denote the module that has channel id. We also assume that the deployer has chosen connection identifiers cn for each connection in the program, and we define $in(cn)$ to be the input channel identifier and $out(cn)$ to be the output channel identifier associated to connection cn.

For channels $\overline{in}$ and $\overline{out}$ that are not connected in the program (and hence that will be connected to physical inputs and outputs in the deployment descriptor), we assume that the deployer configures them with a key K_{in_i} (respectively K_{out_j}) that authenticates messages from (respectively to) a driver PM.

We model the untrusted network at run-time as two sets of messages (M, M_{all}), where M are the message currently still available on the network, and M_{all} is the set of all messages that have ever been put on the network. The attacker can remove messages from M, and can add messages to M (and hence to M_{all})

under the constraints of the Dolev-Yao model (i.e. he can not forge cryptographic messages for which he does not have the key). For the execution of p, we only care about the valid cryptographic messages on the network, i.e. all messages of the form $K_{cn}(nc, n)$ created with connection key K_{cn} (or keys K_{in_i} or K_{out_j}) and with nonce nc and payload n. Hence, we restrict the M and M_{all} to just these messages (any other messages the attacker might put on the network can not influence the execution of p).

Now, the run-time state of a program p consists of:

- The run-time states for all of its modules; and
- The state of the network.

The initial state is $\lfloor p \rfloor = (\overline{\lfloor m \rfloor}, (\{\}, \{\}))$, and run-time events are the run-time events β of each of the PMs, interleaved with attacker steps where the attacker manipulates the network as discussed above.

Given a run-time state s of one of the modules m in p, we use the notation $nonce(cn, s)$ for the value of the nonce associated with connection id cn in s.

Lemma 2. *If $\lfloor p \rfloor$ has a run-time trace $\overline{\beta}$, then p has a trace $\lceil \overline{\beta} \rceil$.*

Proof: We construct a backward simulation from run-time program states to program configurations in the source level semantics. We define $\lceil (\overline{s}, (M, M_{all})) \rceil$ to be the program configuration $(\overline{\lceil s \rceil}; \overline{id : q})$ where (1) run-time module states s are mapped on $\lceil s \rceil$ using the backward simulation on module states that we get from the correct-compiler-assumption, and (2) the q_i are defined as follows: (keep in mind that these are queues in the source level program configuration that contain the values that have been sent to some input channel in but not yet received from that input channel):

- if in is connected with connection identifier cn (i.e. there exists a cn such that $in(cn) = in$), then let nc be $nonce(cn, s)$ where s is the run-time state of $mod(in)$. Hence nc tells us how many events have already been received from that input channel. Now, construct the largest sequence of messages in M_{all} that has the following form:
$$K_{cn}(nc, n_0), K_{cn}(nc + 1, n_1), K_{cn}(nc + 2, n_2), \ldots$$
Then queue $q_i = n_0 :: n_1 :: n_2 :: \ldots$.
- if in is not connected in p then it will later be connected to physical input channel, and the deployer configures it with a key K_{in}. q_i is constructed in a similar way, but using K_{in} instead of K_{cn}.

For the remainder of the proof, we have to show that $\lceil \rceil$ is a backward simulation. This is proven by considering all possible run-time steps. We go over some of the interesting cases:

- If the attacker does a step, this only impacts M, and hence the source program configuration can stay constant to match this. (The backward simulation does not depend on M)

– Any silent step by any of the modules can be matched with zero or more (E-SILENT) steps at source level. This follows directly from the fact that $\lceil\rceil$ on PM states is a backward simulation (Lemma 1).
– If one of the run-time modules performs a $!(id, nc, n)$ event, where id is connected in the program, then a new valid cryptographic message appears on the network (in M and in M_{all}). It is easy to check that the source program can match this with an (E-SEND) step.
– If a driver PM connected to in during deployment sends a new physical input, then a new valid cryptographic message appears on the network (in M and in M_{all}). It is easy to check that the source program can match this with an (E-INPUT) step.
– If one of the run-time modules performs a $?(id, nc, n)$ event, where id is connected in the program, then the nonce in the runtime state of the receiving module will go up with 1. This will remove an element from the corresponding queue in the backward simulation. It is easy to check that the source program can match this with an (E-RECV) step.
– If one of the run-time modules performs a $!(id, nc, n)$ event, where id is not connected in the program, then a new valid cryptographic message appears on the network (in M and in M_{all}) constructed with the key K_{out} that authenticates this message to a driver PM for a physical output device. It is is easy to check that the source program can match this with an (E-OUTPUT) step.

This completes the proof. □

Finally, we turn to the security theorem. Let $p(\overline{in}, \overline{out})$ be a program with primitive inputs $\overline{in}$ and primitive outputs $\overline{out}$. Deployment connects in_i to physical input channels pi_i and out_j to physical output channels po_j.

We use the notation $?pi(n)$ for the occurrence of a physical input event on pi, and $!po(n)$ for the occurrence of a physical output event on po. We extend the notation $\lceil\rceil$ in the obvious way (e.g. $\lceil ?pi(n)\rceil = ?id(n)$ when deployment connects id to pi).

Consider the time frame starting at the end of phase 2a of deployment, and ending at a point where the deployer starts a new attestation of the driver PM for po_j. Suppose that this attestation eventually succeeds, and that the deployer has observed a sequence ρ_{out_j} of outputs on po_j. Then the following holds.

Theorem 2. *p has a trace $\overline{\alpha}$ such that (1) $\overline{\alpha}\downarrow_{out_j} = \lceil\rho_{out_j}\rceil$, and (2) for each primitive input channel $in_i \in \overline{in}$, there has been a contiguous sequence of inputs ρ_{in_i} on pi_i with $\overline{\alpha}\downarrow_{in_i} = \lceil\rho_{in_i}\rceil$.*

Proof: The driver PM for po_j was correctly deployed at the start of the time-frame, and is verified to still be correctly deployed at the end of the time-frame. It follows that it was deployed correctly for the entire time-frame, as an attacker can only destroy the state of a PM (for instance by rebooting the system), but can not redeploy the PM again afterwards.

Since po_j was in a good state the entire time-frame, the physical outputs observed on po_j must have been caused by by run-time events $!(out_j, nc, n)$ of the program. Now consider the entire trace $\overline{\beta}$ of run-time events of the program up to and including the last of these $!(out_j, nc, n)$ events.

For this trace $\overline{\beta}$, we know from Lemma 2 that p has $\overline{\alpha} = \lceil \overline{\beta} \rceil$. From the construction of $\overline{\alpha}$, it follows that $\overline{\alpha} \downarrow_{out_j} = \lceil \rho_{out_j} \rceil$.

Now, for each in_i, the corresponding PM $\lfloor mod(in_i) \rfloor$ will have performed zero or more $?(in_i, nc, n)$ events. From (1) the construction of that module, and (2) from the fact that the deployer connects it to a driver PM using a key K_{in_i} that is shared only between $mod(in_i)$ and the driver PM of pi_i, and (3) the assumption that driver PMs reliably turn po_i events into authenticated messages to in_i, we can conclude that there must have been a contiguous sequence of of inputs ρ_{in_i} on pi_i with $\overline{\alpha} \downarrow_{in_i} = \lceil \rho_{in_i} \rceil$.

$\square$

4 Control Flow and Timing

We performed a detailed performance analysis of this example application; the results of which are shown in Fig. 5. The sequence diagram corresponds to the A_{Avl} application but, for simplicity, shows a variant where there is a direct connection between the sensor module and the display module (i.e., the aggregator is ignored). The given timing information can easily be extrapolated to the complete application and, indeed, to any application.

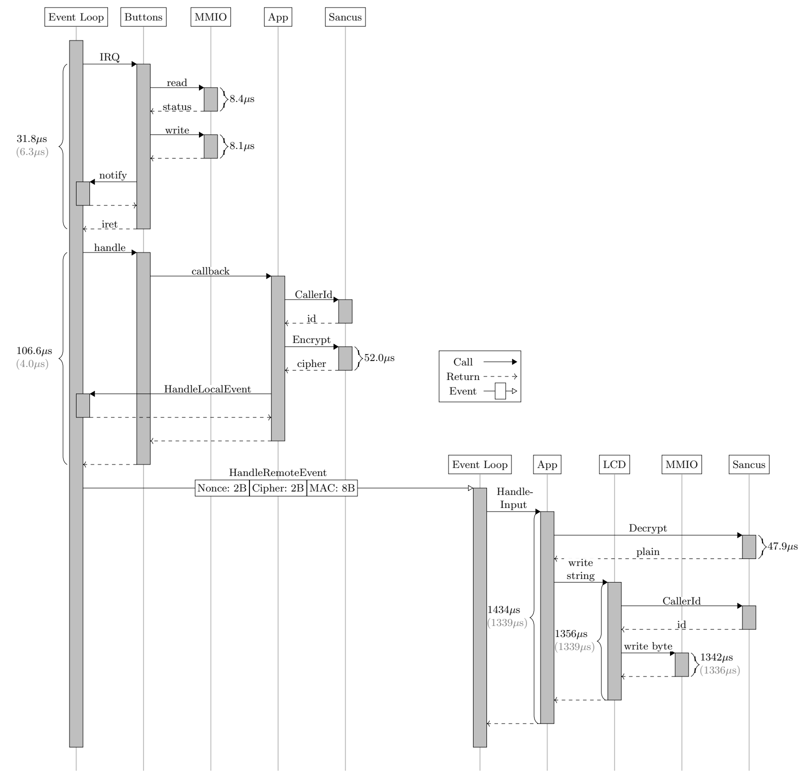

Fig. 5. Sequence diagram showing the control flow and timings of part of the A_{Avl} application in Fig. 2. The sequence diagram shows how a physical input is handled, starting from the moment an IRQ is generated by an input device until the moment a physical output is shown on an output device. The timings for the protected application are shown in black while those for an unprotected version are shown within parentheses and in grey. For the lifelines, "Event Loop" corresponds to our event manager; "Buttons" and "LCD" to the `mod-driver` part of the protected drivers (Sect. 4 of the paper; "MMIO" to the `mod-mmio` part (which is subsumed in the driver for the unprotected application); "App" to the application modules; and "Sancus" to the Sancus core (which does not exists for the unprotected application). Notice how, similar to many OSs, the "Buttons" ISR is kept as short as possible by splitting it in two levels. The first level will notify the event loop to call the second level before returning from the ISR.

5 Related Work

The safe and secure deployment and use of interconnected devices in the domains of Wireless Sensor Networks (WSNs) and the IoT remains an open challenge [16]. The problem of trustworthiness and trust management of low-power low-performance computing nodes has been discussed in previous research. In [6,7] a number of schemes for distributed trust management for WSNs are surveyed. These schemes allow individual nodes to obtain trust values for neighboring nodes by observing their externally visible behavior. Trust values are then propagated through the network, allowing nodes to make decisions based on the trustworthiness of other nodes. In [13], Sancus is used to securely inspect and assess the trustworthiness of unprotected software on WSN and IoT nodes. Trust metrics obtained by this approach can be integrated with the aforementioned trust management schemes, potentially improving the detection time and -rate for security breaches.

While this kind of trust management is suitable to detect the systematic failure or misbehavior of single nodes, there are no inherent guarantees wrt. the authenticity of distributed computations being provided. We address this shortcoming by protecting application code with a powerful yet small PMA. The isolation and attestation features of this PMA protect all components of a distributed application throughout that application's entire life-cycle.

The key feature of all PMAs is to enable the execution of a unit of security sensitive code in isolation from other code. We refer the reader to [1] for a discussion of these properties with respect to security. A number of PMAs have been presented that implement memory isolation for general purpose CPUs [12,2,9,8,18].

Our implementation targets low-cost embedded systems which generally have no virtual memory. As shown in recent work on Program Counter Based Access Control (PCBAC) [5,14,3], memory isolation can be efficiently implemented in hardware for these architectures, enabling minimalist hardware-only Trusted Computing Bases (TCBs). To implement our notion of secure I/O, PMAs have to provide mechanisms to enable a PM to claim exclusive access of an I/O device's registers. Apart from Sancus, this has been demonstrated using SMART [5] and TrustZone [2]. A second feature of PMAs is attestation, the ability to provide assurance of the integrity and isolation of a PM to a third party. SGX [12], SMART [5], TyTAN [3], and Sancus [14] implement cryptographic schemes to attest PMs and to further enable authenticated and confidential communication with a PM.

The VC3 system [17] 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.

An earlier technique to establish a notion of trusted I/O is *BitE* [10]. BitE prevents user-space malware from accessing user input, relying on a trusted mobile device and a host platform capable of attesting its current software state. By introducing PMs that leverage Trusted eXecution Technology (TXT) support in modern CPUs, *Flicker* [9] mitigates the difficulty of attesting a complex host platform. *Bumpy* [11] builds upon BitE and Flicker and establishes a trusted path between encrypting input devices and remote services. Our approach to trusted I/O improves over Bumpy 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. When communicating with a host that interacts with Sancus nodes, techniques such as Flicker, Fides or SGX can be used to protect host services and to further reduce the TCB.

A number of techniques aim to establish a trusted path between an input peripheral and an application by guaranteeing that OS drivers and applications run in isolation and have exclusive access to peripherals [4,19,20], ensuring the confidentiality of user input in the presence of malware. 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. To improve on the approaches listed above, we are currently experimenting with interfacing Sancus-enabled I/O devices with SGX enclaves, providing exclusive and confidential access to isolated host drivers *without* trusting the underlying communication channels.

References

1. Agten, P., Strackx, R., Jacobs, B., and Piessens, F. Secure compilation to modern processors. In *CSF*, pp. 171–185. IEEE, 2012.
2. Alves, T. and Felton, D. TrustZone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.
3. Brasser, F., El Mahjoub, B., Sadeghi, A.-R., Wachsmann, C., and Koeberl, P. TyTAN: Tiny trust anchor for tiny devices. In *DAC*, pp. 34:1–34:6. ACM, 2015.
4. Cheng, Y., Ding, X., and Deng, R. DriverGuard: A fine-grained protection on i/o flows. In *ESORICS*, vol. 6879 of *LNCS*. Springer, 2011.
5. Eldefrawy, K., Francillon, A., Perito, D., and Tsudik, G. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS*, 2012.
6. 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.
7. Lopez, J., Roman, R., Agudo, I., and Fernandez-Gago, C. Trust management systems for wireless sensor networks: Best practices. *Comput. Commun.*, 33(9):1086–1093, 2010.
8. McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. Trustvisor: Efficient tcb reduction and attestation. In *Symposium on Security and Privacy*, pp. 143–158. IEEE, 2010.
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. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *HASP*, pp. 10:1–10:1. ACM, 2013.
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. Noorman, J., Mühlberg, J. T., and Piessens, F. Authentic execution of distributed event-driven applications with a small TCB. In *STM '17*, LNCS, Heidelberg, 2017. Springer. Accepted for publication.
16. Roman, R., Najera, P., and Lopez, J. Securing the internet of things. *Computer*, 44(9):51–58, 2011.
17. 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.
18. Strackx, R. and Piessens, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS*, pp. 2–13. ACM, 2012.
19. Zhou, Z., Gligor, V. D., Newsome, J., and McCune, J. M. Building verifiable trusted path on commodity x86 computers. In *Symp. S&P*. IEEE, 2012.
20. Zhou, Z., Yu, M., and Gligor, V. D. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *Symp. S&P*, pp. 308–323. IEEE, 2014.