Automated Construction of Security Integrity Wrappers for Industry 4.0 Applications

Vivek Nigam[1,3] and Carolyn Talcott[2]

[1] fortiss, Munich, Germany, nigam@fortiss.org
[2] SRI International, Melno Park, USA, clt@csl.sri.com
[3] Federal University of Paraíba, João Pessoa, Brazil

Abstract. Industry 4.0 (I4.0) refers to the trend towards automation and data exchange in manufacturing technologies and processes which include cyber-physical systems, where the internet of things connect with each other and the environment via networking. This new connectivity opens systems to attacks, by, *e.g.*, injecting or tampering with messages. The solution supported by standards such as OPC-UA is to sign and/or encrypt messages. However, given the limited resources of devices, instead of applying crypto algorithms to all messages in the network, it is better to focus on the messages that if tampered with or injected, could lead to undesired configurations.
This paper describes a framework for developing and analyzing formal executable specifications of I4.0 applications in Maude. The framework supports the engineering design workflow using theory transformations that include algorithms to enumerate network attacks leading to undesired states, and to determine wrappers preventing these attacks. In particular, given a deployment map from application components to devices we define a theory transformation that models execution of the application on the given set of (networked) devices. Given an enumeration of attacks (message flows) we define a further theory transformation that wraps each device with policies for signing/signature checking for just those messages needed to prevent the attacks.

1 Introduction

Manufacturing technologies and processes are increasingly automated with highly interconnected components that may include simple sensors and controllers as well as cyber-physical systems and the Internet of Things (IoT) components. This trend is sometimes referred to Industry 4.0 (I4.0). Among other benefits, I4.0 enables process agility and product specialization. This increase of interconnectivity has also enabled cyber-attacks. These attacks can lead to catastrophic events possibly leading to material and human damages. For example, after an attack on a steel mill, the factory had to stop its production leading to great financial loss [1].

A recent BSI report on the security of OPC-UA (machine to machine communication protocol for industrial automation) [7], points out that the lack of signed and encrypted messages on sensitive parts of the factory network can lead to high risk attacks. For example, attackers can inject or tamper with messages, confusing factory controllers and ultimately leading to a stalled or fatal state. Given the limited bandwidth and processing power of I4.0 elements, instead of signing all messages, it is much better to only sign the messages that when not protected could be modified or injected by an intruder to lead to an undesirable situation. This leads to the question of how to determine critical communications.

This paper presents a formal framework for specifying I4.0 applications and analyzing safety and security properties using Maude [4]. The engineering development process from application design and testing to systems deployment is captured by theory transformations with associated theorems showing that results of analysis carried out at the abstract application level hold for models of deployed systems.

Our key contributions are as follows:

– **I4.0 Application Behavior:** We present a formal executable model of the behavior of I4.0 applications in the rewriting logic system Maude [4]. An application is composed of interacting state transition machines which, following the IEC 61499 standard [19], we call function blocks. Maude's search capability is used to formally check such applications for logical defects, which may lead to unsafe conditions.
– **Bounded Symbolic Intruder Model:** To evaluate the security of an application, we formalize a family of bounded intruders parameterized by the number of messages the intruder can inject. Our intruder can generate any clear text message, but can not generate (or read) messages signed by honest devices. To reduce state space complexity the intruder model is converted to one in which messages are symbolic and are instantiated opportunistically according to what can be received at a given time. Using search in the symbolic model all intruder message sets that can lead to a bad state can be enumerated. Each such message defines a flow between two function blocks that must be protected. Proof of the *Intruder Theorem* shows that the concrete and symbolic intruder models yield the same attacks.
– **Deployment transformation:** The application model is suited to reason about functionality and message flows, but does not support reasoning about resources and communication issues that arise when function blocks run on different devices. We define a theory transformation from an application executable specification to a specification of a deployment of that application using a map from application function blocks to a given set of devices. Proof of the *Deployment Theorem* shows that in the absence of intruders, applications and their deployments satisfy the same function block based properties. Proof of the *Deployment Intruder Theorem* shows that any bounded intruder attack at the system level can be found at already at the application level. Thus one can carry out security verification at the application level as the results can be transfered to deployed applications.
– **Security Integrity Wrappers:** Use of security wrappers is a mechanism to protect communications [3]. Here it is used to secure message integrity between devices using signing. Since signing is expensive, it is important to minimize message signing. We define a transformation from a specification of a deployed application to one in which devices are wrapped with a policy enforcement layer where the policies are computed from a set of message flows that must be protected as determined by the enumeration of possible attacks. The proof of the *Wrapping theorem* shows that the wrapping transformation protects the deployed system against identified attacks.

We have implemented the framework and carried out a number of experiments demonstrating analysis, deployment, and wrapping for variations of a PickNPlace application. The Maude code along with documentation, scenarios, sample runs and a technical report with details and proofs can be found at `https://github.com/SRI-CSL/WrapPat.git`. An early version of the framework was presented in [14] where we demonstrated the use of the search command to find logical defects and enumerate attacks, and proposed the idea of device wrappers. That paper contains a number of experiments, including scalability results. The new contributions include the theorems and proofs, implementation of the deployment and wrapping functions, and a simplified version of the symbolic intruder model.

Plan: Section 2 gives an overview of technical ideas and theorems, and describes a motivating example, which will be used as a running example in the paper. Section 3 presents the formalization of our I4.0 framework and bounded attack model in Maude: the application level, the deployment and security wrapper transformations, and theorems characterizing the guarantees of the transformations. Section 4 discusses related work, and Section 5 concludes with ideas for future work.

2 Overview

Threat Model We assume that devices have their pair of secret and public key. Moreover, that devices can be trusted and that a secret key is only known by its corresponding device. However, the communication channels shared by devices are not trusted. An intruder can, for example, inject and tamper with (unsigned) messages in any communication channel. This intruder model reflects the critical types of attacks in Industry 4.0 applications as per the BSI report [7].

To protect communications between function blocks on different devices we use the idea of formal wrapper [3] to transform a system `S` into a system, `wrap(S,emsgs)`, in which system devices are wrapped in a policy layer protecting communications between devices by signing messages and checking signatures on flows. Intuitively, a security integrity wrapper enforces a policy that specifies which incoming events a device will accept only if they are correctly signed and which outgoing events should be signed. By using security integrity wrappers it is possible to prevent injection attacks. For example, if all possible incoming events expected in a device are to be signed, then any message injected by an intruder would be rejected by the device. However, more messages in security integrity wrappers means greater computational and network overhead. One goal of our work is to derive security integrity wrappers, $WR_1, \ldots, WR_n$, for devices, $Dev_1, \ldots, Dev_n$ in which software, called function blocks, are to be executed, to ensure the security of an application while minimizing the number of events that must be signed.

Figure 1 depicts the key components in achieving this goal with the inputs:

- **Application (App):** a set, $\{FB_1, \ldots, FB_n\}$, of function blocks (FBs) and links, $Links$ between output and target input ports. An FB is a finite state machine similar to a Mealy Machine. An App executes its function blocks in cycles. In each cycle, the input pool is delivered to function block inputs and each function block fires one transition if possible. The remaining inputs are cleared, the function block outputs

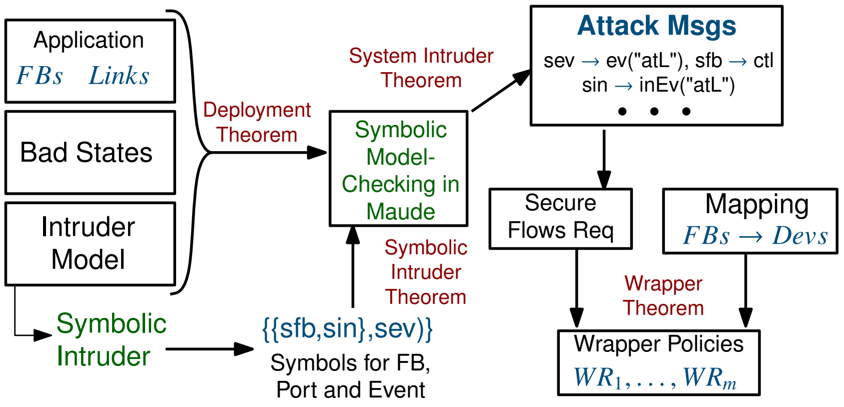

Fig. 1. Methodology Overview

are collected, routed along the application links, and stored in the application input pool.

- **Bad State:** a predicate (`badstate`) specifying which combined FB states should be avoided, for example, states that correspond to catastrophic situations.
- **Intruder Capabilities:** The intruder is given a set of all possible messages deliverable in the given application. For up to n times the intruder can pick a message from this set and inject it into the application input pool at any moment of execution.

We use a symbolic representation of intruder messages and Maude's search capability to determine which messages, called *attack messages*, that an intruder can inject to drive the system to a bad state. Due to the finiteness of the FBs, applications either get stuck or are periodic. Thus, due to Maude's loop detection, the search is finite, as a search path is interrupted whenever a state that has been visited is re-visited. Using reflection and the search descent functions, we enumerate the critical events, *i.e.*, injected message sets leading to a bad state, given an application in a symbolic intruder environment.

Deploying an application is a theory transformation [13]. The function `deployApp` takes an application and a deployment mapping from FBs to devices and returns a system model that is the deployed version of the application corresponding to the mapping.

From the enumerated attack messages, we derive which flows between function blocks on different devices need to have their events signed. Finally, from these flows, we are able to derive the security integrity wrapper policies for a given mapping of function blocks to devices.

Notice that we are able to capture multi-stage attacks, where the system is moved to multiple states before reaching a bad state. This is done by using stronger intruders that can use a greater number of messages.

Challenges To achieve our goal, we encounter a number of challenges.

- **Challenge 1 (Deployment Agnostic):** As pointed out above, the deployment of FBs on devices can affect the security requirements of flows. Analysis at the system level is more complex than at the application level. Thus it is important to understand how analysis on the application level can be transferred to the system level.
- **Challenge 2 (Symbolic Intruder):** Our intruder possess a set of concrete messages and a bound n on the number of injections. The search space grows rapidly with the bound. To reduce the search space, the concrete messages and bound n is replaced by

n distinct symbolic messages. The symbols are instantiated only when required. It is important to understand the realtion of the symbolic model to the concrete model.

– **Challenge 3 (Complete Set of Attack Messages):** Given an intruder, how do we know that at the end the set of attack messages found is a complete set for any deployment?
– **Challenge 4 (System Security by Wrapping):** How do we know that the wrappers constructed from identified flows and deployment mapping ensure the security of the system assuming our threat model?

To address these challenges, we prove the following theorems:

Symbolic Intruder Theorem (Theorem 1) states that each execution of an application `A` in a symbolic intruder environment has a corresponding execution of `A` in the concrete intruder environment with the same bound, and conversely. Thus, using the symbolic intruder is sound and complete with respect to the concrete intruder–enumeration of attacks gives the same result in both cases. The key to this result is the soundness and completeness of the symbolic match generation.

Deployment Theorem (Theorem 2) states that executions of an application `A` and a deployment `S` of `A` are in close correspondence. In particular the underlying function block transitions are the same and thus properties that depend only on function block states are preserved.

System Intruder Theorem (Theorem 3) states that, letting `A`, `S` be as in the Deployment Theorem, (1) for any execution of `S` in an intruder environment there is a corresponding execution of `A` in that environment; and (2) for any execution of `A` in an intruder environment that does not deliver any intruder messages that should flow on links internal to some device, has a corresponding execution from `S` in that environment. Corresponding executions preserve attacks and FB properties. The condition in part (2) is because internal messages are protected by the device execution semantics.

Wrapper Theorem (Theorem 4) Let `A` be an application, `S` a deployment of `A`, and `emsgs` a set of messages containing the attack messages enumerated by symbolic search with an n bounded intruder. The wrapper theorem says that the wrapped system `wrap(S,emsgs)` is resistant to attacks by an n bounded intruder.

2.1 Example

Consider an I4.0 unit, called Pick and Place (PnP),[4] used to place a cap on a cylinder. The cylinder moving on the conveyor belt is stopped by the PnP at the correct location. Then an arm picks a cap from the cap repository, by using a suction mechanism that generates a vacuum between the arm gripper and the cap. The arm is then moved, so that the cap is over the cylinder and then placed on the cylinder. Finally, the cylinder with the cap moves to the next factory element, e.g., storage element.

An application implementing the PnP logic has three function blocks that communicate using the channels as shown in Figure 2. The controller, ctl, coordinates with the vac and track function blocks as specified by the finite machine in Figure 2. For

[4] See `https://www.youtube.com/watch?v=Tkcv-mbhYqk` starting at time 55 seconds for a very small scale version of the PnP.

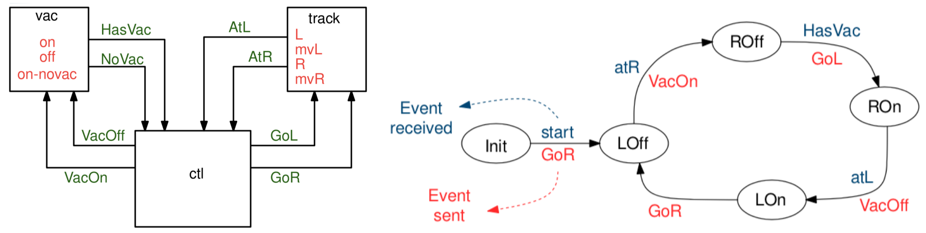

Fig. 2. PnP Function Blocks, ctl, vac, and track. The internal states of vac and track are shown in their corresponding boxes and their transitions are elided. The complete specification is shown in the finite machine to the right.

example, after starting, it sends the message GoR to the Arm that then moves to the right-most position where the caps are to be picked. When the Arm reaches this position, it informs the controller by sending the message atR. The controller then sends the message VacOn to the vac function block that starts its vacuum mechanism. If a vacuum is formed indicating that a cap has been picked, vac sends the message on-hasVac to the ctl. The controller then sends GoL to the track that then moves to the left-most position where the cylinder is located on which the cap has to be placed. The track sends the message atL. The controller then sends the message VacOff to the vac to turn off the vacuum mechanism causing the cap to be placed over the cylinder.

For larger scale PnP, the hazard "Unintended Release of Cap" is catastrophic, for example picking bricks rather than caps, as dropping a brick can hurt someone that is near the PnP. By performing analysis, such as STPA (Systems-Theoretic Process Analysis), one can determine that this event can occur when *The track function block is at state mvL and the vac function block is in state on-noVac or in state off.* This is because when starting to move to the position to the left, the gripper may have succeeded to grab a cap. However, while the arm is moving, the vacuum may have been lost causing the cap to be released, *i.e.*, the vac function block is in state on-noVac or off. An intruder can cause such an event by injecting the message VacOff to the vac when the arm is moving left, that is, in state mvL, while the gripper is holding something.

Following our methodology, shown in Figure 1, we feed to our Symbolic Model-Checker the PnP function blocks, its bad state above, and a symbolic intruder that can inject at most one message. One can specify stronger intruders, but this weak intruder is already able to lead the system into a bad state. Indeed, from the model-checker's output, we find that there are four different attack messages. One of them is shown in Figure 1, where the intruder impersonates the track and sends to the ctl a message atL while the track is still moving.

From the identified attack messages we can see that messages in the flow from the track to the ctl involving the message atL should be protected.

Finally, suppose track and ctl are deployed in dev_1 and dev_2, respectively, then the computed security integrity wrapper on dev_1 will sign atL messages, and the security integrity wrapper on dev_2 will check whether atL messages are signed by dev_1. If track and ctl are deployed on the same device, there is no need to sign atL messages as we trust devices to protect internal communications.

3 Formalization of the I4.0 framework in Maude

We now describe the formal representation of applications, and the deployment and wrapping transformations. We formalize theorems. We describe the main structures, operations, and rules using snippets from the Maude specification. Examples come from the Maude formalization of the PnP application of Section 2.

3.1 Function blocks

An I4.0 application is composed of a set of interconnected interactive finite state machines called function blocks. A function block is characterized by its finite set of states, finite sets of inputs and outputs, a finite set of possible events at each input or output, and a finite set of transitions. We call this a class and we give FBs both an instance and a class identifier to allow for multiple occurrences of a given class.

The Maude representation of a function block (FB) is a term of the form `[fbId : fbCid | fbAttrs]`, where `fbId` is the FB identifier, `fbCid` its class identifier and `fbAttrs` is a set of attribute-value pairs, including `(state : st)`, `(oEvEffs : oeffs)`, and `(ticked : b)`, with `state`, `oEvEffs`, `ticked` being the attribute tags, `st` the current state, `oeffs` a set of signals/events to be transmitted (out effects), and `b` a boolean indicating whether the FB has fired a transition in the current cycle.

A transition is a term of the form `tr(st0,st1,cond,oeffs)` where `st0` is the initial state and `st1` the final state, `cond` is the condition, and `oeffs` is the set of outputs. A condition is a boolean combination of primitive conditions `(in is ev)` specifying a particular event (`ev`) at input `in`. A transition `tr(st0,st1,cond,oeffs)` is enabled by a set of inputs if they satisfy `cond` and the current state of the function block state `st0`. In this case, the transition can fire, changing the function block state to `st1` and emitting `oeffs`.

Example FB. The FB with class id `vac` has states

```
 st("off"), {st("on"), st("on-novac");
```

inputs

```
  inEv("VacOn"), inEv("VacOff");
```

outputs

```
  outEv("NoVac"), outEv("HasVac").
```

The initial state, `vacInit(id("vac"))`, of an FB with class `vac` and identifier `id("vac")` is

```
[id("vac") : vac | state : st("off") ; ticked : false ;
                   iEvEffs : none ; oEvEffs : none]
```

The function `trsFB(fbCid)` returns the set of transitions for function blocks of class `fbCid`. `trsFB(fbCid,st)` selects the transitions in `trs(fbCid)` with initial state `st`. For example `trsFB(vac, st("off"))` returns three transitions

```
tr(st("on"), st("off"), inEv("VacOff") is ev("VacOff"),
   outEv("NoVac") :~ ev("NoVac"))
tr(st("off"), st("on-novac"), inEv("VacOn") is ev("VacOn"),
   outEv("NoVac") :~ ev("NoVac"))
tr(st("off"), st("on"), inEv("VacOn") is ev("VacOn"),
   outEv("HasVac") :~ ev("HasVac"))
```

We compile a transition condition into a representation as a set of constraint sets which simplifies satisfaction checking, and matching when messages are symbolic. We can think of a constraint set (CSet) as a finite map from function block inputs to finite sets of events. A set of inputs `ieffs` $= \{(\texttt{in}_i \rhd \texttt{ev}_i) | 1 \leq i \leq k\}$ satisfies a CSet, `css`, just if `css` has size k, the $\texttt{in}_i$ form a set equal to the domain of `css`, and $\texttt{ev}_i$ is in $\texttt{css}(in_i)$ for $1 \leq i \leq k$. The function `condToCSet(cond)` returns the set of CSets such that an input set satisfies some CSet in the result just if it satisfies `cond`. This is lifted to transitions by the function

```
tr2symtr(tr(st1,st2,cond,oeffs) =
         symtr(st1,st2,condToCSet(cond),oeffs) .
```

For the `vac` example, the CSet

```
condToCSet( inEv("VacOn") is ev("VacOn"))
```

maps `inEv("VacOn")` to the singleton `ev("VacOn")`.

3.2 Application structure and semantics

An application term has the form `[appId | appAttrs]`. Here `appAttrs` is a set of attribute-value pairs including `(fbs : funBs)` and `(iEMsgs : emsgs)`, where `funBs` is a set of function blocks (with unique identifiers), and `emsgs` is the set of incoming messages of the form `{{fbId,in},ev}`.

We use `fbId`, `fbId0` ... for FB identifiers, `in/out` for FB input/output connections, and `ev` for the event transmitted by a message. Terms of the form `{fbId,in/out}` are called Ports. For entities X with attributes, we write X.tag for the value of the attribute of X with name ‘tag’.

The initial state of the PickNPlace (PnP) application described in Section 2 is

```
[id("pnp") | fbs : (ctlInit(id("ctl")
                trackInit(id("track")) vacInit(id("vac"))) ;
  iEMsgs : {{id("ctl"),inEv("start")},ev("start")} ;
  oEMsgs : none ; ssbs : none]
```

where the message `{{id("ctl"),inEv("start")},ev("start")}` starts the application controller.

Links of the form `{{fbId0,out},{fbId1,in}}` connect output ports of one FB to inputs of another possibly the same FB. They also connect application level inputs to FB inputs and FB external outputs to application level outputs. In a well formed application, each FB input has exactly one incoming link. In principle the link set is an attribute of the application structure. In practice, since it models fixed 'wires' connecting function block outputs and inputs and does not change, to avoid redundant information in traces, we specify a function `appLinks(appId)` which is defined in application specific scenario modules.

As an example, here are the two links that connect `vac` outputs to controller inputs.

```
{{id("vac"),outEv("NoVac")}, {id("ctl"),inEv("NoVac")}}
{{id("vac"),outEv("HasVac")},{id("ctl"),inEv("HasVac")}}
```

Application Execution Rules. There are two execution rules for application behavior and two rules modeling bounded intruder actions, one for the concrete case and one for the symbolic case. To ensure that an FB fires at most one transition per cycle, each FB is given a boolean `ticked` attribute, initially `false`, which is set to `true` when a transition fires, and reset to `false` when the outputs are collected.

The rule `[app-exe1]` fires an enabled function block transition and sets the ticked attribute to `true`.

```
crl[app-exe1]:
  [appId | fbs : ([fbId : fbCid | (state : st) ;
     (ticked : false) ; oEvEffs : none ;  fbAttrs] fbs1) ;
     iEMsgs : (emsgs0 iemsgs) ; ssbs : ssbs0 ;  appAttrs ]
=>
  [appId | fbs : ([fbId : fbCid | (state : st1) ;
     (ticked : true) ; oEvEffs : oeffs ;  fbAttrs] fbs1) ;
     iEMsgs : iemsgs) ; ssbs : (ssbs0 ssbs1) ;  appAttrs ]
 if symtr(st,st1,[css] csss,oeffs) symtrs := symtrsFB(fbCid,st)
 /\ size(emsgs0) = size(css)
 /\({ssbs1} ssbss) := genSol1(fbId,emsgs0,css) .
```

The function `genSol1(fbId,emsgs0,css)` returns a set of substitutions, consisting of all and only substitutions that match `emsgs0` to a solution of the CSet, `css`. In the case of concrete messages, *i.e.*, not containing symbols, the function `genSol1` just returns an empty substitution if `emsgs0` satisfies `css`. When rewriting, just one partition of `iemsgs`, one choice of (symbolic) transition, and one satisfying substitution is selected. Search will explore all possible choices.

When `[app-exe1]` is no longer applicable (`hasSol(fbs,iemsgs)` is `false`), `[app-exe2]` collects and routes generated output and prepares for the next cycle.

```
crl[app-exe2]: [appId | fbs : fbs ; iEMsgs : iemsgs ;
                       oEMsgs : oemsgs ; ssbs : ssbs ; attrs]
=> [appId | fbs : fbs2 ; iEMsgs : emsgs0 ;
           oEMsgs : (oemsgs emsgs1) ; ssbs : ssbs  ; attrs1]
 if not hasSol(fbs,iemsgs)
 /\ tick := notApp(attrs)
 /\ not getTicked(attrs)  --- avoid extracting when no trans
 /\ attrs1 := setTicked(attrs, true)
 /\ {fbs2,emsgs0,emsgs1} :=
    extractOutMsgs(tick,fbs,none, none,none,appLinks(appId)) .
```

The function `extractOutMsgs` removes outputs from the function blocks that fired and routes them using `appLinks(appId)` to the linked FB input or application output. Application level inputs are accumulated in `emsgs0` and outputs are accumulated in `emsgs1`. The ticked attribute of each FB is set to the value of `tick`. In the case of a basic application, this will be `false` indicating the FB is ready for the next cycle. When the application level execution rules are used in a larger context, (`notApp(attrs)` is `true`), `extractOutMsgs` ensures that each FBs ticked attribute is `true`, allowing further message processing before repeating the execution cycle. If the application has a ticked attribute, it is set to `true`, to indicate it has completed the current cycle. `fbs2` collects the updated function blocks.

3.3 Intruders

An application `A` in the context of an intruder is represented in the concrete case by a term of the form `[A, emsgs, n]` where `emsgs` is a set of specific messages (typically all the messages that could be delivered) and n is the number of injection actions remaining. The rule `[app-intruder-c]` (omitted) selects one of the candidate messages, injects it, and decrements the counter.
An application `A` in the context of a symbolic intruder is represented by a structure of the form `[A, smsgs]` where `smsgs` are symbolic intruder messages of the form `{{idSym,inSym},evSym}` (`idSym`, `inSym`, `evSym` are symbols standing for function block identifiers, inputs, and events respectively). We require different messages to have distinct symbols. The rule `[app-intruder]` selects one of the intruder messages, and moves it from the intruder message set to the incoming messages `iEMsgs`.

```
rl[app-intruder]:
[[appId | fbs : fbs ; iEMsgs : emsgs0 ;  attrs], emsg emsgs]
=>
[[appId | fbs : fbs ; iEMsgs : (emsgs0 emsg) ; attrs], emsgs] .
```

We note that this rule works equally well with concrete or symbolic messages, allowing one to explore consequences of injecting specific messages. Using `genSol1`, a symbolic message can be instantiated to any deliverable message. Also, if a message is injected after all function blocks have been ticked and before `[app-exe2]` is applied, it will be dropped by `[app-exe2]`, since function block inputs are cleared before collecting the next round of inputs.

3.4 The Intruder Theorem

We define the following correspondence between symbolic and concrete intruder states:
`[A,smsgs]` $\sim$ `[A,cmsgs,n]` holds only if
- `size(smsgs)` $= n$,
- `As.fbs` $=$ `Ac.fbs`, and
- `(As.iEMsgs)[ssbs]` $=$ `Ac.iEMsgs`

for some symbol substitution `ssbs`.[5] Two rule instances correspond if they are instances of the same rule. Also, in the `[app-exe1]` case the instances are the same transition of FBs with the same identifier, and in the `[app-exe2]` case the instances collect the same outputs.

An execution trace is an alternating sequence of (application) states and rule instances connecting adjacent states as usual. A symbolic trace TrS from `[A,smsgs]` and a concrete trace TrC from `[A,emsgs,n]` correspond just if they have the same length and the i^{th} elements correspond as defined above.

Theorem 1. Let `[A,smsgs]` $\sim$ `[A,cmsgs,n]` be corresponding initial application states in symbolic and concrete intruder environments respectively, where no intruder messages have been injected.

If TrS is an execution trace from `[A,smsgs]` then there is a corresponding execution trace TrC starting with `[A,cmsgs,n]` and conversely.

Proof. By induction on trace length. The base case is simple in either direction, since an intruder message is only involved if the rule is an `app-intruder` rule. Let

$$TrS = TrS_0 \to [\mathtt{As}_k, \mathtt{smsgs}_k] - rl_k \to [\mathtt{As}_k + 1, \mathtt{smsgs}_{k+1}]$$

be an execution trace from `[A,smsgs]`. By induction, let

$$TrC_0(\mathtt{pmsgs}) \to [\mathtt{Ac}_k, \mathtt{cmsgs}, n_k]$$

be the set of corresponding concrete traces from `[A,cmsgs,n]` where `pmsgs` are parameters for delayed choices of injected concrete messages that remain in `iEMsgs` (have been injected and not delivered or cleared), thus were injected since the last `[app-exe2]` rule. If rl_k is an instance of `[app-exe1]` then

$$\mathtt{As}_k.\mathtt{iEMsgs} = \mathtt{iemsgs} = \mathtt{iemsgs0}\ \mathtt{emsgs0}$$

and the function block with identifier `fbId` has a transition delivering `emsgs0[ssbs]`. Let `iemsgs0` $=$ `iemsgs00 iemsgs01` and `emsgs0` $=$ `emsgs00 emsgs01` where `iemsgs00`, `emsgs00` are concrete and `iemsgs01`, `emsgs01` are symbolic. By the correspondence

$$\mathtt{Ac}_k.\mathtt{iEMsgs} = \mathtt{iemsgs00}\ \mathtt{ipmsgs01}\ \mathtt{emsgs00}\ \mathtt{pmsgs01}$$

where `ipmsgs01 pmsgs01` are the injection message parameters such that the following equations are satisfied:

$$\mathtt{size(pmsgs01)} = \mathtt{size(emsgs01)} \quad \mathtt{size(ipmsgs01)} = \mathtt{size(iemsgs01)}$$

$\mathtt{Ac}_k$ can deliver the same messages to the same function block. Let `pmsgs01` $=$ `emsgs01[ssbs]`. We extend TrC by a applying of `[app-exe1]` to

$$[\mathtt{A}_{k+1}, \mathtt{pmsgs00}] = [Ac_k[\mathtt{pmsgs01} = \mathtt{emsgs01[ssbs]}], \mathtt{cmsgs}, n_k].$$

[5] Note that the attributes `ssbs` and `oEMsgs` do not affect rule application.

For rl_k an instance of `[app-exe2]` or the intruder rule it is easy to see that TrC extends to a corresponding trace.

Conversely, let

$$TrC = TrC_0 \to [\mathtt{Ac}_k, \mathtt{cmsgs}_k, n_k] - rl_k \to [\mathtt{Ac}_{k+1}, \mathtt{cmsgs}_{k+1}, n_{k+1}]$$

be a concrete trace. By induction let $TrS_0 \to [\mathtt{As}_k, \mathtt{smsgs}_k]$ be a corresponding symbolic trace. If rl_k is an instance of `crl[app-exe1]` then

$$\mathtt{Ac}_k.\mathtt{iEMsgs} = \mathtt{iemsgs} = \mathtt{iemsgs0}\ \mathtt{emsgs0}$$

and function block with identifier `fbId` has a transition delivering `emsgs0`. Let `ssbs` be a substitution such that $\mathtt{As}_k.\mathtt{iEMsgs} = \mathtt{iemsgs'} = \mathtt{iemsgs0'}\ \mathtt{emsgs0'}$ and `emsgs0'[ssbs] = emsgs0`. By 'completeness' of `genSol1`, `ssbs` will be a solution generated by `genSol1` and

$$[\mathtt{As}_k, \mathtt{smsgs}_k] - rl_k \to [\mathtt{As}_{k+1}, \mathtt{smsgs}_k]\ [\mathtt{Ac}_{k+1}, \mathtt{cmsgs}_{k+1}, n_{k+1}]$$

extending TrS_0 to TrS corresponding to TrC. If rl_k is an instance of `[app-exe2]` or an intruder rule it is easy to see that TrS_0 extends as desired.

Corollary 1. Search using the symbolic intruder model for paths reaching a `badState` finds all successful (bounded intruder) attacks.

We define the function `getBadEMsgs([A,smsgs])` that returns the set of injected message sets that lead to `badState`. This function uses reflection to enumerate search paths reflecting the command

```
  search [A,smsgs] =>+ appInt:AppIntruder
          such that badState(appInt:AppIntruder) .
```

Injected symbolic messages are determined by looking for adjacent states where the symbolic message set decreases. The symbols of injected messages that were actually delivered are in the domain of the value of the `sbss` attribute of the final state.

In the PnP application for an intruder with a single message, `getBadEMsgs` returns four attack message sets

```
{{{id("ctl"),inEv("HasVac")},ev("HasVac")}}
{{{id("ctl"),inEv("atL")},ev("atL")}}
{{{id("track"),inEv("GoL")},ev("GoL")}}
{{{id("vac"),inEv("VacOff")},ev("VacOff")}}
```

Recall from Section 2 that the PnP application state satisfies `badState` if the `track` FB is in state `st("mvL")`, presumably carrying something from right to left, and the `vac` FB is in an *off* state (`st("on-novac")` or `st("off")`).

3.5 Deploying an Application

Once an application is designed, the next step is determining how to deploy FBs on devices. We model deployment as a theory transformation, introducing a data structure to represent deployed applications, called *System*s, extending the application module with rules to model system level communication elements, and defining a function mapping applications to their deployment given an assignment of FBs to host devices.

A deployed application is represented in Maude by terms of the form: `[sysId | appId | sysAttrs]` where `sysAttrs` is a set of attribute-value pairs including `(devs : devs)` and `(iMsgs : msgs)`. `devs` is a set of devices, and `msgs` is a set of system level messages of the form `{srcPort,tgtPort,ev}` where `srcPort`/`tgtPort` are terms of the form `{devId, {fbId, out/in}}`.

A device is represented as an application term with additional attributes including `(ticked : b)` which indicates whether all FBs have had a chance to execute. The function blocks of the application named by `appId` are distributed amongst the devices. The function `sysMap(sysId)` maps each FB identifier to the identifier of the device where the FB is hosted. Each device has incoming/outgoing ports corresponding to links between its function blocks and function blocks on other devices.

The function `deployApp(sysId,A,sysMap(sysId))` produces the deployment of application `A` as a system with identifier `sysId` and FBs distributed to devices according to `sysMap(sysId)`.

```
ceq deployApp(sysId,app,idmap) =
       mkSys(sysId,getId(app),devs,msgs)
if emsgs := getIEMsgs(app)
/\ devs := deployFBs(getFBs(app),none,idmap)
/\ msgs := emsgs2imsgs(sysId,emsgs,idmap,none) .
```

The real work is done by the function `deployFBs(fbs,none,idmap)` which creates an empty device for each device identifier in the range of `idmap` (setting `iMsgs` to `none` and `ticked` to `true`). Then each FB (identifier `fbId`) of `app` is added to the `fbs` attribute of the device identified by `idmap[fbId]`.

Note that the `deployApp` function can be applied to any state $\mathtt{A}_k$ in an execution trace from `A`. A system $\mathtt{S}_k$ can be abstracted to an application by collecting all the device FBs in the application `fbs` attribute, collecting the `iEMsgs` attributes of devices into the `iEMsgs` attribute of the application and adding system level input messages to the `iEMsgs` attribute of the application (after conversion to application level).

The execution rules for applications apply to devices in a system. There are two additional rules for system execution: `[sys-deliver]` and `[sys-collect]`.
The rule `[sys-deliver]` delivers messages associated to the `iMsgs` attribute. The rule requires `isDone` to hold of the system devices, which means all the devices have their `ticked` attribute set to `true`. The target port of a system level message identifies the device and function block for delivery.

The rule `[sys-collect]` collects and distributes messages produced by the application level execution rules. It collects application level output messages from each device and converts them to system level output messages. Messages from device `iEMsgs` attributes are split into local and external. The local messages are left on the device, the external messages are converted to system level input messages.

We define a correspondence between execution traces from an application `A`, and a deployment `S = deployApp(sysId,A,idmap)` of that application. An application state `A1` corresponds to a system state `S1` just if they have the same function blocks and the same undelivered messages. (Note that the deployment and abstraction operations are subsets of this correspondence relation.) An instance of the `[app-exe1]` rule in an application trace corresponds to the same instance of that rule in a system trace (fires the same transition for the same function block). An instance of `[app-exe2]` in an application trace corresponds to a sequence

```
app-exe2+;sys-collect;sys-deliver
```

in a system trace collecting and delivering corresponding messages.

Theorem 2. Let `A` be an application and `S = deployApp(sysid,A,idmap)` be a deployment of `A`. Then `A` and `S` have corresponding executions.

Proof. This is a direct consequence of the definition of corresponding traces.

Corollary 2. `A` and `S` as above satisfy the same properties that are based only on FB states and transitions. This is because corresponding traces have the same underlying function block transitions.

Intruders at the system level Deployed applications are embedded in an intruder environment analogously to applications. We consider a simple case where the intruder has a finite set of concrete messages to inject, using it to show that any attack at the system level can already be found at the application level. A system in a bounded intruder environment is a term of the form `[sys,msgs]` where `sys` is a system as above, and `msgs` is a finite set of system level messages. The deployment function is lifted by

```
deployAppI(sysId,[A,emsgs],idmap) =
  [deployApp[sysId,A,idmap],deployMsgs(emsgs,appLinks(A),idmap)]
```

where `deployMsgs` transforms application level messages `{fbport,ev}` to system level, `{srcdevport,tgtdevport,ev}` using the link and deployment maps.

The intruder injection rule, `[app-intruder]`, is lifted to `[sys-intruder]` and the correspondence relation of the deployment theorem is lifted in the natural way to the intruder case.

Theorem 3. Assume `Ai = [A,emsgs]` where `A` is an application in its initial state (no intruder messages injected) and `Si = deployAppI(sysId,Ai,idmap)`.
1. If TrS is a trace from `Si` then there is a corresponding trace from `Ai`.
2. If `TrA` is a trace from `Ai` that delivers no intruder messages that flow on links internal to a device, then there is a corresponding trace from `Si`.

Proof. The proof is the same as for the correspondence of an application and its deployment. The additional condition in part 2 is needed because a device protects communications between FBs it hosts by having no port for delivery of such messages. In particular, if all the FBs are hosted on a single device then no intruder messages can be delivered.

Corollary 3. If a `badState` is reachable from `Si` then `sys2app(msgs)` is an element of `getBadEMsgs([A,smsgs])` where `size(smsgs) = size(msgs)`.

3.6 Wrapping

Towards the goal of signing only when necessary (Section 2) we define the transformation `wrapApp(A,smsgs,idmap)` of deployed applications as:

```
wrapSys(deployApp(sysId,A,idmap),flatten(getBadEMsgs([A,smsgs])))
```

where flatten unions the sets in a set of sets. `wrapSys(S,emsgs)` wraps the devices in `S` with policies for signing and checking signatures of messages on flows defined by `emsgs` as described below.

A wrapped device has input/output policy attributes `iPol`/`oPol` used to control the flow of messages in and out of the device. An input/output policy is an `iFact`/`oFact` set where an `iFact` has the form `[i : fbId ; in, devId]` and an `oFact` has the form `[o : fbId ; out]`. If `[i : fbId ; in, devId]` is in the input policy of a device then a message `{{fbId,in}, ev}` is accepted by that device only if `ev` is signed by `devId`, otherwise the message is dropped. Dually, if `[o : fbId ; out]` is in the output policy of a device, then when a message `{{fbId,out}, ev}` is transmitted `ev` is signed by the device. Following the usual logical representation of crypto functions, we represent a signed event by a term `sg(ev,devId)`, assuming that only the device with identifier `devId` can produce such a signature, and any device that knows the device identifier can check the signature.

The function `wrapSys(S,emsgs)` invokes the function `wrap-dev` to wrap each of its devices, `S.devs`. In addition to the device, the arguments of this function include the set of messages, `emsgs`, to protect, the application links and the deployment map. The links determine the sending FB, and the deployment determines the sending/receiving devices. If these are the same, no policy facts are added. Otherwise, policy facts are added so the sending device signs the message event and the receiving device checks for a signature according to the rules above.

```
ceq wrap-dev(dev,{{fbId,in},ev} emsgs,links,idmap,ipol,opol)
  = wrap-dev(dev,emsgs,links,idmap,(ipol ipol1), (opol opol1))
if {{fbId0,out},{fbId,in}} links0 := links
/\ devId1 := idmap[fbId]
/\ devId0 := idmap[fbId0]
/\ devId1 =/= devId0    ---- not an internal link
/\ devId := getId(dev)
**** if emsg sent from dev add opol to sign outgoing
/\ opol1 := (if devId == devId0
              then [o : fbId0 ; out ]
              else none
              fi)
**** if emsg rcvd by dev, require signed by sender devId0
/\ ipol1 := (if devId == devId1
              then [i : fbId ; in, devId0]
              else none
              fi) .

eq wrap-dev(dev,emsgs,links,idmap,ipol,opol) =
    addAttr(dev,(iPol : ipol ; oPol : opol)) [owise] .
```

Theorem 4. Assume `A` is an application, `allEMsgs` is the set of all messages deliverable in some execution of `A`, and `smsgs` is a set of symbolic messages of size n. Assume `badState` is not reachable in an execution of `A`, and `emsgs` contains `flatten(getBadEMsgs([A,smsgs]))`.

1. Let `wA` $=$ `[wrapSys(deployApp(sysId,A,idmap),emsgs]`. Every execution from `wA` has a corresponding execution from `A` and conversely. In particular `badState` is not reachable from `wA`.
2. `badState` is not reachable from

$$\texttt{wAC} = \texttt{[wrap(deploy(A,idmap),emsgs),allEMsgs,n]}$$

Proof 1. The proof is similar to the proof of the deployment theorem part 1, noting that by definition of the wrap function, any message in `emsg` will be signed by the sending device and thus will satisfy the receiving device input policy and be delivered in the `wA` trace as it will in the `A` trace.

Proof 2. Assume `badState` is reachable from `wAC`. Let `wAC` $rl_0 \ldots rl_k$ $\texttt{wAC}_{k+1}$ be a witness execution where `badState` holds of $\texttt{wAC}_{k+1}$. By the assumption on `A` from part 1, at least one intruder message must have been delivered.

Let $\{\texttt{emsg}_1 \ldots \texttt{emsg}_l\}$ be the intruder messages delivered in the trace, say by rules $rl_{j_1} \ldots rl_{j_l}$. None of these messages are in `emsgs` since their events cannot be signed by one of the devices, and thus would not satisfy the relevant input policy. Thus there is a corresponding trace from the unwrapped system

$$\texttt{AC} = \texttt{[deploy(A,idmap),allEMsgs,n]}$$

and by the *Deploy Intruder Theorem* there is a corresponding trace from `[A,allEMsgs,n]` reaching a `badState`. But `emsgs` contains all messages that are part of an intruder message set which if injected can cause `badState` to be reached. A contradiction.

4 Related Work

There are a number of recent reports concerning the importance of cybersecurity for Industry 4.0. Two examples are the German Federal Office for Information Security (BSI) commissioned report on OPC UA security [7], and the ENISA study on good practices for IoT security [6]. OPC Unified Architecture (OPC UA) is a standard for networking for Industry 4.0 and includes functionality to secure communication. The BSI commissioned report describes a comprehensive analysis of security objectives and threats, and a detailed analysis of the OPC UA Specification. The analyses are informal but systematic, following established methods. A number of ambiguities and issues were found in this process. The ENISA report provides guidelines and security measures especially aimed at secure integration of IoT devices into systems. It includes a comprehensive review of resources on Industry 4.0 and IoT security, defines concepts, threat taxonomies and attack scenarios. Again, systematic but informal.

Although there is much work on modeling cyber physical systems and cyber-physical security (see [12] for recent review), much of it is based on simulation and testing. The formal modeling work focuses on general CPS and IoT not on the issues specific to I4.0 type situations. Lanotte *et al.* [10] propose a hybrid model of cyber and physical systems and associated models of cyber-physical attacks. Attacks are classified according to target device(s) and timing characteristics. Vulnerability to a given class is assessed based on the trace semantics. A measure of attack impact is proposed along with a means to quantify the chances of success. The proposed model is much more detailed than our model, considering device dynamics, and is focussed on traditional control systems rather than IoT in an Industry 4.0 setting. The attacks on devices modeled include our injection attacks. The Lanotte et. al. work is complementary to ours, while being more detailed we suspect our more abstract model combined with symbolic analysis is more scalable. The work in [16] relates to our work in proposing a method using model-checking to find all attacks on a system given possible attacker actions. The authors do not propose mitigations. SOTERIA [2] is a tool for evaluating safety and security of individual or collections of IoT applications. It uses model-checking to verify properties of abstract models of applications derived automatically from code (of suitable form). It requires access to the application source code.

The idea of using theory transformations to relate the application, system level specifications and reduce many reasoning problems to reasoning at the application level is based on the notion of formal patterns reviewed in [13]. An early example of wrapping to achieve security guarantees is presented in [3] to mitigate DoS attacks.

5 Conclusions and Future Work

This paper presents a formal framework in rewriting logic for exploring I4.0 (smart factory) application designs and a bounded intruder model for security analysis. The framework provides functions for enumerating message injection attacks, and generating policies mitigating such attacks. It provides theory transformations from application specifications to specifications of systems with application components executing on devices, and for wrapping devices to protect against attacks using the generated policies. Theorems relating different specifications and showing preservation of key properties are given. We believe that formal executable models can be valuable to system designers to find corner cases and to explore tradeoffs in design options concerning the cost and benefits of security elements.

Future work includes theory transformations to refine the system level model to a network model with multiple subnets and switches, adding timing and modeling constraints induced by use of the TSN network protocol. As in our previous work [8], we are investigating the complexity of security properties given intruder models weaker than the traditional Dolev-Yao intruder [5]. We are also considering increasing the expressiveness of function block specifications to include time constraints as in [9] to automate the verification of properties based on time trace equivalence [15], such as privacy attacks. Finally, since these devices have limited resources, they may be subject to DDoS attacks. Symbolic verification can be used to check for such vulnerabilities [18].

Another important direction is developing theory transformations for correct-by-construction distributed execution [11]. This means accounting for real timing considerations and network protocols, and identifying conditions under which application and system level properties are preserved. An important use of the framework that we intend to investigate is relating safety and security analyses and connecting formal analyses to the engineering notations used for safety and security.

We are also currently extending our implementation to support the automated exploration of mappings of function blocks to devices. In particular, we are investigating the extension of [17] to take into account security objectives in addition to device performance limitations, device capabilities, and deadlines.

Acknowledgements This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892. Talcott is partly supported by ONR grant N00014-15-1-2202 and NRL grant N0017317-1-G002. Nigam is partially supported by NRL grant N0017317-1-G002, and CNPq grant 303909/2018-8.

References

1. Cyberattack on a German steel-mill, 2016. Available at `https://www.sentryo.net/cyberattack-on-a-german-steel-mill/`.
2. Z. B. Celik, P. McDaniel, and G. Tan. SOTERIA: Automated IoT safety and security analysis. https://arxiv.org/pdf/1805.08876, 2018.
3. Rohit Chadha, Carl A. Gunter, José Meseguer, Ravinder Shankesi, and Mahesh Viswanathan. Modular preservation of safety properties by cookie-based DoS-protection wrappers. In *Proc. FMOODS 2008*, volume 5051 of *LNCS*, pages 39–58. Springer, 2008.
4. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
5. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
6. ENSIA. Good practices for security of internet of things in the context of smart manufacturing, 2018.
7. Mr. Fiat and et.al. OPC UA security analysis, 2017.
8. Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, and Andre Scedrov. Bounded memory Dolev-Yao adversaries in collaborative systems. *Inf. Comput.*, 238:233–261, 2014.
9. Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn L. Talcott. Time, computational complexity, and probability in the analysis of distance-bounding protocols. *Journal of Computer Security*, 25(6):585–630, 2017.

10. Ruggero Lanotte, Massimo Merro, Riccardo Muradore, and Luca Vigano. A formal approach to cyber-physical attacks. In *30th IEEE Computer Security Foundations Symposium*, pages 436–450. IEEE Computer Society, 2017.
11. Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. Automatic transformation of formal maude designs into correct-by-construction distributed implementations. Technical report, 2019.
12. Yuriy Zacchia Lun, Alessandro D'Innocenzo, Ivano Malavolta, and Maria Domenica Di Benedetto. Cyber-physical systems security: a systematic mapping study. *CoRR*, abs/1605.09641, 2016.
13. José Meseguer. Taming distributed system complexity through formal patterns. *Sci. Comput. Program.*, 83:3–34, 2014.
14. Vivek Nigam and Carolyn Talcott. Formal security verification of industry 4.0 applications. In *ETFA, Special Track on Cybersecurity in Industrial Control Systems*, 2019.
15. Vivek Nigam, Carolyn Talcott, and Abraão Aires Urquiza. Symbolic timed trace equivalence. In *Catherine Meadow's Festschirft*, 2019.
16. Farid Molazem Tabrizi and Karthik Pattabiraman. IOT: Formal security analysis of smart embedded systems. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 1–15. ACM, NY, 2016.
17. Tarik Terzimehic, Sebastian Voss, and Monika Wenger. Using design space exploration to calculate deployment configurations of IEC 61499-based systems. In *14th IEEE International Conference on Automation Science and Engineering*, pages 881–886, 2018.
18. Abraão Aires Urquiza, Musab A. AlTurki, Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn L. Talcott. Resource-bounded intruders in denial of service attacks. In *CSF*, pages 382–396, 2019.
19. L. H. Yoong, P. S. Roop, Z E Bhatti, and M. M. Y Kupz. *Model-Driven Design Using IEC 61499: A Synchronous Approach for Embedded Automation Systems*. Springer, 2015.