Automated Construction of Security Integrity Wrappers for Industry 4.0 Applications

Vivek Nigam[a,c], Carolyn Talcott[b]

[a]fortiss, GmbH, Munich, DE.
[b]SRI International, CA, USA.
[c]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 communication protocols such as OPC-UA is to sign and/or encrypt messages. However, given the limited resources of devices and the high performance requirements of I4.0 applications, 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 applications on the given set of (network) 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.

In addition, we report on a series of experiments checking for attacks by a bounded intruder against variations on a Pick-n-Place application, investigating the effect of increasing bounds or increasing application size and further minimizing the number of messages that must be signed.

Key words: Industry 4.0, bounded intruder, function block, theory transformation, security, safety, verification, policy, Maude, rewriting logic.

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

Email addresses: `nigam@fortiss.org` (Vivek Nigam), `clt@cs.stanford.edu` (Carolyn Talcott)

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 report from the *Bundesamt für Sicherheit in der Informationstechnik* (BSI) on the security of Open Platform Communication Unified Architecture (OPC UA) (machine to machine communication protocol for industrial automation) [12], 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.

Cryptographic signing provides message integrity thus enabling systems to defend against tampering and injection attacks. Message signing, however, is a computationally expensive operation[2]. Moreover, many I4.0 applications, like motion control, require the movement of components to synchronize in a microsecond range[3]. To achieve both performance and security requirements, more powerful (and thus expensive) hardware may be required, *e.g.*, CPUs that have built-in hardware encryption. Therefore, instead of requiring all messages to be signed, 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.

To answer this question, we use formal methods to reason about I4.0 specifications developed using Model-based System Engineering approach (MBSE). MBSE has been advocated for the development of embedded systems also for I4.0 applications through the standard IEC 61499 [28, 27]. Following this approach, embedded systems are developed by decomposing and implementing system functions into a collection of communicating function blocks. A function block is an executable model of a function between inputs variables and outputs variables. The behavior of function blocks are specified by using executable models, such as state machines. Existing tools, such as 4diac[4] and AutoFOCUS[5], support the development of I4.0 systems, including the specification of function blocks using state machines, automated code generation from these specifications, and deployment into devices. However, there has been little focus on the formal analyses of such applications, in particular, on how attacks can lead to harm and how to avoid such attacks.

This paper presents a formal framework for specifying I4.0 applications following this MBSE approach and analyzing safety and security properties using Maude [8]. The engineering development process from application design and testing to systems deployment is captured by theory transformations with associated theorems showing

[1] https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/Lagebericht2014.pdf?__blob=publicationFile

[2] https://medium.com/logos-network/benchmarking-hash-and-signature-algorithms-6079735ce05

[3] http://www.hit.bme.hu/~jakab/edu/litr/TimeSensNet/TSN-Time-Sensitive-Networking-White-Paper.pdf

[4] https://www.eclipse.org/4diac/

[5] https://www.fortiss.org/en/results/software/autofocus-3

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 [8]. An application is composed of interacting state transition machines which, following the IEC 61499 standard [27], we call function blocks.
- **Bounded Symbolic[6] Intruder Model:** The security verification problems that we consider are undecidable in general, but decidable PSPACE-complete if we bound the number of messages that the intruder can inject or tamper [1]. Using the number of messages an intruder can inject/tamper can be used as a metric for the level of security of I4.0 applications. The greater the number of messages that intruder can inject or tamper, the greater is his attack power. Indeed, any attack using n messages can be performed by an attacker capable of injecting/tampering $m \geq n$ messages. Therefore, showing that an intruder with n messages is not able to carry out an attack provides a form of quantitative assurance on the level of security of the system.

  We use this fact 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 message that is not encrypted, 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 resulting 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. Such applications models are deployed into a system architecture, composed by hardware units and communication mediums. Accordingly, 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 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 [7]. Here it is used to secure message integrity between devices using signing. Since signing is expensive, it is important to minimize message

[6] Throughout this paper we use the terminology *symbolic* to indicate the use of symbols (variables) in representations of model entities such as messages, executions, and analyses. This allows compact representation of the state space and enables formal analysis of non-trivial systems.

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.

- **Minimal protection set.** In the case that there are multiple attacks it is only necessary to protect one message from each attack, not all messages from all attacks. We introduce the notion of *minimal protection set* and present an algorithm for computing such sets. Thus further improving the efficiency of wrapping policies.

We have implemented the framework and carried out a number of experiments demonstrating the 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 [21] 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. In an another co-joint paper [1], we investigated the complexity of security verification problems involving bounded intruders and extends the experiments with four selected scenarios constructed from the example described in Section 2.1. This paper is an extension of our WRLA20 workshop paper [22]. The new contributions in the workshop paper included the theorems and proofs, implementation of the deployment and wrapping functions, and a simplified version of the symbolic intruder model. Moreover, Section 4.2 defines the new concept of minimal protection set that contains the messages that are enough to be signed by security wrappers to ensure security (under the assumptions on the given intruder model). We propose an algorithm to compute this minimal set and apply it to the four examples described in Section 4. With this new concepts, we refine security wrappers reducing the number of messages to be encrypted as compared to our previous work [22] while still ensuring security. Depending on the scenario this reduction can be of more than 50% of messages when compared to the approach in [22].

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 describes how our machinery supports automated reasoning. It also shows how to improve the efficiency of the security wrapper. Section 5 discusses related work, and Section 6 concludes with ideas for future work.

2. Overview

Threat Model. We assume that devices have their pair of secret and public keys. 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

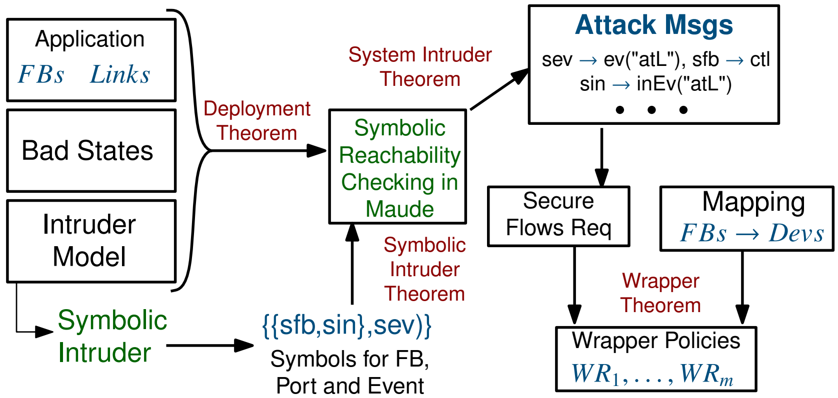

Figure 1: Methodology Overview

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 [12].

To protect communications between function blocks on different devices we use the idea of formal wrapper [7] 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 [18]. 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 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. FBs are finite state machines that either get stuck or are periodic. Therefore, since there is a bounded number of FBs in an application, the overall state-space is finite. This means that, due to Maude's in built memoization, search always terminate provided there is enough memory. We extract the critical events, *i.e.*, injected message sets leading to a bad state, from attack traces of a given an application in a symbolic intruder environment. This is done by using Maude's reflection features enabling one to manipulate with search traces.

Deploying an application can be seen as a theory transformation [20]. 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 configurations 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 may inject a given 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. The challenge is to ensure the soundness and completeness of symbolic search. That is, an execution using the symbolic model corresponds to at least one execution using the concrete model and vice-versa.
- **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 3.1) 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. The key to this result is the soundness and completeness of the symbolic match generation.

Deployment Theorem (Theorem 3.3) 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 3.5) 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 (concrete or symbolic) 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 3.7) 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.

Remark:. The formal machinery developed in this paper is to enable early verification of applications by identifying (minimal) requirements on which messages shall be protected by means of security wrappers. These requirements shall be used during the development in implementation decisions, such as the computational power of devices, or whether specialized Hardware Security Modules with Hardware Encryption shall be used to increase the efficiency of encryption. For example, it has been shown that more expensive CPUs with in built hardware encryption are up to six times faster than CPUs without in built hardware encryption.[7] Furthermore, these requirements may guide the deployment of FBs on devices if one assumes that the connection between FBs in the same device are implicitly secure. Indeed, these requirements can be used together with design space exploration [25] techniques.

2.1. Example

Consider an I4.0 unit, called Pick and Place (PnP),[8] 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.

Following the IEC 61499 standard [27]. Model-Based System Engineering (MBSE) tools, such as 4diac[9], specify such Industry 4.0 by using function blocks. A function block is an executable specification, typically a Mealy machine, and an application is a collection of communicating function blocks. From these specifications, existing MBSE tools generate code that can be deployed in the devices used in factory.

[7] https://www.tomshardware.com/reviews/clarkdale-aes-ni-encryption,2538-5.html

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

[9] https://www.eclipse.org/4diac/

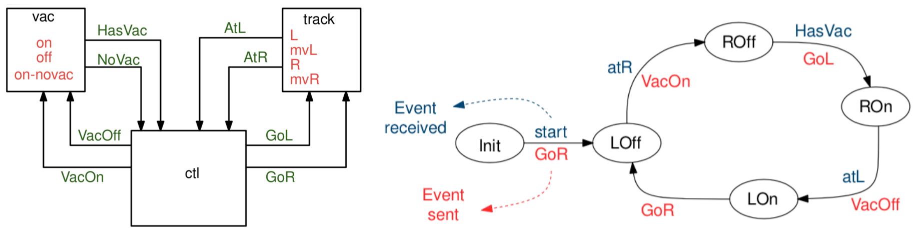

Figure 2: PnP Function Blocks, ctl, vac, and track. The internal states of vac and track are shown in their corresponding boxes and their transitions are elided. The complete specification is shown in the finite machine to the right.

An application implementing the PnP logic has three function blocks (FBs) 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 example, after starting, it sends the message GoR to the track that then moves to the right-most position (state mvR) where the caps are to be picked. When the track reaches this position, it informs the controller by sending the message AtR. ctl 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 ctl. ctl then sends GoL to the track. This causes the track to move to the left-most position (state mvL) where the cylinder is located and on which the cap has to be placed. The track sends the message AtL. ctl then sends the message VacOff to the vac to turn off the vacuum mechanism causing the cap to be placed over the cylinder.

As illustrated by the PnP execution above, the execution of applications is assumed to be synchronous. That is, a global execution cycle, also called hypercycle, only ends when all FBs have executed their steps. This is normally achieved by using a time synchronization protocol. In particular, FBs execute internally generating events which are then communicated to other FBs to be processed in the next global cycle. Finally, we also point out that FBs may also exchange data and not only events. These data communication channels may contain sensitive data that shall be protected. In this paper, we do not consider such attacks, but only attacks from the manipulation of event links. Notice that these events do not possess any complex structure being simple constants.

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)[10], 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

[10] https://psas.scripts.mit.edu/home/get_file.php?name=STPA_handbook.pdf

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 Reachability-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 reachability-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.

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 characterization a FB class. To allow for multiple occurrences of a given FB class in an appliication the state of a FB has both an instance and a class identifier. The events communicated among function blocks do not have any complex structure being constants.

The Maude representation of a 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)`, `(iEvEffs : ieffs)`, `(oEvEffs : oeffs)`, and `(ticked : b)`, with `state`, `iEvEffs`, `oEvEffs`, `ticked` being the attribute tags, `st` the current state, `ieffs` a set of signals/events to be processed (incomming effects), `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 of the form `o :~ ev` specifying that the event `ev` is to sent on the output port `o`. A condition is a boolean combination of primitive conditions `(in is ev)` specifying a particular event (`ev`) at input `in`. `tr(st0,st1,cond,oeffs)` is a transition 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 identifier `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. 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 $\texttt{ieffs} = \{(\texttt{in}_i \rhd \texttt{ev}_i) | 1 \leq i \leq k\}$ 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 $\texttt{css}(\texttt{in}_i)$ 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`. Here is the idea. Let $\texttt{condF} = \texttt{DNF}(\texttt{NNF}(\texttt{cond}))$ be the disjunctive negative normal form of `cond`. `condF` is a disjuction of clauses (conjunctions) whose elements have the form `(in is ev)` or `not(in is ev)`. Since the set of possible values of `ev` is finite, call it `allE`, we allow the second component of `(in is ev)` to be a set and replace `not(in is ev)` by `(in is allE-ev)`. Then, for each clause we replace the set of constraints for a given input, `in`, by the intersection of the associated event sets. This preserves satisfiability since the I4.0 model delivers at most one event on each input when a transition fires, so a conjunction demanding two or more events on an input is not satisfiable. Next remove any clauses containing a conjunct `(in is empty)` as they are unsatisfiable. The remaining disjuncts are converted to maps such that `(in is evs)` maps `in` to the set `evs`. This is the set of constraint sets `condToCSet(cond)`. For the `vac` example, the CSet

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

maps `inEv("VacOn")` to the singleton `ev("VacOn")`. As another example, a condition that captures the constraint that `track` requires messages from both `ctl` and `vac` to move left is

```
(inEv("GoL") is ev("GoL")) and
((inEv("HasVac") is ev("HasVac")) or
 (inEv("noVac") is ev("noVac")))
```

Its disjunctive normal form is

```
(inEv("GoL") is ev("GoL")) and (inEv("HasVac") is ev("HasVac"))
or
(inEv("GoL") is ev("GoL")) and (inEv("noVac") is ev("noVac"))
```

Thus the result of applying `condToCSet` to this condition is two `CSets`: one maps `inEv("GoL")` to `ev("GoL")` and `(inEv("HasVac")` to `ev("HasVac")`; and the other maps `inEv("GoL")` to `ev("GoL")` and `(inEv("noVac")` to `ev("noVac")`.

The function `condToCSet` is lifted to transitions by the function

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

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.[11] 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 following two rules specify the nomimal semantics of I4.0 applications, *i.e.*, without the presence of intruders. The first rule, `[app-exe1]`, specifies the internal execution of a FB, while the second rule, `[app-exe2]`, specifies the end of a global execution when no FB can make an internal execution.

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`, *i.e.*, `genSol1` is sound and complete. Note that this could be the empty set of substitutions if there are no solutions. In the case of concrete messages, *i.e.*, not containing symbols, the function `genSol1` just returns an set consisting of the empty substitution if `emsgs0` satisfies `css`, while it returns an empty set of substitutions if `emsgs0` fails to satisfy `css`. `genSol1`, equationally defined in Maude, directly implements

[11] Otherwise, if an input port of a FB receives two different incoming links, the execution semantics of the FB is not well defined as it is not clear which incomming event from which incomming link.

the notion of satisfaction described above, where CSets and symbolic transitions are introduced. 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 ;
    attrs]
=>
  [appId |
    fbs : fbs2 ;
    iEMsgs : emsgs0 ;
    oEMsgs : (oemsgs emsgs1) ;
    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}}`, where `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 a correspondence `[As,smsgs]` $\sim$ `[Ac,cmsgs,n]` between symbolic and concrete intruder states as follows:
`[As,smsgs]` $\sim$ `[Ac,cmsgs,n]` holds only if

- `size(smsgs)` $= n$,
- `As.fbs` $=$ `Ac.fbs`, and
- `(As.iEMsgs)[ssbs]` $=$ `Ac.iEMsgs`

for some symbol substitution `ssbs`.[12] 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 3.1. 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

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

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

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

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

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

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

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, derived from the soundeness of `genSol1`,

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

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

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

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

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

For rl_k an instance of `[app-exe2]` or the intruder rule, TrC extends to a corresponding trace because `[app-exe2]` is only applied when there is no solution which is preserved by the correspondence. Similarly, the symbolic execution of the intruder rule is enabled if the set of intruder messages is not empty. In this case, the bound of messages the intruder can inject in the concrete case will not be exceeded.

Conversely, let

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

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

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

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

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

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 3.2. 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) .
```

Since the symbols in the symbolic intruder messages are unique, the concrete messages used by the intruder to carry out an attack can be determined from the final substitution.

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 unique 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 3.3. 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 3.4. `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.

3.5.1. 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 3.5. 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 3.6. 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 3.7. 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

wAC = [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` $rl_0 \ldots rl_k$ $\mathtt{wAC}_{k+1}$ be a witness execution where `badState` holds of $\mathtt{wAC}_{k+1}$. By the assumption on `A` from part 1, at least one intruder message must have been delivered.

Let $\{\mathtt{emsg}_1 \ldots \mathtt{emsg}_l\}$ 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

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

and by the *Deploy Intruder Theorem* there is a 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. Towards Automated Reasoning

In the preceding sections we developed theory transformations that allow security analysis of Industry 4.0 systems to be carried out at the application level and provide automatic generation of policies and enforcement wrappers to protect against considered attacks.

Section 4.1 reports on some proof of concept experiments described in our previous work [1]. In that work, we also showed that while the security problem of determining whether an intruder can lead a system to a bad state is undecidable when considering an unbounded intruder, such as the Dolev-Yao intruder [9], the problem is PSPACE-complete when considering a bounded intruder as we do here. Despite the high complexity, the proof of concept experiments demonstrate the feasibility of automated verification in realistic size systems.

Sections 4.2 and 4.3 introduce machinery that refines the analysis of the automated reasoning leading to the need of less messages to be protected through message signing. In particular, Section 4.2 describes the refinement analysis problems addressed, and Section 4.3 introduces the formal machinery with our solution. We show its effectiveness by revisiting the experimental results discussed in Section 4.

4.1. Automated Reasoning

In this section we report on a series of experiments carried out in our previous work [1]. We investigated the effect of varying the intruder bound and increasing the size of the application. The experiments are based on the Maude I4.0 formalization described in [21, 22]. The scenarios analyzed are variants of a Pick-n-Place application, as described below. We use these scenarios to illustrate the analysis refinement algorithm described in Sections 4.2 and 4.3.

- (PnP) This scenario is the one described in Section 2.1.
- (2PnP) This scenario is depicted in Figure 3. It is a an application containing two instances of PnP and a coordinator that ensures that the start of the cycle of each

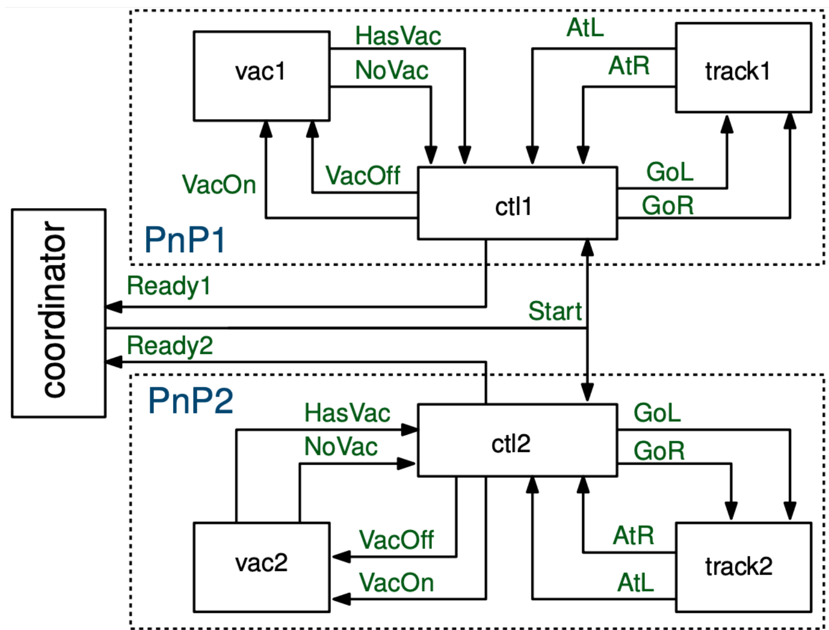

Figure 3: Illustration of the 2PnP application. This set-up has been described in [1].

instance of PnP happens at the same time, *i.e.*, the instance controllers send the initiating GoR at the same time.

- (PnP-2Msgs) This scenario modifies the logic of the PnP so that the track at the right (where the caps are) waits for two signals to head left (where the cap has to be placed): GoL from ctl and HasVac/NoVac from vac (to confirm that vac has received and processed the VacOn message); and when vac is on it requires two signals to turn off: VacOff from ctl and AtL from track. Intuitively, this means that the intruder would need at least two actions to lead this system to a bad state.
- (2PnP-2Msgs) This scenario is similar to the scenario 2PnP, but uses PnP-2Msgs instead of PnP.

For PnP/ PnP-2Msgs, *badState* holds if vac state is off or on-novac and track state is mvL. For 2PnP/2PnP-2Msgs, *badState* holds if one of the component PnP applications satisfies badState.

For each scenario, Maude search was used to check reachability of bad states in the presence of a bounded intruder with the bound on the number of intrusions between 0 and 3. Note that unreachability in the bound 0 case shows that the application alone is safe with respect to the considered bad state. Table 1 summarizes experiments using the four scenarios described above.

These experiments show that it is feasible in practice to formally verify simple scenarios and even more complicated ones. However, as expected by the complexity results reported in [1], the computational effort increases exponentially as we increase the size of the system. Moreover, increasing the bound on intruders impacts search

Table 1: Attack search for different Pick-n-Place scenarios with bounded intruder. The values in parentheses, $\times n$, for a scenario and bound on intruder, denotes that Maude traversed n times more states than the scenario PnP with the same value for the bound on intruder. The experiments were run on a MacBook Pro, 2.4 Ghz Intel Core i5, 16GB memory. These experiments appeared in our previous work [1].

Scenario	Bound on Intruder	Number of States	Time(ms)	BadState?
PnP	0	23	4	no
	1	84	11	yes
	2	406	47	yes
	3	1651	178	yes
2PnP	0	84 ($\times$3.7)	40	no
	1	388 ($\times$4.6)	182	yes
	2	2873 ($\times$7.1)	1409	yes
	3	26440 ($\times$16.0)	19631	yes
PnP-2Msgs	0	29 ($\times$1.3)	40	no
	1	722 ($\times$8.5)	177	no
	2	1854 ($\times$4.6)	912	yes
	3	10248 ($\times$6.2)	4965	yes
2PnP-2Msgs	0	114 ($\times$4.9)	88	no
	1	6814 ($\times$81.1)	5277	no
	2	22179 ($\times$54.1)	18208	yes
	3	153824 ($\times$93.1)	225898	yes

as expected. Higher bound values means that intruders are capable to carry out more complex attacks. For example, in the scenarios 2PnP and 2PnP-2Msgs the intruder needs at least two actions to carry out an attack.

4.2. Analysis Refinement

An intruder may need to send more than one message in order to carry out an attack that could lead to harm. The approach outlined in Section 2 for constructing the policies of security wrappers would sign *all the messages* that the intruder could use to trigger an attack. As our goal is to reduce the number of signed messages for performance reasons, we investigate in this section how we could refine this analysis so to reduce the number of messages required to be signed.

To build more refined security wrappers, we rely on the following observation: to block an attack, it suffices to block the intruder to send any single message necessary for carrying out the attack.

For example, let $\mathcal{M} = \{\mathsf{msg}_1, \ldots, \mathsf{msg}_n\}$ be the messages necessary for carrying out an attack. Then instead of constructing security wrappers that would sign all messages $\mathsf{msg} \in \mathcal{M}$, we can pick a non-empty subset of messages $\mathcal{M}' \subseteq \mathcal{M}$ and require that the messages in $\mathcal{M}'$ to be signed. In fact, we could pick a singleton set

$\mathcal{M}' = \{\mathsf{msg}_j\}$ for some $1 \leq j \leq n$. If this message msg_j is required to be signed, then the attacker cannot complete the attack.

There are some problems in implementing this solution:

- **Problem 1:** how to compute the necessary messages, $\mathcal{M}$, to carry out an attack? With the approach described and implemented in Section 2, we obtain an upper bound of messages. That is, the attack message sets, $\mathcal{A}$, in Figure 1 may contain messages that are not strictly necessary to carry out an attack, but it does contain all messages that are necessary for carrying out an attack. Formally, $\mathcal{M} \subseteq \mathcal{A}$, but not necessarily $\mathcal{A} = \mathcal{M}$.

  This is because of our intruder model specification. Recall that the intruder is given a fixed number, n, of (symbolic) messages that he can use to carry out an attack. If an attack can be carried out using fewer messages than available to the attacker, then the set of attack messages found by our search machinery may contain spurious messages that are not needed for carrying out the attack.
- **Problem 2:** There may be more than one possible attack. Therefore, our machinery would find multiple sets of attack messages $\mathcal{A}_1, \ldots, \mathcal{A}_m$. How can we minimize the set of message required to be encrypted while still mitigating all possible attacks?

In the following we show how to solve these problems.

4.3. Minimal Protection Sets

Intuitively, for each attack, it is enough to secure at least one of the messages used by the intruder to carry out that attack. We call such sets *protection sets*.

Definition 4.1. *Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be the attack sets on a given system by an intruder sending at most m messages. A protection set is a set of messages such that if all messages in this set are protected then no one of the attacks corresponding to the attack sets $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is possible. The protection set is minimal if when any message is removed it fails to be a protection set,* i.e.*, there is an attack.*

For example, assume that the following attack sets:

$$\{\mathsf{msg}_2\}, \{\mathsf{msg}_1, \mathsf{msg}_2\}, \{\mathsf{msg}_1, \mathsf{msg}_3, \mathsf{msg}_5\}, \text{ and } \{\mathsf{msg}_4\}. \tag{1}$$

The naive approach depicted in Figure 1 would lead to securing all messages in the attack sets: $\{\mathsf{msg}_1, \mathsf{msg}_2, \mathsf{msg}_3, \mathsf{msg}_4, \mathsf{msg}_5\}$. Indeed this set is a protection set. However it is not minimal as it is possible to remove msg_1, *i.e.*, not secure msg_1, and the attacks are still not possible. The protection sets $\{\mathsf{msg}_1, \mathsf{msg}_2, \mathsf{msg}_4\}$, $\{\mathsf{msg}_2, \mathsf{msg}_3, \mathsf{msg}_4\}$ and $\{\mathsf{msg}_2, \mathsf{msg}_4, \mathsf{msg}_5\}$ are minimal.

We describe next an algorithm to compute minimal protection sets from the attack sets computed using the approach depicted in Figure 1.

Notice that the search for attacks by an intruder with n messages enumerates all attack sets $\mathcal{A}$ such that $|\mathcal{A}| \leq n$. Let $all\mathcal{A}[n] = \{\mathcal{A} \mid \mathcal{A} \text{ is an attack set and } |\mathcal{A}| \leq n\}$. If $\mathcal{A} \in all\mathcal{A}[n]$ of size j is not minimal then there must be some $\mathcal{A}' \in all\mathcal{A}[n]$ of size $i < j$ such that $\mathcal{A}' \subset \mathcal{A}$. Thus we can reduce $all\mathcal{A}[n]$ to $min\mathcal{A}[n]$ that contains all and only minimal attack sets. A protection set is one whose intersection with each $\mathcal{A} \in min\mathcal{A}[n]$ is non-empty.

We compute minimal protection set for a given enumeration `asetSets` of all attacks of size $<= n$ as follows.

Step 1: Turn the input into a list

```
asetSetsL = emsgsSet2emsgsList(asetSets)
```

whose j^{th} element (counting from 0) is the attack sets from `asetSets` of size $j+1$.

Step 2: Prune the obtained list, to remove any attack set that contains an attack set of smaller size

```
asetSetsLp = pruneEMsgss(asetSetsL).
```

Step 3: From the pruned list we compute candidate minimal protection sets as follows: We work with structures that are pairs `[emsgs,emsgssl]` where `emsgs` is a partial candidate protection set, and `emsgssl` is the result of removing `emsgs` from each attack set in `asetSetsLp` (and removing empty sets). Starting with the single pair `[none,asetSetsLp]`, a pair `[emsgs,emsgssl]` is processed by computing the sets `[emsgs emsg, emsgssl/emsg]` such that `emsg` is in `mxOcc(emsgssl)` and `emsgssl/emsg` is the result of removing `emsg` from each attack set in `emsgssl` (and removing empty sets). Here `mxOcc(emsgssl)` is the set of `emsgs` that occur in the maximum number of attack sets in `emsgssl`. When `emsgssl/emsg` is empty, this means all attack sets have been covered by `emsgs emsg` and it is added to an accumulated set of candidate minimal protection sets.

Step 4: We verify whether `emsgs emsg` is minimal or further reduce the size by removing elements of a candidate set one by one and checking whether the result intersects every attack set.

To illustrate our algorithm, consider the attack sets in Equation 1. We first order the set of attack sets into a list according to its size. This leads to the list:

$$[\{\mathsf{msg}_2\}, \{\mathsf{msg}_4\}, \{\mathsf{msg}_1, \mathsf{msg}_2\}, \{\mathsf{msg}_1, \mathsf{msg}_3, \mathsf{msg}_5\}]$$

We then start removing from this list any attack set that is a superset of another attack set in the list. For example, the attack set $\{\mathsf{msg}_1, \mathsf{msg}_2\} \supset \{\mathsf{msg}_2\}$ and therefore it is removed. It results in the following list

$$\mathcal{A}_L = [\{\mathsf{msg}_2\}, \{\mathsf{msg}_4\}, \{\mathsf{msg}_1, \mathsf{msg}_3, \mathsf{msg}_5\}]$$

Now we start with the pair $[\emptyset, \mathcal{A}_L]$. Pick a message that appears in the most attack sets in $\mathcal{A}_L$. In the case above, any message will do as all messages appear once. Say we picked msg_4. This results in the pair:

$$[\{\mathsf{msg}_4\}, [\{\mathsf{msg}_2\}, \{\mathsf{msg}_1, \mathsf{msg}_3, \mathsf{msg}_5\}]]$$

The algorithm continues by picking say msg_5 leading to the pair:

$$[\{\mathsf{msg}_4, \mathsf{msg}_5\}, [\{\mathsf{msg}_2\}]]$$

and finally msg_2, returning the minimal set:

$$\{\mathsf{msg}_2, \mathsf{msg}_4, \mathsf{msg}_5\}.$$

The algorithm computes a protection set that contains at least one message of each given attack set. The protection set computed is minimal as we check that removing any element would enable an attack.

Theorem 4.2. *The algorithm described above results in a minimal protection set.*

Proof. The algorithm input I is $all\mathcal{A}[n]$ the set of attack message sets of size at most n. A protection set for I is a set of messages such that if the intruder is unable to send any of these messages, no attack in I can be carried out.

The algorithm has three stages: (1) converting the input to a list

$$\texttt{asetSetsLp} = \texttt{pruneEMsgss(emsgsSet2emsgsList(asetSets);}$$

(2) computing refinement sequences of partial protection sets $\texttt{[emsgs}_i\texttt{,emsgssl}_i\texttt{]}$ by adding a message in $\texttt{maxOcc(emsgssl}_i\texttt{)}$ to $\texttt{emsgs}_i$ and removing it from all message sets of $\texttt{emsgssl}_i$ until there are no more messages to remove; (3) removing redundant messages from the resulting protection sets.

1. Claim: A message set `emsgs` is a protection set for the input I iff it has non-empty intersection with each message set in `asetSetsLp`.

By construction, if `emsgs` is an attack set of I of size $j+1$ then either `emsgs` is an element of $\texttt{asetSetsLp}[j]$ or there is some $i < j$ and `emsgs0` in $\texttt{asetSetsLp}[i]$ that is contained in `emsgs`. Furthermore, if `emsgs` is an element of $\texttt{asetSetsLp}[j]$ then for $i < j$ no `emsgs0` in $\texttt{asetSetsLp}[i]$ is a subset of `emsgs`.

(Forward implication) Suppose `emsgs` is a protection set and there is some `emsgs0` in `asetSetsLp` that has empty intersection with `emsgs`. Since `emsgs0` is an attack set and all its element are available to the attacker, `emsgs` can not be a protection set for I.

(Backward implication) Suppose `emsgs` has non-empty intersection with each message set in `asetSetsLp`. If there is an attack it must use $\texttt{emsgsA} - \texttt{emsgs}$ for some `emsgsA` in I. This is another attack set and is either in `asetSetsLp` or contains some message set from `asetSetsLp` which intersects `emsgs`. A contradicton.

Thus Claim 1 is proved.

2. We claim stage 2 produces a (finite) tree of partial protection sets such that the message set of the leaves are protection sets for the I. In particular for each node,$[\texttt{emsgs}, \texttt{emsgssl}]$, of the tree if `emsgs1` intersects every set of `emsgssl` then $\texttt{emsgs} \cup \texttt{emsgs1}$ intersects every message set of `asetSetsLp` (recall the root). Clearly this holds for the root of the tree, $[\texttt{none}, \texttt{asetSetsLp}]$. Assume the claim holds for a node $[\texttt{emsgs}, \texttt{emsgssl}]$. Its children have the form

$$[\texttt{emsgs,emsg}, \texttt{emsgssl/emsg}]$$

where $\texttt{emsg} \in \texttt{maxOcc(emsgsl)}$ and `emsgssl/emsg` is the result of removing `emsg` from each message set of `emsgssl` (and removing empty sets). Thus if

Table 2: Number of messages that are signed by the Naive Security Wrappers and the Refined Security Wrapper

Case	PnP	2PnP	PnP-2Msgs	2PnP-2Msgs
Naive	3	3	7	14
Refined	1	1	3	6

`emsgs1` intersects each (non-empty) message set of `emsgssl/emsg` then the set `emsgs1,emsg` intersects each (non-empty) message set of `emsgssl` since if a message set of `emsgssl` does not intersect `emsgs1` it must be because it was removed by `emsg` and hence intersects with `emsg`. Clearly the tree is finite, since the branches have finite choices and at each level the second component gets smaller. Thus Claim 2 is proved.

Finally, stage 3 just removes messages that can be eliminated with out violating the intersection property, to produce minimal protection sets.

Note that the algorithm `genMinProts` is sound (by the above proposition) but is not complete. It will generate some minimal protection sets, but there may be some that it misses. If completeness is more important than efficiency, the algorithm can be modified to consider every message that occurs in some set in `emsgssl` rather than restricting attention to messages in `mxOcc(emsgssl)`. This will be, however, less efficient.

We applied our algorithm to the the four scenarios presented in Section 4. The results are summarized by Table 2. PnP and 2PnP have pruned attack sets of size 1 for bounds up to 3. Thus the union of these sets is the minimal protection set. For scenarios PnP-2Msgs and 2PnP-2Msgs, pruned attack sets are of size 2 and there is a single minimal protection set. In the PnP-2Msgs scenario the naive protection set has size 7, and the minimal protection set has size 3. In the 2PnP-2Msgs scenario the naive protection set has size 14, and the minimal protection set has size 6.

5. 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 [12], and the ENISA study on good practices for IoT security [10]. 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 [17] 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.* [15] 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 work in [24] relates to our work in proposing a method using formal methods to find all attacks on a system given possible attacker actions. The authors do not propose mitigations. SOTERIA [6] is a tool for evaluating safety and security of individual or collections of IoT applications. It uses formal methods to verify properties of abstract models of applications derived automatically from code (of suitable form). It requires access to the application source code.

Several mature tools based on formal methods, such as TAMARIN [19], Maude-NPA [11], ProVerif [5] and OFMC [4], have been applied for the verification of security protocols. These works are based on similar symbolic techniques used here, such as modeling intruder symbolically following the Dolev-Yao intruder model [9]. The application here is different as we verify embedded systems and not communication protocols. This impacts the type of analyses that are required. For example, the types of event messages transmitted between devices is far simpler than the messages transmitted in security protocols. Moreover, as Industry 4.0 applications are cyber-physical systems, safety becomes important. Therefore, the main goal is not to preserve the confidentiality of some data, but to guarantee the safety of the system even in the presence of intruders. The formal model proposed here reflects this as it takes as input not the messages that shall be confidential, but system configurations that are hazardous. Finally, it is not clear whether existing tools can be used to recommend policies for security wrappers as done by the machinery proposed in this paper.

The MBSE tool TTool [3] provides automated support for security verification using ProVerif. It is to the best of our knowledge the only MBSE tool integrated with formal security verification tools. Following an MBSE approach, system specification uses function blocks whose behavior are specified using activity diagrams. It implements a model to model translation [2] from TTool specification to ProVerif specifications enabling the verification of security properties, such as confidentiality and authenticity. As with ProVerif, TTool does not support the use of formal verification to identify how intruders can lead to harm neither support automated methods to construct security wrappers.

The complexity of periodic system such as those used in Industry 4.0 has been subject of the paper [1]. It has been shown that if the intruder is not bounded, reachability problems are undecidable. Moreover, the same problems are PSPACE-complete if the intruder is bounded. This paper complements the existing work by demonstrating that existing methods can be used in realistic size application, such as the PnP.

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 [20]. An early example of wrapping to achieve security guarantees is presented in [7] to mitigate DoS attacks.

6. 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 [13], we are investigating the complexity of security properties given intruder models weaker than the traditional Dolev-Yao intruder [9]. We are also considering increasing the expressiveness of function block specifications to include time constraints as in [14] to automate the verification of properties based on time trace equivalence [23], 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 [26].

Another important direction is developing theory transformations for correct-by-construction distributed execution [16]. 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 [25] 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 was partially supported by U. S. Office of Naval Research under award numbers N00014-15-1-2202 and N00014-20-1-2644, and NRL grant N0017317-1-G002. Nigam was partially supported by NRL grant N0017317-1-G002, and CNPq grant 303909/2018-8.

References

[1] Alturki, M.A., Kirigin, T.B., Kanovich, M.I., Nigam, V., Scedrov, A., Talcott, C.L., 2021. On security analysis of periodic systems: Expressiveness and complexity, in: Proceedings of the 7th International Conference on Information Systems Security and Privacy, ICISSP 2021, Online Streaming, February 11-13, 2021, pp. 43–54. URL: https://doi.org/10.5220/0010195100430054, doi:10.5220/0010195100430054.

[2] Ameur-Boulifa, R., Lugou, F., Apvrille, L., 2018. Sysml model transformation for safety and security analysis, in: Security and Safety Interplay of Intelligent Software Systems - ESORICS 2018 International Workshops, ISSA/CSITS@ESORICS 2018, Barcelona, Spain, September 6-7, 2018, Revised Selected Papers, pp. 35–49. URL: https://doi.org/10.1007/978-3-030-16874-2_3, doi:10.1007/978-3-030-16874-2_3.

[3] Apvrille, L., 2008. Ttool for DIPLODOCUS: an environment for design space exploration, in: Proceedings of the 8th international conference on New technologies in distributed systems, NOTERE '08, Lyon, France, June 23-27, 2008, pp. 28:1–28:4. URL: https://doi.org/10.1145/1416729.1416764, doi:10.1145/1416729.1416764.

[4] Basin, D.A., Mödersheim, S., Viganò, L., 2005. OFMC: A symbolic model checker for security protocols. Int. J. Inf. Sec. 4, 181–208. URL: https://doi.org/10.1007/s10207-004-0055-7, doi:10.1007/s10207-004-0055-7.

[5] Blanchet, B., 2013. Automatic verification of security protocols in the symbolic model: The verifier proverif, in: Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures, pp. 54–87. URL: https://doi.org/10.1007/978-3-319-10082-1_3, doi:10.1007/978-3-319-10082-1_3.

[6] Celik, Z.B., McDaniel, P.D., Tan, G., 2018. Soteria: Automated iot safety and security analysis, in: 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018, pp. 147–158. URL: https://www.usenix.org/conference/atc18/presentation/celik.

[7] Chadha, R., Gunter, C.A., Meseguer, J., Shankesi, R., Viswanathan, M., 2008. Modular preservation of safety properties by cookie-based dos-protection wrappers, in: Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings, pp. 39–58. URL: https://doi.org/10.1007/978-3-540-68863-1_4, doi:10.1007/978-3-540-68863-1_4.

[8] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., 2007. All About Maude: A High-Performance Logical Framework. volume 4350 of *LNCS*. Springer. doi:10.1007/978-3-540-71999-1_1.

[9] Dolev, D., Yao, A., 1983. On the security of public key protocols. IEEE Transactions on Information Theory 29, 198–208. doi:10.1109/TIT.1983.1056650.

[10] ENISA, 2018. Good practices for security of internet of things in the context of smart manufacturing. Available at https://www.enisa.europa.eu/publications/good-practices-for-security-of-iot.

[11] Escobar, S., Meadows, C.A., Meseguer, J., 2007. Maude-npa: Cryptographic protocol analysis modulo equational properties, in: Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures, pp. 1–50. URL: https://doi.org/10.1007/978-3-642-03829-7_1, doi:10.1007/978-3-642-03829-7_1.

[12] Fiat, M., et al., 2017. OPC UA security analysis. Available at https://opcfoundation.org/wp-content/uploads/2017/04/OPC_UA_security_analysis-OPC-F-Responses-2017_04_21.pdf.

[13] Kanovich, M.I., Kirigin, T.B., Nigam, V., Scedrov, A., 2014. Bounded memory Dolev-Yao adversaries in collaborative systems. Inf. Comput. 238, 233–261. URL: http://dx.doi.org/10.1016/j.ic.2014.07.011, doi:10.1016/j.ic.2014.07.011.

[14] Kanovich, M.I., Kirigin, T.B., Nigam, V., Scedrov, A., Talcott, C.L., 2017. Time, computational complexity, and probability in the analysis of distance-bounding protocols. Journal of Computer Security 25, 585–630. URL: https://doi.org/10.3233/JCS-0560, doi:10.3233/JCS-0560.

[15] Lanotte, R., Merro, M., Munteanu, A., Viganò, L., 2020. A formal approach to physics-based attacks in cyber-physical systems. ACM Trans. Priv. Secur. 23, 3:1–3:41. URL: https://doi.org/10.1145/3373270, doi:10.1145/3373270.

[16] Liu, S., Sandur, A., Meseguer, J., Ölveczky, P.C., Wang, Q., 2020. Generating correct-by-construction distributed implementations from formal Maude designs, in: NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings, pp. 22–40. URL: https://doi.org/10.1007/978-3-030-55754-6_2, doi:10.1007/978-3-030-55754-6_2.

[17] Lun, Y.Z., D'Innocenzo, A., Malavolta, I., Benedetto, M.D.D., 2016. Cyber-physical systems security: a systematic mapping study. CoRR abs/1605.09641. Available at http://arxiv.org/abs/1605.09641.

[18] Mealy, G.H., 1955. A method for synthesizing sequential circuits. Bell System Technical Journal 34, 1045–1079. URL: https://doi.org/10.1002/j.1538-7305.1955.tb03788.x, doi:10.1002/j.1538-7305.1955.tb03788.x.

[19] Meier, S., Schmidt, B., Cremers, C., Basin, D.A., 2013. The TAMARIN prover for the symbolic analysis of security protocols, in: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, pp. 696–701. URL: https://doi.org/10.1007/978-3-642-39799-8_48, doi:10.1007/978-3-642-39799-8_48.

[20] Meseguer, J., 2014. Taming distributed system complexity through formal patterns. Sci. Comput. Program. 83, 3–34. URL: https://doi.org/10.1016/j.scico.2013.07.004, doi:10.1016/j.scico.2013.07.004.

[21] Nigam, V., Talcott, C.L., 2019. Formal security verification of industry 4.0 applications, in: 24th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2019, Zaragoza, Spain, September 10-13, 2019, pp. 1043–1050. URL: https://doi.org/10.1109/ETFA.2019.8869428, doi:10.1109/ETFA.2019.8869428.

[22] Nigam, V., Talcott, C.L., 2020. Automated construction of security integrity wrappers for industry 4.0 applications, in: Rewriting Logic and Its Applications - 13th International Workshop, WRLA 2020, Virtual Event, October 20-22, 2020, Revised Selected Papers, pp. 197–215. URL: https://doi.org/10.1007/978-3-030-63595-4_11, doi:10.1007/978-3-030-63595-4_11.

[23] Nigam, V., Talcott, C.L., Urquiza, A.A., 2019. Symbolic timed trace equivalence, in: Foundations of Security, Protocols, and Equational Reasoning - Essays Dedicated to Catherine A. Meadows, pp. 89–111. URL: https://doi.org/10.1007/978-3-030-19052-1_8, doi:10.1007/978-3-030-19052-1_8.

[24] Tabrizi, F.M., Pattabiraman, K., 2016. Formal security analysis of smart embedded systems, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016, pp. 1–15. URL: http://dl.acm.org/citation.cfm?id=2991085.

[25] Terzimehic, T., Voss, S., Wenger, M., 2018. Using design space exploration to calculate deployment configurations of IEC 61499-based systems, in: 14th IEEE International Conference on Automation Science and Engineering, CASE 2018, Munich, Germany, August 20-24, 2018, pp. 881–886. URL: https://doi.org/10.1109/COASE.2018.8560591, doi:10.1109/COASE.2018.8560591.

[26] Urquiza, A.A., AlTurki, M.A., Kanovich, M.I., Kirigin, T.B., Nigam, V., Scedrov, A., Talcott, C.L., 2019. Resource-bounded intruders in denial of service attacks, in: CSF, pp. 382–396. doi:10.1109/CSF.2019.00033.

[27] Yoong, L.H., Roop, P.S., Bhatti, Z.E., Kuo, M.M.Y., 2015. Model-Driven Design Using IEC 61499 - A Synchronous Approach for Embedded and Automation

Systems. Springer. URL: https://doi.org/10.1007/978-3-319-10521-5, doi:10.1007/978-3-319-10521-5.

[28] Zoitl, A., Lewis, R., 2014. Modelling control systems using IEC 61499. Control Engineering Series 95, The Institution of Electrical Engineers, London. doi:10.1049/PBCE095E. 2nd Edition.