Formal Security Verification of Industry 4.0 Applications

Vivek Nigam*‡ Carolyn Talcott†
*fortiss, Munich, Germany, nigam@fortiss.org
‡Federal University of Paraíba, João Pessoa, Brazil,
†SRI International, Melno Park, USA, clt@csl.sri.com

***Abstract*—Without appropriate counter-measures, cyber-attacks can exploit the increased system connectivity provided by Industry 4.0 (I4.0) to cause catastrophic events, by, *e.g.*, injecting or tampering with messages. The solution supported by standards, such as, OPC-UA, is to sign or encrypt messages. However, given the limited resources of devices, instead of encrypting all messages in the network, it is better to encrypt only the messages that if tampered with or injected, could lead to undesired configurations. This paper describes the use of formal verification to analyse the security of I4.0 applications. We formalize in Rewriting Logic, I4.0 applications and systems, *i.e.*, networked sets of devices, and a symbolic intruder model. Our formalization can be executed by the tool Maude to automate such security analysis, *e.g.*, determine which messages are sufficient to sign in order avoid injection and tampering attacks.**

I. Introduction

While manufacturing has greatly profited from the increased inter-connectivity of devices, it 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].

Many attacks can be avoided if adequate counter-measures are put in place. As reported by the recent BSI report on the security of OPC-UA [8], 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 thus confusing factory controllers and ultimately leading to a stalled or fatal state.

However, not all messages have to be signed or encrypted. For example, messages that are not used in the safety critical parts of the factory do not necessarily need to be encrypted as even if tampered with they do not lead to catastrophic events.[1] So, given the limited bandwidth and processing power of I4.0 settings, instead of signing all messages, it is much better to only sign or encrypt the messages that when not protected could be modified or injected by an intruder to lead to a catastrophic event.

This paper proposes the use of formal methods for answering such security questions for I4.0 applications. Formal verification has been successfully used in domains, such as, protocol security verification, finding new attacks to protocols [4], [10]. We greatly believe that I4.0 can profit from formal verification in order to analyse the security of I4.0 applications and systems.

Our key contributions are as follows:

I4.0 Behavior: We demonstrate how the behavior of I4.0 applications, such as, those implemented in 4diac™ (see https://www.eclipse.org/4diac/), can be formalized in rewriting logic [12], [13]. An application is composed of function blocks that exchange messages. With our machinery, it is possible to formally verify such applications for logical defects, which may lead to catastrophic events. Moreover, function blocks can also be mapped to devices, consisting in a System level perspective of the application. Devices communicate through the network.

Security Wrappers: A security wrapper is associated with a device and contains security policies to be obeyed by the device in order to harden the security of the system.

To ensure their integrity, messages may be signed by a device. However, instead of signing all messages in the network, we propose security policies in security wrappers. The policies specify which messages sent have to signed by wrapper's device and specifies which messages received by the wrapper's device are to be accepted only if they are signed by a particular origin device. We formalize security wrappers and the policies described above in rewriting logic.

Symbolic Intruder Model: In order to evaluate the security of an application, we formalize an intruder model in rewriting logic. We propose two models, one where the intruder is capable of injecting messages in the network and another where the intruder can tamper with messages. Following the traditional Dolev-Yao intruder model [6], our intruders cannot, however, construct or tamper with a message that is signed by any other device. Finally, in order to improve automated verification, our intruder can generate symbolic messages, which are constructed using symbols that are resolved during automated verification. The use of symbolic intruder models greatly improves automated verification, allowing to carry out more sophisticated security analysis, *e.g.*, determining which messages may be used by intruders to carry out attacks.

Automated Verification: Finally, we demonstrate the feasibility of our approach by carrying out automated verification using the rewriting tool Maude [5]. We present a number of experiments demonstrating the types of analysis that can be done and investigate the scalability of our approach. Our formalization amounts to more than 2500 lines of Maude code.

Section II describes a motivating example, which will be

[1]There may be other reasons to encrypt a message, *e.g.*, if they contain confidential data.

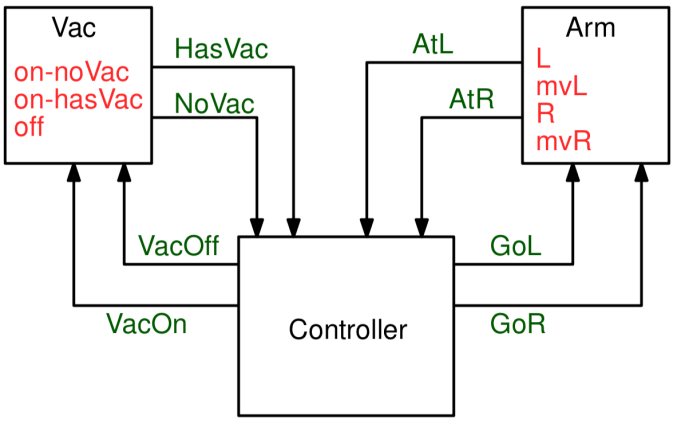

Fig. 1: PnP Functions Blocks and Connections.

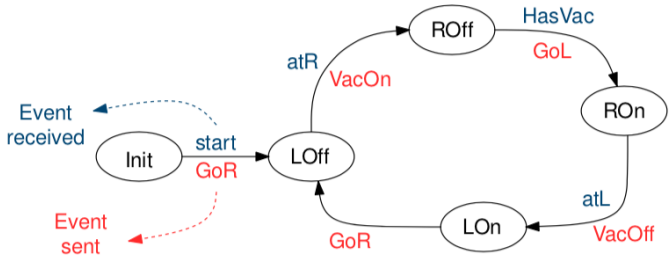

Fig. 2: PnP Controller Specification.

used as running example in the paper. Section III describes the formalization of applications, systems and security wrappers in Rewriting Logic. Section IV describes the formalization of two intruder models, one where the intruder can tamper with messages, and another where the intruder can inject messages. Section V describes how one can carry out security analysis in order to harden the application against such intruders. We also analyse the scalability of our tool. Finally, Section VI concludes by discussing related and future work.

II. Motivating Example

Consider an I4.0 unit, called Pick and Place (PnP),[2] 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 cap 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 the following three function blocks that interact as depicted in Figure 1.

- The Vac function block communicates with the Gripper. It has three states: state on-hasVac denoting that the suction is on and vacuum has been established, indicating that the gripper successfully collected the cap; state on-noVac denoting that the suction is on and vacuum has not been established, indicating that the gripper did not collect the cap; and off denoting that the suction mechanism is off. This function block informs the Controller whether the vacuum has been generated (message HasVac) or not (message NoVac);
- The Arm function block communicates with the PnP arm. It has four states: state L denoting that the arm is at the location to the left, where the cylinder is located; state mvL denoting that the arm is moving to the location to the left; state R denoting that the art is at the location to the right, where the caps are stored; and state mvR denoting that the arm is moving to the location to the right. This function block sends to the Controller the messages atR and atL informing, respectively, that the arm is at the position to the right and to the left;
- The Controller function block implements the logic of the PnP, to place the cap on the cylinder. It is implemented by the Mealy machine depicted in Figure 2. The Controller starts at state Init, moving to state LOff, when receiving the message start, which causes the Controller to send the message GoR to the Arm, in order to position the arm to the position to the right, where the caps are stored. Once the arm reaches the position to the right, denoted by the message atR, the controller sends the message VacOn to Vac to activate the suction. If a vacuum is established, denoted by HasVac, then the Controller sends GoL to the Arm, in order to move the arm to the position to the left. When the arm reaches this position denoted by atL, the controller sends the message VacOff to release the cap (over the cylinder).

Catastrophic Event for PnP For larger scale PnP, the hazard "Unintended Release of Cap" is catastrophic as the cap 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 Arm 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 this catastrophic event by injecting the message VacOff to the Vac when the arm is moving left, that is, at state mvL. This leads to a catastrophic event because the Vac will be at the state off, causing the cap to fall while the arm is moving.

While this attack can be found manually for this small example, it is much better to use automated methods to find such attacks. This is more so as the application size grows and with it the number of cases to consider, as one can miss a case. We will use formal verification methods to carry out these analysis in an automated fashion.

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

III. Applications, Systems and Security Wrappers

The basic functional element of an I4.0 implementation is the function block. We model function blocks formally as finite automata with input and output (Mealy machines). A set of function blocks is composed by linking inputs and outputs to form an application, which represents the intended functionality. In practice groups of one or more function blocks are deployed on (physical) devices which are networked together to implement the application. We assume that internally devices are trustworthy, executing function blocks as intended. But there may be rogue devices/intruders with access to/control of the network. Wrappers with policies concerning allowed message flows can help protect devices and their function blocks.

In the following subsections we describe our formalization of function blocks, applications and systems in the Maude rewriting logic system and show how the resulting models can be executed (prototyping/testing) and formal analyzed.

A. Function blocks and applications

A function block specifies a unit of functionality together with an interface consisting of its input ports and its output ports. It could represent the interface to a sensor or actuator (device driver) such as the Arm (track) and Gripper (Vac), of our running example (Figure 1); or it could serve as coordinator of actions of other function blocks, such as the Controller (Ctl) of our example.

The state of a function block has a unique identifier, an automaton state name , a set of inputs (signals on its input ports) and a set of outputs (signals on its output ports). In Maude a function block state is represented as a term of the form `fb(id,state,inEvs,outEvs)`. Here `inEvs` is a set of inputs of the form `(in :~ ev)` where `in` identifies an input port and `ev` is the event/message arriving on the port. Similarly, `outEvs` is a set of output of the form `(out :~ ev)`.

The semantics of a function block is given by a set of transitions associated with the function block identifier. A transition is a term of the form `tr(stPre,stPost,cond,outEvs)`, where `stPre` is the state before the transition, `stPost` is the state after, `cond` is the condition on the inputs for the transition to fire, and `outEvs` is the resulting outputs. Conditions are boolean combinations of atomic statements of the form `(in is ev)` which is satisfied by any set of inputs that contains `(in :~ ev)`.

Figure 2 shows the state transition diagram for the controller of our running example, PnP. The following transition, formalizes the leftmost arrow in Figure 2. It says that in the initial state (`st("init")`), when `ev("start")` is received, the controller moves to the state `st("LOff")` (the controller assumes the arm is at the left and the vacuum is off initially) and outputs `(outEv("GoR") :~ ev("GoR"))`.

```
tr(st("init"), st("LOff"),
   inEv("start") is ev("start"),
   outEv("GoR") :~ ev("GoR"))
```

The following rewrite rule specifies the effect of firing a transition of a function block `id`:

```
crl[fbExe1]: fb(id,cur,inEvEffs,none)
 => fb(id,postSt,none,ouEvtEffs)
if not (inEvEffs == none)
/\ tr(cur,postSt,cond,ouEvtEffs) trs
   := getTrs(inEvEffs,cur,trsFB(id),none) .
```

`trsFB(id)` is the set of transitions for `id`. The function `getTrs(inEvEffs,cur,trs,none)` returns the elements of `trs` with before state `cur` and condition satisfied by `inEvEffs`.
Thus applying the `fbExe1` rule, using the transition described above, transforms the control function block in the state above the arrow to the state below the arrow.

```
fb(ctl,st("init"),inEv("start"):~ev("start"),none)
=>
fb(ctl,st("LOff"),none,outEv("GoR"):~ev("GoR"))
```

An application is a set of function blocks together with a set of application input and output ports and a set of links connecting the function block ports to each other and to the application ports. Figure 1 shows the structure of our example application, with three function blocks and arrows representing the links. Formally we represent an application structure by an identifier and two maps, one associating the identifier to the application function blocks (their identifiers), and the other associating the identifier to the application links. The state of an application is represented by a term of the form `[[appId,fbs,iMsgs,oMsgs]]` where `appId` is the application identifier, `fbs` is the set of states of the application's function blocks, `iMsgs` and `oMsgs` are sets of input and output messages of the form `{{id,in},ev}` and `{{id,out},ev}`, respectively. The application delivers inputs to its function blocks, allows them to execute and then collects and routes the resulting outputs. This is formalized by the following two rewrite rules:

```
crl[app-exe1]:
 [[id,fbs,iMsgs,none]]
 =>
 [[id,deliverToFBs(fbs,iMsgs),none,none]]
 if not (iMsgs == none) .

crl[app-exe2]:
 [[id,fbs,none,none]]
 =>
 [[id,fbs1,msgs0,msgs1]]
if isDone(fbs)
/\ {fbs1,msgs0,msgs1}
 := extractOutMsgs(id,fbs,...,appLinks(id)) .
```

where `deliverToFBs` is a function that updates the function blocks `fbs` with their corresponding incomming messages in `iMsgs`.

Using these rules, and the function block rules we can "watch" the application run. Here are the first few rewrite steps, starting from the initial state, `appInit`. (The application is named `app` and `fb(vac, st("off"), none, none)` is the initial state of the `vac` function block, `vacInit`.)

```
[[app,fb(vac, st("off"), none, none)
      fb(track, st("L"), none, none)
      fb(ctl, st("init"), none, none),
      {{ctl,inEv("start")},ev("start")},none]]
=>  --- apply app-exe1, deliver message to ctl
[[app,fb(vac, st("off"), none, none)
```

```
       fb(track, st("L"), none, none)
       fb(ctl, st("init"),
              inEv("start"):~ ev("start"),none),
       none, none]]
  => --- apply fbExe1 to the ctl FB
  [[app,fb(vac, st("off"), none, none)
       fb(track, st("L"), none, none)
       fb(ctl, st("LOff"), none,
               outEv("GoR") :~ ev("GoR")),
       none, none]]
  => --- apply app-exe2 to collect outputs
  [[app,fb(vac, st("off"), none, none)
       fb(track, st("L"), none, none)
       fb(ctl, st("LOff"), none, none),
       {{track,inEv("GoR")},ev("GoR")},none]]
  => ---apply app-exe1, deliver message to track
    ...
```

Using search we can as questions such as "can the application progress to a state where the vacuum is on and the arm is moving left?".

```
search [1] appInit =>+
 [[app,vacFB trackFB ctlFB,none,none]] such that
   idOf(vacFB) == vac and idOf(trackFB) == track
   and  getState(vacFB) == st("on")
   and getState(trackFB) == st("mvL") .
```

The answer is yes. In this state there is a message from track to itself to continue moving left and the control is in state `st("ROn")` meaning it received `ev("hasVac")` and is waiting for the track to report arriving at the left.

We can ask if the application can reach a bad state, where the arm is moving left `st("mvL")` and the vacuum is off.

```
search in SCENARIO-VAC-TRACK-CTL : appInit =>*
 app:Application such that
   idOf(vacFB) == vac and idOf(trackFB) == track
   and getState(vacFB) == st("off")
       or getState(vacFB) == st("on-novac")
   and getState(trackFB) == st("mvL") .
```

The answer is no. However it is possible for the vacuum to fail, *i.e.*, to reach state `st("on-novac")`. In our simple app, once this happens, execution stops – there are no available messages.

Suppose the controller transitions for responding to `ev("HasVac")` from `vac` and `ev("atL")` from `arm`

```
    tr(st("ROff"), st("ROn"),
      inEv("HasVac") is ev("HasVac"),
      outEv("GoL") :~ ev("GoL"))
    tr(st("ROn"), st("LOn"),
      inEv("atL") is ev("atL"),
      outEv("VacOff") :~ ev("VacOff"))
```

are replaced by the following variants

```
    tr(st("ROff"), st("ROn"),
      inEv("HasVac") is ev("HasVac"),
      (outEv("GoL") :~ ev("GoL"))
      (outEv("VacOff") :~ ev("VacOff")) )
    tr(st("ROn"), st("LOn"),
      inEv("atL") is ev("atL"),none)
```

where the controller (mistakenly) sends the `ev("GoL")` (to `arm`) and `ev("VacOff")` (to `vac`) simultaneously rather than waiting until the arm has confirmed that it has reached the left. Then when we ask if a bad state is reachable there is a solution.

```
   app:Application --> [[badapp,
   fb(vac, st("off"), none,
           outEv("NoVac") :~ ev("NoVac"))
   fb(track, st("mvL"), none,
             outEv("GoL1") :~ ev("GoL1"))
   fb(ctl, st("ROn"), none, none),none,none]]
```

That is, one can use our machinery to identify logical errors due to the bad controller.

B. Systems

An application is deployed by assigning the application's function blocks to run on specific devices. The devices are networked together forming a system. Formally, a device has an identifier and an associated set of function blocks. The state of a device is represented by a term of the form `[devId,fbs]` where `devId` is the device identifier and `fbs` is the set of states of the function blocks associated to (deployed on) the device. A system is a collection of devices networked together according the links of the underlying application. A system is represented formally by an identifier and an association of the identifier to the devices (their identifiers). The links, device, and system ports are inferred from the links associated to the underlying application.

The state of a system is represented by a term of the form `[sysId,appId,devs,iMsgs,oMsgs]`. For example, we deploy our PnP application using three devices (named `dev1`, `dev2`, and `dev3`), assigning `vac` to `dev1`, `track` to `dev2`, and `ctl` to `dev3`. The initial state, `sysInit`, of the deployed system, which we name `sys` is the term

```
  [sys,app,devInit1 devInit2 devInit3,msg0,none]
```

where `devInit1`, the initial state of device `dev1` is `[dev1,vacInit]` and similarly for `devInit2` and `devInit3`. `msg0` is the start message.

```
  msg0 = {portSysIn,{dev3,
          {ctl,inEv("start")}}, ev("start")}
```

The rules for executing a system have the same overall structure as the rules for executing an application. There is a rule `sys-exe1` that delivers the system level input messages to the devices that in turn delivers them to the deployed function blocks, thus enabling the function block execution rules. There is also a rule `sys-exe2` that applies when all the devices have processed their messages (all the function blocks are done). It gathers the resulting outputs and splits them into messages for devices in the system and messages for entities outside the system. Using these rules we can carry out steps corresponding to the application execution the system level starting from `sysInit`. In three steps we arrive at

```
[sys,app,
  [dev1,fb(vac, st("off"), none, none)]
  [dev2,fb(track,st("L"),none,none]
  [dev3,fb(ctl, st("LOff"), none,none)],
  {dev2, {track,inEv("GoR")}},ev("GoR")},none]
```

The point is that the deployed system behaves like the application, which is what is desired.

Suppose an intruder can inject a single message when the system starts. Already this can cause trouble. For example, an early `ev("VacOn")` message can cause the vacuum to turn on while the arm is moving right, and thus not replying to the later `ev("VacOn")` message from the controller, causing production to stop.

```
search [1] [sys,app,
  [dev1,fb(vac, st("off"), none, none)]
  [dev2,fb(track, st("L"), none, none)]
  [dev3,fb(ctl, st("init"),none,none)],
  {portSysIn,{dev1,{vac,
              inEv("VacOn")}},ev("VacOn")}
  {portSysIn,{dev3,{ctl,
              inEv("start")}},ev("start")},none]
  =>+ [sys,app,
  [dev1,fb1:FB][dev2,fb2:FB][dev3,fb3:FB],
        none,none] such that
   idOf(fb1:FB) == vac and idOf(fb2:FB) == track
   and getState(fb1:FB) == st("on")
   and getState(fb2:FB) == st("mvR") .
Solution 1 (state 11)
  fb1:FB --> fb(vac, st("on"), none, none)
  fb2:FB --> fb(track, st("mvR"), none,
               outEv("GoR1") :~ ev("GoR1"))
  fb3:FB --> fb(ctl, st("LOff"),
             inEv("HasVac"):~ ev("HasVac"),none)
```

C. SecurityWrappers

How can networked systems be protected against intruders that can access the network to interfere with communications? One idea is to 'wrap' devices with a security policy layer that allows each device to control message flow across its border. The idea is that messages in flows that need to be protected are signed and messages with signatures that do not check are dropped. Formally a wrapped device is a term of the form `[dev,iPol,oPol]` where `dev` is the device being wrapped, `iPol`/`oPol` are input/output policies.

To reason about wrapper effects, we model an intruder as an intruder device `[iid | msgs]` consisting of an identifier `iid` and a (finite) set of messages `msgs` to inject. Now a system state contains wrapped devices and also intruder devices: `[sys,app,wdevs,idevs,msgs0,msgs1]`. The execution rules for systems are adapted to deliver messages to wrapped devices by checking input policies before delivery, and to use output policies to sign messages as needed when collecting output messages. There is an additional rule that selects an intruder message and injects it into the system message pool. To see the ideas in action, suppose the intruder injects a message, `ev("VacOff")`, from `dev3` to `vac` on `dev1`.

```
imsg ={{dev3,{ctl,outEv("VacOff")}},{dev1,{vac,
               inEv("VacOff")}},ev("VacOff")} .
idev = [iid,imsg] .
```

To protect `"VacOff"` messages from the controller on `dev3` to the vacuum on `dev1`, the `dev3` wrapper is given the output policy `[outEv("VacOff")]`, meaning messages output on port `outEv("VacOff")` must be signed; and the `dev1` wrapper is given the input policy `[inEv("VacOff"),dev3]`, meaning messages input on port `inEv("VacOff")` must be checked for signature by `dev3`.

Let `wsysIntruderNoPol` be the system with the above intruder and wrapped devices with no policies, and let `wsysIntruderPol` be the result of adding the policies discussed above to `dev1` and `dev3`. Then using search we can confirm that in the absence of protective policies the intruder can succeed in driving the system to a bad state where the vacuum is off (`st("off")`) and the track is moving left ((st("mvL"), presumably having dropped what it was carrying midway.

```
search wsysIntruderNoPol =>* wsys
        such that badState(wsys) .
Solution 1 (state 98)
wsys -->[sys,app,
 [[dev1,fb(vac, st("off"),none,
       outEv("NoVac"):~ev("NoVac"))],none,none]
 [[dev2,fb(track, st("mvL"),none,
       outEv("GoL1") :~ ev("GoL1"))],none,none]
 [[dev3,fb(ctl,st("ROn"),none,none)],none,none],
   none,none,none,nil]
```

And with policies in place, the bad state can not be reached.

```
search wsysIntruderPol =>* wsys
     such that  badState(wsys) .
No solution.
```

IV. Symbolic Intruder Model

Our threat model is inspired by analysis carried out in the BSI report for OPC-UA protocol [8]. In particular, we assume an intruder that can inject and tamper with messages. However, we also assume that signatures are perfect, *i.e.*, constructed using keys that are not known to the intruder nor can they be guessed. Therefore, intruders cannot tamper with signed messages nor inject signed messages.

As formal verification traverses all states of the application, it suffers from the *well-known* "State-space explosion problem", as the number of states increases exponentially with the size of the system. In order to mitigate this problem, we rely on two established techniques in formal verification: symbolic verification [2] and bounded verification [3].

A. Symbolic Verification

Instead of guessing which (concrete) message to inject or how to tamper with a message in the network, we post-pone this decision to when it has to be made. This is accomplished by using symbolic messages. The use of symbolic messages improves automated verification by reducing the search space and also allows for more types of analysis to be carried out. We illustrate the use of symbolic messages with the following example.

Example Consider two function blocks `FB1` and `FB2`, where `FB1` is expecting a message with the event `ev1` to make a transition, while `FB2` is expecting a message with `ev2` to make a transition.

An attacker that is injecting messages into the network has as goal to lead the system to a bad state. This means that he

attempts to activate function block transitions by sending events that are expected by these function blocks.

Therefore, in the configuration above, the intruder can inject a message to function block `FB1` with event `ev1` or a message to function block `FB2` with event `ev2`, that is, messages of the form:

`{src,FB1,ev1}` or `{src,FB2,ev2}`

where `src` is some origin identification token.

There are two problems of using such concrete messages for automated verification. The first problem is that one needs to provide as input these messages to the automated verification tool, that is, one needs to enumerate all possible messages that could activate function blocks. The second problem is that one needs to carry out the same verification effort for each one of these messages.

By using symbolic messages, instead, we only need to provide the following symbolic message as input to the automated verification procedure:

sym-msg = `{src,sid,sev}`

where `sid` is a symbol which shall be instantiated with a function block identifier, and `sev` is a symbol which shall be instantiated with a concrete event. The automated verification procedure will instantiate these symbols during search by solving using constraint solving techniques.[3]

In the example above, the verification procedure will attempt to find an instance for the symbols in sym-msg, so that it can be used to trigger either a transition for `FB1` or a transition for `FB2`. This leads to two possible solutions:

$$sol1 = \{\texttt{sid} \rightarrow \texttt{FB1}, \texttt{sev} \rightarrow \texttt{ev1}\}$$
$$sol2 = \{\texttt{sid} \rightarrow \texttt{FB2}, \texttt{sev} \rightarrow \texttt{ev2}\}$$

The solution $sol1$ instantiates the symbol `sid` to `FB1` and `sev` to `ev1` and *mutandis mutatis* for the second solution $sol2$ and `FB2` and `ev2`. These solutions when applied to the symbolic message sym-msg correspond exactly to the concrete messages shown above.

As illustrated by the example above, the use of symbolic messages enables security analysis to be carried out, such as, "Which messages exactly the intruder can inject that can lead to bad states and which do not lead to a bad state?". As we illustrate in Section V, from the answer to this question, a security engineer can harden the system by deploying security wrappers adequately, namely, enforcing the encrypting or signing of the messages that lead to bad states when injected by the intruder.

B. Bounded Intruders

The second approach for dealing with the "State-Space Explosion Problem" is to constraint the verification problem by assuming some bounds on the state-space. This means that verification will be only carried out on the state-space specified by this bound. The exact bound depends on the on parameters, such as, system size, available computational power.

[3]We also implemented such techniques for our verification prototype. Its details are out of the scope of this paper.

For security verification, we bound the intruder. Our bounded intruder can only inject or tamper with a bounded number of messages. The bound can be configured by the user. The greater the bound, the greater will be the power of the intruder. This means that a more complicated attack involving a sequence of injected messages or messages that have been tampered with may be found, at the cost of performance, as a greater number of states have to be traversed.

We now show how to specify in Rewriting Logic these two types of intruders: the intruder capable of injecting messages and the intruder capable of tampering with messages. In order to do so, we extend the definition of systems (Section III-B) to include the two types of intruders, as follows:

- `[{ sys }, inject(msgs)]`, specifying an intruder that can inject at any moment in the system `sys` any one of the messages in `msgs`;
- `[{ sys }, tamper(msgs)]`, specifying an intruder that can tamper with a message sent by a device in the system in `sys`, modifying it to be one of the messages in `msgs`.

We formalize the semantics of these intruders by rewrite rules. The following rule `intruder-injection` rule specifies that the intruder in possession of message `msg` can inject this message in a system `sysId`. This is accomplished by moving `msg` to the slot with the input messages `iMsg`.

```
rl[intruder-injection]:
[{[sysId,appId,devs,iMsgs,oMsgs]},
   inject(msg msgs) Intrs]
=>
[{[sysId,appId,devs,msg iMsgs,oMsgs]},
   inject(msgs) Intrs] .
```

The following rule `intruder-tamper` specifies that an intruder in possession of message `msg` can tamper with the message `msg0` to be processed by system `sysId` to become the message `msg`. This is accomplished by replacing the message `msg0` by `msg`.

```
rl[intruder-tamper]:
[{[sysId,appId,devs,msg0 iMsgs,oMsgs]},
   tamper(msg msgs) Intrs]
=>
[tamper,
{[sysId,appId,devs,msg iMsgs,oMsgs]},
   tamper(msgs) Intrs] .
```

In a similar fashion, it is possible to construct intruder models where the intruder is capable to block messages. This can be accomplished by simply deleting a message from the input or output messages.

Furthermore, the variable `Intrs` allow us to combine intruders with different capabilities. For example, the configuration

```
[{ sys }, inject(msgs0) tamper(msgs1) ]
```

includes two intruders: one that can inject messages `msgs0` and another that can tamper with messages to become like one of the messages in `msgs1`.

V. Automated Verification

We will use our motivating example described in Section II to illustrate how one can use our formal models to carry out security analysis.

Security Analysis without Security Policies

Consider the systems with the deployed Vac, Arm, and Controller function blocks deployed in their own devices `dev1`, `dev2` and `dev3`, respectively, belonging to the system `sys`. We assume, first, that there are no security policies in the devices' security wrappers.

We can now analyse how this system can be attacked by an intruder that can inject a single message. This is done by allowing the intruder to inject a symbolic message of the form:

```
symmsg = {src,{sdev,{sfb,sin}},sev}
```

where `src`, `sdev`, `sfb`, `sin`, and `sev` are symbols for, respectively, source of the message, destination device, function block to be sent to, the in port of the function block, and the event to be delivered.

The initial configuration is shown below

```
confInit = [ { sys }, inject(symmsg)]
```

where the intruder can inject the symbolic message `symmsg` to the system containing the application with the Vac, Arm, and Controller specifications.

We can now use the following search command in Maude to find whether it is possible for the application to reach an arbitrary state `sys1` that is a bad state:

```
search confInit =>* [ { sys1 }, I] such
        that badstate(sys1) .
```

As described in Section IV, while searching, our machinery keeps track of the values for the symbols in the symbolic message `symmsg`. If a bad state can be reached, the values are returned.

For the search command above, Maude finds 18 solutions, *i.e.*, ways the intruder can lead the system to a bad state. An example of a solution for the symbols in `symmsg` is as follows:

```
sev :~ ev("atL")   sdev :~ dev3
sfb :~ ctl         sin :~ inEv("atL")
```

which specifies that the message injected contains the event `ev("atL")`, to activate the `ctl` function block at device `dev3`.

Many of the 18 solutions are redundant as the intruder uses the same injected message, but at different points in the execution of the application. If we consolidate the injected messages in these solutions, there are only 4 messages that the intruder could use to lead to a bad state. From the completeness of the search tool, *this means that protecting against these 4 messages is enough to harden against such intruders.*

Systematically Configuring Security Wrappers

We harden the system by configuring the security policies of the devices' security wrappers. For example, in order to protect the system against an intruder injecting the message with the event `atL` shown above, we add the following out policy to the device, `dev2`, which contains the `track` function block and generates the message with event `atL`:

```
[o : outEv("atL")]
```

This means that this message will be signed by device `dev2`. Similarly, we add the in policy to the security wrapper the device `dev3` as follows:

```
[i : inEv("atL"), dev2]
```

specifying that the messages arriving at `inEv("atL")` have to be signed by device `dev2`.

Let `configInitAtL` be the configuration obtained by adding the two policies above to the system `sys`. If we perform the search:

```
search configInitAtL =>* [ { sys1 }, I]
       such that badstate(sys1) .
```

Maude returns 17 solutions, in contrast to 18 with the previous search. Among the attacks, none of them use the message with `ev("atL")`.

By adding appropriate policies for the remaining 3 possible messages that the intruder can inject, we can show that the system is resistant to these intruders.

State Space	Function Blocks	Execution Time
23	3	0.003 s
529	6	0.123 s
12167	9	3.7 s
279841	12	164.7 s
–	15	–

Fig. 3: Scalability Results without Symmetry Optimizations. We interrupted the experiment with 12 function blocks after 1 hour.

A. Scalability Experiments

Each one of the analyses carried out above took less than 10 ms in a 2.2 GHz Intel Core i7 with 16 GB of RAM. We describe some preliminary analysis on the scalability of our tool. To do so, we systematically built scenarios by running several instances of the Pick and Place application (Section II) in parallel. Table 3 summarizes our experiments. Our machinery is able to verify up-to 4 instances of the Pick and Place running in parallel, *i.e.*, 12 Function Blocks, in less than 165 seconds traversing a total of 279841 states. When adding an additional instance of the Pick and Place, however, our machinery was not able to traverse all states in 1 hour.

We can, however, further improve these results by exploiting the properties of the scenario. In particular, we know that each one of the Pick and Place are running independently, *i.e.*, they do not communicate among themselves. One way to exploit this fact is by exploring symmetry. Two configurations are equivalent if one can be turned into the other by shuffling the

State Space	Function Blocks	Execution Time
23	3	0.003 s
276	6	0.067 s
2300	9	0.650 s
14950	12	5.4 s
376740	15	311.2 s
1560780	18	3308.2 s
–	21	–

Fig. 4: Scalability Results with Symmetry Optimizations. We interrupted the experiment with 21 function blocks after 1 hour.

Pick and Place instance. For example, consider the systems, S_1 and S_2, with the following sequence of Pick and Place instances:

S_1 : `[PnP(off,ROff,L),PnP(on-hasVac,ROn,mvL)]`
S_2 : `[PnP(on-hasVac,ROn,mvL),PnP(off,ROff,L)]`

where `PnP(st1,st2,st3)` denotes a Pick and Place, where the Vac, Controller, Arm are in, respectively, the states `st1`, `st2`, `st3`. For verification, the systems S_1 and S_2 can be considered equivalent, as by exchanging the order of the Pick and Places in S_1 we obtain S_2 and vice-versa.

We implemented in Maude such optimization, by using the built-in Maude equational theories. We can specify that the system is a *multiset* of Pick and Place applications, instead of a *list* of applications. This improved considerably the scalability results as can be seen in Table 4. We were able to verify systems with up-to 6 instances of the Pick and Place.

VI. Conclusions and Related Work

In this paper we presented a formal model of I4.0 application designs, and showed how such models can be executed and analyzed using search to better understand possible behaviors. We showed how the abstract application model can be refined to a model of the application deployed on networked devices. We proposed a weak attacker model and showed how search and symbolic execution can find all the points of attack. Using this information the deployed system can be protected by wrapping devices with a policy layer that uses signatures to ensure intended message flows. We believe that our approach 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 refining the network model to one with multiple subnets and switches, adding timing and modeling constraints induced by use of the TSN network protocol, relating security to safety, and further automation of analysis techniques.

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 [8], and the ENISA study on good practices for IoT security [7].

OPC Unified Architecture (OPC UA) is a standard for networking for Industry 4.0. It covers different levels from the control through to the corporate level in a manufacturer-independent manner. OPC UA is equipped with integrated security functionality to secure the 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 a much work on modeling cyber physical systems and cyberphysical security (see [11] for recent review), much of it is based on simulation and testing. The formal modeling work focuses on general CPS not on the issues specific to I4.0 type situations.

An exception is the work of Lanotte et.al [9] that propose a hybrid model of cyber and physical systems and associated model of cyber-physical attacks. Attacks are classified according to target device(s) and timing characteristics (initiation and duration). 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 and tampering attacks. We feel that 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.

References

[1] Cyberattack on a German steel-mill, 2016. Available at https://www.sentryo.net/cyberattack-on-a-german-steel-mill/.

[2] D. Basin and L. V. Sebastian Mödersheim. OFMC: A symbolic model checker for security protocols. *Interational Journal of Information Security*, 2004.

[3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[4] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. 1997.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[6] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[7] ENSIA. Good practices for security of internet of things in the context of smart manufacturing, 2018.

[8] M. Fiat and et.al. OPC UA security analysis, 2017.

[9] R. Lanotte, M. Merro, R. Muradore, and L. Vigano. A formal approach to cyber-physical attacks. In *30th IEEE Computer Security Foundations Symposium*, pages 436–450. IEEE Computer Society, 2017.

[10] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS*, pages 147–166, 1996.

[11] Y. Z. Lun, A. D'Innocenzo, I. Malavolta, and M. D. D. Benedetto. Cyber-physical systems security: a systematic mapping study. *CoRR*, abs/1605.09641, 2016.

[12] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[13] J. Meseguer. Twenty years of rewriting logic. *J. Algebraic and Logic Programming*, 81:721–781, 2012.