Computing and Informatics, Vol. 38, 2019, 1001–1032, V 2020-Jan-16

FORMAL VERIFICATION OF SECURITY PATTERN COMPOSITION: APPLICATION TO SCADA

Fadi Obeid, Philippe Dhaussy

Lab-STICC CNRS, UMR 6285
Ensta Bretagne, 2 Rue François Verny
29200 Brest, France
e-mail: {fadi.obeid, philippe.dhaussy}@ensta-bretagne.fr

Abstract. Information security was initially required in specific applications, however, nowadays, most companies and even individuals are interested in securing their information assets. The new requirement can be costly, especially with the high demand on security solutions and security experts. Security patterns are reusable security solutions that prove to be efficient and can help developers achieve some security goals without the need for expertise in the security domain. Some security pattern combinations can be beneficial while others are inconsistent. Model checking can be used to verify the production of combining multiple security patterns with an architecture. Supervisory control and data acquisition (SCADA) systems control many of our critical industrial infrastructures. Due to their limitations, and their augmented connectivity, SCADA systems have many unresolved security issues. In this paper, we demonstrate how we can automatically generate a secure SCADA model based on an insecure one and how to verify the generated model.

Keywords: Information security, security patterns, formal verification, model checking, SCADA

1 INTRODUCTION

Information security is a tremendous challenge, whereas meeting security requirements without affecting the services is a difficult task. General security guidelines [20] are useful in avoiding many security problems and building a fairly secure system. Furthermore, some security problems are highly recurrent and efforts to resolve them are redundant. A best solution may be found in a specific context which can later be beneficial for others. A security pattern describes how to apply a specific security measure to solve a security problem.

To achieve multiple security goals, multiple security patterns can be used. However, while some security patterns are better when combined, others are inconsistent and can raise new problems. Combining multiple security patterns in an application can have additional issues depending on the application itself.

An efficient security solution should solve a security issue to fulfill some security requirements without affecting other system requirements. Model checking can be used to verify that the system meets its requirements and ensures correct functionality [6]. It can also verify the consistency of the security patterns with the architecture and with each other. This is important to validate the security properties of the system as well as the security properties of the patterns themselves.

Supervisory control and data acquisition (SCADA) systems control many of our critical industrial infrastructures, such as nuclear and chemical plants. Due to their limitations, and their augmented connectivity, SCADA systems raise tremendous challenges for security experts. With their requirements and the criticality of their security, it is very difficult to achieve a good security level in SCADA systems. Finally, attacks on SCADA have risen lately [5].

The main objective of our work is to automatically generate a secure architecture based on an insecure architecture and some security requirements. We also need to verify the result using model checking to make sure the security requirements of the architecture are respected. This requires three different elements:

1. a library of security patterns,
2. a tool to automatically combine the patterns with an architecture,
3. a model-checker to verify the resulting model.

The effiency of the patterns and the choice of the security policies are out of the scope of this paper.

In this paper we demonstrate how we can automatically generate a secure SCADA model based on an insecure one. The generated model applies multiple security patterns based on a security policy. The model can later be verified using model checking to validate the security properties and ensure correct functionality. Finally, we present some measurements regarding the complexity of our approach.

We start by introducing a background study of related work in Section 2. We specify the abstract model in Section 3, any implementation of our approach should respect this model. In Section 4, we illustrate the included security patterns and their security properties. We define our approach for security patterns combination in Section 5. The verification approach is explained in Section 6. In Section 7, we describe our implementation approach along with an example and some complexity studies. Finally, we conclude in Section 8.

2 RELATED WORK

Yoshioka et al. [26] offer a survey on security patterns. Many security patterns were introduced by Yoder and Barcalow [25] and Fernandez and Pan [14]. Schumacher et al. [21] provide a full description of integrating security patterns into systems. Hafiz et al. [17] present a pattern language unifying and classifying all published security patterns at the time. Wassermann and Cheng [24] detail many security patterns and enable the verification of these patterns by adding formal constraints to them.

Avalle et al. [3] wrote a survey on formal verification of security protocol implementations, focusing on the automatic verification of models close to the real implementation. There are many research efforts on model checking of design patterns [2, 10, 1, 22] which we consider enriching. Dong et al. [11] investigate model checking of security pattern combinations where the authors explain how wrongly combining security patterns may result in several errors.

Concerning SCADA security, many studies [19, 18, 27] were conducted to address the security problems in SCADA and/or give some theoretical solutions and guides for improving SCADA security. Other studies [23, 15] aim to enhance the security level and fortify the provided services along with the managed critical data. SCADA-specific security solutions and SCADA-specific IDS are proposed by Fovino et al. [16] and Zhu and Sastry [28], respectively. Fernandez et al. [12] propose the use of security patterns to design secure SCADA systems.

To the best of our knowledge, there is no research work that considers formally verifying properties on an auto generated architecture with a combination of security patterns. In our work, we are interested in generating secure architecture (using security patterns) and providing automatic proofs of the robustness of the generated architecture based on specific security requirements.

3 ABSTRACT MODEL

Figure 1 describes the abstract model on which security patterns are applied. We consider an architecture containing multiple entities (*Entity*) communicating between each other through communication channels (*Channel*). The communication channel has a *Fifo* to organize the order of messages (*Message*). The entities are divided into two categories: *Components* which are the assets of the architecture and need to be secured, and *Clients* that are seen as guests to the architecture. Any entity can receive and send messages. The client (*Client*) can additionally initiate requests and is considered as an external entity of the system. The communication component (*ComComp*) can only forward messages from one entity to the other. The access component (*AccComp*), in addition to forwarding messages, owns resources (*Resource*) and responds to requests by accessing these resources.

During experimentations, multiple formats for the message were used. We found that the most convenient method is to separate message parts depending on the security hypothesis of these parts.

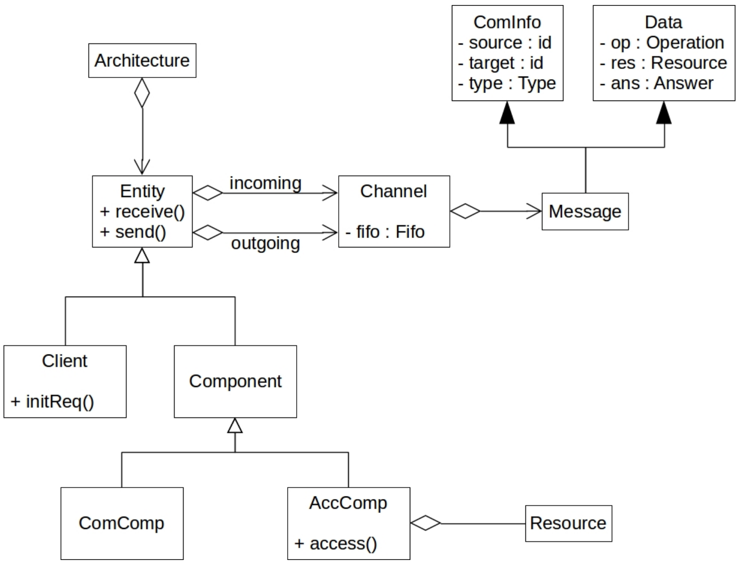


Figure 1. Abstract model

We consider the *comInfo* to have integrity, meaning if it is modified by an entity x, the source of the message becomes x. However, this part is not confidential and any other entity can read its contents so it can forward the message if needed. In addition for integrity, the *data* part is considered to have confidentiality, meaning that only the target of the message can read this part.

The message is therefore formatted as follows:

- *comInfo* containing information about the communication:
  - *source*: the source of the message,
  - *target*: the target of the message,
  - *type*: the type of the message, either a request *REQ* or a response *RESP*.
- *data* containing the details of the request or the response:
  - *op*: the operation concerned by the message (*READ* or *WRITE*),
  - *res*: the resource concerned by the message,
  - *ans*: the answer, *ACK* or *NAK* for *RESP* and *null* for *REQ*.

Figure 2 demonstrates the automata of a communication component, an access component, and a client, respectively. A communication component receives a message and goes to *Received* state, depending on whether it can forward the message or not, it either goes back to *Idle*, or it goes to *Sending*. An access component behaves differently after the *Received* state, if the target of the message is the component itself, it goes to *Respond*, else, it goes to *Sending* to forward the message. From

Respond, the access is accomplished by passing through *Access* and *Respond* again, finally, when the response is ready, the *Sending* state is visited to send the response. In the case of the client, it can initiate a request, and send this request, then it waits for a response at *waitResp*. When the response arrives, the client goes back to *Idle* state so it can send other requests.

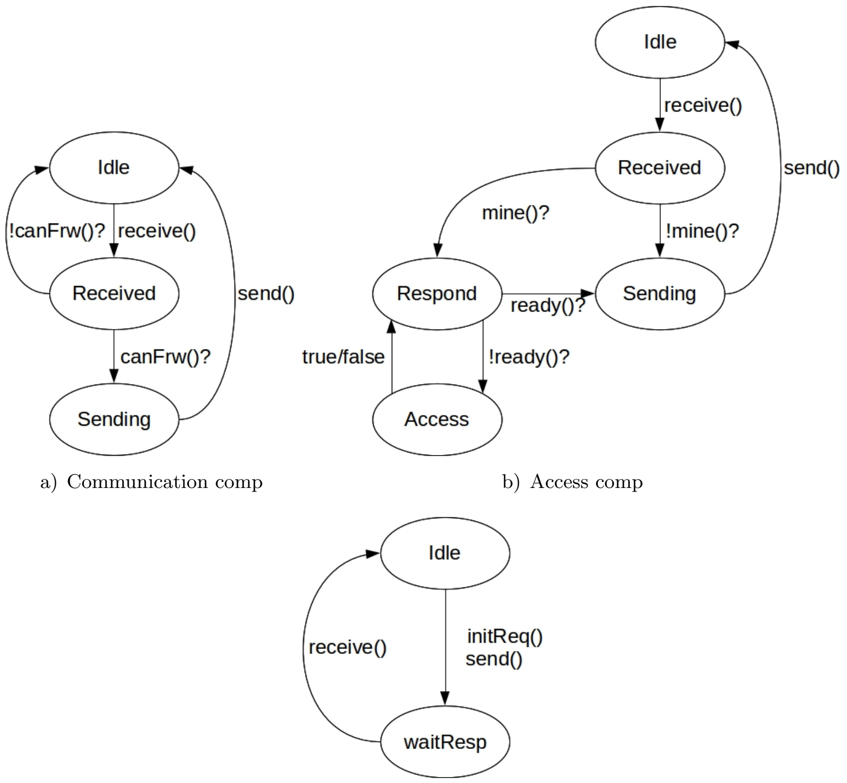


Figure 2. Automata of the interacting entities

In our approach, we assume that any component is physically robust, meaning that its behavior cannot be affected directly from the outside. The information passing from one part of the component to another is secure, it is not visible to other entities nor can they change it. The only way to affect a component is by sending a message to this component resulting in the component's behavior.

Our concept is to secure this behavior in a way that no matter what happens outside the components, the system behaves correctly, while no security properties are violated. Any integrated security pattern is also considered internal to the component. This means that any relation/communication/sharing between these patterns, and between these patterns and the component, is not visible to nor modifiable by any other entities.

Any implementation should respect this model, meaning that we can create more complicated entities, but the minimal structure should be respected. In addition, some behavior properties are used to limit possible problems and simplify the application of the patterns.

We start by explaining the required notations:

- $\mathcal{A}comps$, $\mathcal{C}comps$, and $\mathcal{C}lts$ are the sets of access components, communication components, and clients, respectively,
- $\mathcal{C}omps = \mathcal{A}comps \cup \mathcal{C}comps$ and $\mathcal{E}nts = \mathcal{C}omps \cup \mathcal{C}lts$,
- $received(e,\ m)$ is detected when entity e receives message m,
- $sent(e,\ m)$ is detected when e sends m,
- *any* is a joker used when the variable does not matter, for example, we write $received(e, any)$ detects the event where e received any message,
- $e.state$ is the current state of the entity e.

We formalize the properties shared by the different components as follows:

- Receive messages only when at *Idle* state.

$$\begin{aligned}&\textbf{prt\_ARCH\_1} : \forall c \in \mathcal{C}omps,\\ &\Box[received(c, any) \Rightarrow c.state = Idle].\end{aligned} \tag{1}$$

- Send messages only from *Sending* state.

$$\begin{aligned}&\textbf{prt\_ARCH\_2} : \forall c \in \mathcal{C}omps,\\ &\Box[sent(c, any) \Rightarrow c.state = Sending].\end{aligned} \tag{2}$$

- Never leave *Sending* state without sending a message.

$$\begin{aligned}&\textbf{prt\_ARCH\_3} : \forall c \in \mathcal{C}omps,\\ &\Box[c.state = Sending \Rightarrow \Diamond sent(c, any)].\end{aligned} \tag{3}$$

- To reduce the number of unnecessary configurations, some variables are given a null value (exp. $mess = MESS\_NULL$) when the component is at *Idle* state.

$$\begin{aligned}&\textbf{prt\_ARCH\_4} : \forall c \in \mathcal{C}omps,\\ &\Box[c.state = Idle \Rightarrow c.mess = MESS\_NULL].\end{aligned} \tag{4}$$

The formal description of the properties is close to implementation, since the included parameters and states are included in the minimal requirements for any application to our approach. However, when describing the security patterns in the following section, some formal properties are more abstract so they would consider different implementations.

4 SECURITY PATTERNS

The security pattern is a reusable solution to a recurring security problem. It is usually constructed by security specialists and used by developers who lack security knowledge. It provides a detailed guideline of how to implement the best found solution for a specific security problem.

Patterns should be considered as methodological tools to describe technical solutions related to security. They impose decisions that must be taken into consideration when designing architectures. They also facilitate communication between experts and non-experts.

We consider that the set of security measures to implement for an architecture are not, in general, managed at a single point. Due to the specificities of the patterns, the responsibility for managing the entire implemented security policy is shared between several patterns implemented on multiple components.

We can summarize the essential objective of each pattern as follows:

- *SAP* unifies the access points, instead of having to take security measures in different places, they are taken at the unified access point.
- *CHP* is dedicated to performing verifications and applying countermeasures based on a security policy.
- *AUTH* implements a security policy based on access rights.
- *FWLL* implements a security policy to restrict incoming and outgoing messages based on specified filters.

We divide these patterns into two categories: active patterns and passive patterns. *SAP* and *CHP* are the active patterns that should make decisions, calls, and verifications. *AUTH* and *FWLL* are the passive patterns which represent the forms that the security policies should take. Based on *AUTH* and *FWLL*, *CHP* can verify the conformity of a request with the implemented security policy.

We consider two main objectives related to these patterns:

- Objective 1: Securing access to a component's resources:
  - *SAP* limits the access points to a component's resources to a single access point, and it checks the availability of requested resources.
  - *CHP*, solicited by *SAP*, verifies access rights and takes countermeasures.
  - *AUTH* specifies the access rights.
- Objective 2: Securing access to a set of components:
  - *SAP* limits access points to an area to a single-point component, and it verifies the availability of the recipient entities of the messages.
  - *CHP* is called by *SAP* to verify that the message entering or exiting the area can pass (respects the security policy), if not, a countermeasure is taken.
  - *FWLL* specifies the security policy used to filter messages.

4.1 Single Access Point (*SAP*)

The *SAP* pattern was introduced by Yoder and Barcalow [25]. It can be integrated into different types of implantation: a system, an application, a server, etc.

The goal of the *SAP* pattern is to unify the points on which an area can be accessed in order to improve the control and monitoring of entries.

Passing through a single access point is considered as a control, we therefore use the notation *controlled* on anything that passes through a single access point.

We use *SAP* for two types of control:

- Separating the architecture into areas with one access point between each two areas, and one access point to the external zone. In this case, *SAP* also verifies the availability of a target before forwarding a message. This control can be applied on communicating components with no resources to access. We call this *SAP_C* where *C* stands for communication.
- Separating the access to resources inside a component from the rest of the jobs handled in this component. In this case, at each access request, *SAP* also verifies if the targeted resource is available. This control can be applied on access components with resources to unify their access verifications. We call this *SAP_A* where *A* stands for access.

Since *SAP* is called each time, additional security patterns and security measures can later be patched to the *SAP* so they can handle other security requirements. After verifying the availability of the resource (or the component), if the verification fails, a negative response is prepared and sent. If the verification does not fail, either another security pattern(s) is/are called, or the request is fulfilled by accessing the resource (or forwarding the message).

Based on the *SAP* property classification proposed by Wassermann and Cheng [24], we have formalized some of these properties for different *SAP* applications. Notice that in this article we include some but not all of the properties.

We start by defining some required notations:

- $\mathcal{M}ess$ is the list of all possible messages.
- $\mathcal{R}es$ is the list of all resources.
- $\mathcal{O}pRes$ is the list of all possible operations on resources.
- $\mathcal{SC}comps$ is the list of all communication components using *SAP_C*.
- $\mathcal{SA}comps$ is the list of all components using *SAP_A*.
- $\forall c_c \in \mathcal{SC}comps$, $c_c.comps$ is the list of all components considered inside the secure zone using c_c as a single access point.
- $\forall c_c \in \mathcal{SC}comps$, $\forall m \in \mathcal{M}ess$, $controlled(c_c, m)$ is true if the message m has already been controlled by c_c (passed through the single access point).
- $\forall c_a \in \mathcal{SA}comps$, $c_a.res$ is the list of all resources owned by c_a.

- $\forall c_a \in \mathcal{S}Acomps$, $\forall e \in \mathcal{E}nts$, $\forall o \in \mathcal{O}pRes$, $accessed(c, e, o.oper, o.res)$ is true if e has accessed the resource $o.res$ in c_a with the operation $o.op$.
- $\forall c_a \in \mathcal{S}Acomps$, $\forall e \in \mathcal{E}nts$, $\forall o \in \mathcal{O}pRes$, $controlled(c, e, o)$ is true if c_a has already controlled the request o demanded by e.
- $m' = neg(c, Mess, m)$ is the error message corresponding to m, where c is the source of this error message where $m'.comInfo.target = m.comInfo.source$, $m'.comInfo.source = c$, and $m'.data.ans = NAK$.

We classify the properties depending on the *SAP* application, and the general type of the security property.

- Single Access Point of communication components (*SAP_C*):
  - Authenticity: All messages coming from outside an area protected by a component $c_c \in \mathcal{S}Ccomps$ should pass through this c_c before they go inside the area. Therefore, if a message is received by a component protected by c_c, and the source of the message is not protected by c_c, the message should have been controlled by c_c.

$$\begin{aligned}&\textbf{prt\_SAP\_C\_1.a}:\\ &\forall m \in \mathcal{M}ess, \quad \forall c_c \in \mathcal{S}Ccomps, \quad \forall c \in c_c.comps,\\ &\Box[received(c, m) \wedge m.source \notin c_c.comps\\ &\ \Rightarrow controlled(c_c, m)].\end{aligned} \tag{5}$$

  - Any message received inside an area protected by component $c_c \in \mathcal{S}Ccomps$, without being controlled in advance by c_c, originated from inside the protected area.

$$\begin{aligned}&\textbf{prt\_SAP\_C\_1.b}:\\ &\forall m \in \mathcal{M}ess, \quad \forall c_c \in \mathcal{S}Ccomps, \quad \forall c \in c_c.comps,\\ &\Box[received(c, m) \wedge \neg controlled(c_c, m)\\ &\ \Rightarrow m.source \in c_c.comps].\end{aligned} \tag{6}$$

  - Availability: When a message is received by a component $c_c \in \mathcal{S}Ccomps$, if it is coming from the outside to the inside of the protected zone (the source is not in the protected list), and the target is not available, a negative response is sent to the source of the message. We consider that a target is unavailable if it is not in the list of protected components.

$$\begin{aligned}&\textbf{prt\_SAP\_C\_2}:\\ &\forall m \in \mathcal{M}ess, \quad \forall c_c \in \mathcal{S}Ccomps,\\ &\Box[received(c_c, m) \wedge \{m.source, m.target\} \nsubseteq c_c.comps\\ &\ \Rightarrow \Diamond sent(c_c, neg(c_c, m))].\end{aligned} \tag{7}$$

- Confidentiality: Messages exchanged between components inside a secure zone should never be read by outside entities.

$$\begin{aligned}&\textbf{prt\_SAP\_C\_3}:\\ &\forall m \in \mathcal{M}ess, \quad \forall c_c \in \mathcal{SC}comps, \quad \forall e \in \mathcal{E}nts,\\ &\Box[\mathit{received}(e,m) \wedge m.source \in c_c.comps\\ &\ \wedge m.target \in c_c.comps \Rightarrow e \in c_c.comps].\end{aligned} \tag{8}$$

- Integrity: Messages exchanged between components inside a secure zone should never be modified by outside entities.

$$\begin{aligned}&\textbf{prt\_SAP\_C\_4}:\\ &\forall m \in \mathcal{M}ess, \quad \forall c_c \in \mathcal{SC}comps, \quad \forall e \in \mathcal{E}nts,\\ &\Box[\mathit{sent}(e,m) \wedge m.source \in c_c.comps \wedge m.target \in c_c.comps\\ &\ \Rightarrow e \in c_c.comps].\end{aligned} \tag{9}$$

- Single Access Point of access components (*SAP_A*):
  - Authenticity: Each access to a resource owned by component $c_a \in \mathcal{SA}comps$ is controlled (passes through the single access point).

$$\begin{aligned}&\textbf{prt\_SAP\_A\_1}:\\ &\forall c_a \in \mathcal{SA}comps, \quad \forall e \in \mathcal{E}nt, \quad \forall o \in \mathcal{O}pRes,\\ &\Box[\mathit{accessed}(c_a, e, o.oper, o.res)\\ &\ \Rightarrow \mathit{controlled}(c_a, e, o)].\end{aligned} \tag{10}$$

  - Availability: When a component $c_a \in \mathcal{SA}comps$ controls an access, if the requested resource is not available, a negative response is sent (no matter what the operation is). The negative response is about the whole message and not only the access request, therefore, the received message is used to generate the negative response.

$$\begin{aligned}&\textbf{prt\_SAP\_A\_2}:\\ &\forall c_a \in \mathcal{SA}comps, \quad \forall m \in \mathcal{M}ess, \quad \forall r \in \mathcal{R}es,\\ &\Box[\mathit{received}(c_a, m) \wedge \mathit{controlled}(c_a, m.comInfo.source,\\ &(\mathit{any}, r)) \wedge r \notin c_a.res\\ &\ \Rightarrow \Diamond \mathit{sent}(c_a, \mathit{neg}(c_a, m))].\end{aligned} \tag{11}$$

We notice that these properties do not mention what is authorized and what is not. As mentioned before, *SAP* only organizes and controls messages and access, without applying a security policy. In the case of other security patterns applied on the single access point, the properties of these patterns should consider the already applied pattern.

4.2 Check Point (*CHP*) Pattern

The main objective of *CHP* [25] is to enforce a security policy and activate appropriate countermeasures in the case of violations. While *CHP* can be found in some cases without *SAP*, in our work, we always apply these patterns together. Therefore, we can consider *SAP* in the formalization of *CHP* properties.

CHP does not contain the security policy, but enforces this policy. Therefore, in its formalization, we do not specify what the policy is, but only how to act depending on different situations regarding the policy.

For instance, if a certain policy is violated, the component can be temporarily shut, or can have a specific reaction varying according to what was violated and how. For simplicity, in this document, a countermeasure would be either to neglect a message or a request, or to send negative responses.

Whenever a single access point is visited, if *CHP* is included in the component, it should be visited before the continuation of the job. However, if there is an early refusal due to availability issues (unavailable target or resource), there is no need to call *CHP*.

In the same fashion as for *SAP*, we formalize some of the *CHP* properties extracted from the work of Wassermann and Cheng [24]. The following additional notations are required:

- $\mathcal{S}comps = \mathcal{SC}comps \cup \mathcal{SA}comps$ is the list of all secure components,
- $\forall c_s \in \mathcal{S}comps$, $c_s.policy$ is the security policy applied in c_s,
- $\mathcal{J}obs$ is the list of all possible jobs, which includes any resource access, and the forwarding of any message,
- $counter(p)$ is the countermeasure for violating the security policy p,
- $checked(c_s, job)$ is true if c_s has already checked if job is conform to the security policy applied in c_s,
- $accomplished(c_s, job)$ is true if c_s has accomplished job (the access is accomplished or the message is forwarded),
- $respects(p, job)$ is true if job respects the security policy p,
- $triggered(c_s, a)$ is true if c_s has triggered the action a.

The properties are:

- Availability: Each time *CHP* verifies a job that does not violate the policy, the job should continue (forwarding a message, or accessing a resource).

$$\begin{aligned}&\textbf{prt\_CHP\_1}:\\ &\forall c_s \in \mathcal{S}comps, \quad \forall job \in \mathcal{J}obs,\\ &\Box[checked(c_s, job) \wedge conform(c_s.policy, job)\\ &\ \Rightarrow \Diamond accomplished(c_s, job)].\end{aligned} \tag{12}$$

- Availability: Each time *CHP* verifies a job which does violate the security policy, the appropriate countermeasure should be applied.

$$\begin{aligned}&\textbf{prt\_CHP\_2}: \\ &\forall c_s \in \mathcal{S}comps, \quad \forall job \in \mathcal{J}obs, \\ &\Box[checked(c_s, job) \wedge \neg conform(c_s.policy, job) \\ &\ \Rightarrow \Diamond triggered(c_s, counter(c_s.policy))].\end{aligned} \tag{13}$$

- Authenticity: If a job is accomplished, then it must be verified, this includes the secured jobs only, such as forwarding messages or accessing resources.

$$\begin{aligned}&\textbf{prt\_CHP\_3}: \\ &\forall c_s \in \mathcal{S}comps, \quad \forall job \in \mathcal{J}obs, \\ &\Box[accomplished(c_s, job) \\ &\ \Rightarrow checked(c_s, job) \wedge conform(c_s.policy, job)].\end{aligned} \tag{14}$$

4.3 Authorization Pattern ($AUTH$)

The authorization pattern ($AUTH$) was initially described by Fernandez et al. [14]. The main objective of $AUTH$ is to describe a security policy regarding secure access to objects. The concept is to specify protections on objects, then, create explicit access rights relations between some subjects and some objects so that the subjects have the privilege of accessing these objects.

In our work, the objects are the resources owned by the different components of type *AccComp*, and the subjects are any type of entity. Protections can be specified for an operation without the other (exp. resource r is protected for writing and unprotected for reading). Permissions can also be granted to an entity regarding specific operations or resources or both.

As mentioned before, *CHP* enforces the application of a security policy. In this case, the security policy ensured by *CHP* is an authorization pattern to ensure secure access to protected resources.

In order to apply the $AUTH$ policy on an access component, we use the syntax:

- $protect(c_a, op, r)$: Protect the resource r owned by component c_a for the operation *op*.
- $permit(e, c_a, op, r)$: Explicitly allow entity e to access the resource r owned by the component c_a using the operation *op*.

These functions can be used to specify the security policy which will later be combined with an insecure architecture generating a secure one. We can also use the wild-card *any* to generalize some of the rules. For example, $protect(c, any, r)$ means that resource r owned by component c should be protected for any type of access (read, write, execute, etc.).

Once protections and permissions are specified, we can use following predicates:

- $\mathit{protected}(c_a, op, r)$: which is true if the resource r owned by the access component c_a is protected for the operation op,
- $\mathit{permitted}(c_a, e, op, r)$: which is true if the entity e has explicit permission to access the resource r owned by the access component c_a using the operation op,
- $\mathit{allowed}(c_a, e, op, r)$: which is true if e should be allowed to access r owned by c_a using op. This is true if either the r is not protected for the operation op, or if there is a specific permission for e to have such access.

Notice that

$$\mathit{allowed}(c_a, e, op, r) = (\neg \mathit{protected}(c_a, op, r)) \vee (\mathit{permitted}(c_a, e, op, r)).$$

We are only interested in the essential property of $AUTH$ which specifies that each proceeded access is in fact allowed.

$$\begin{aligned}&\textbf{prt\_AUTH\_1}:\\ &\forall c_a \in \mathcal{S}Acomps, \quad \forall e \in \mathcal{E}nts, \quad \forall o \in \mathcal{O}pRes,\\ &\Box[\mathit{accessed}(c_a, e, o.oper, o.res)\\ &\ \Rightarrow \mathit{allowed}(c_a, e, o.oper, o.res)].\end{aligned} \tag{15}$$

4.4 Firewall Pattern (FWLL)

There are many descriptions for the Firewall pattern [21, 13, 7]. The main objective of $FWLL$ is to filter messages between different communication areas. It can function on the physical, transport, or application layer. At the physical layer, the firewall filter is applied to data packets transmitted over a network. Depending on information in these packets, some of them, or all of them, should be refused. At the transport layer, the filter collects packets until it has enough information to decide whether these packets are transferable or should be refused. Finally, at the application layer, $FWLL$ evaluates all messages against the associated rules concerning services. For example, a given service may not send messages to a certain entity.

In our work we consider a specific $FWLL$ approach which is most comparable to the application layer. In the same fashion as $AUTH$, the concept is to apply a specific security policy using $FWLL$. The policy decides on specific rules to refuse some messages depending on one or more variables in these messages. In addition, the policy can specify some exceptions for any of these rules.

As stated before, the *data* part of the message is confidential, and only the target can read it. This means that a firewall cannot base its filter on this part of the message. However, it can filter messages depending on their source, their target, and the type of the message.

In order to apply the $FWLL$ policy on a communication component, we use the following syntax:

- *protect*$(c_c, comps)$: Applies the *FWLL* pattern to the communication component c_c. All components specified in the list *comps* are considered protected by this firewall, meaning that they are seen by the firewall as internal components. For instance, when *SAP_C* checks for target availability, this is the list it will check. In addition to having a list of inside components $c_c.comps$, c_c would also have $c_c.rules$ which are the rules for filtering messages based on their communication information *comInfo*.
- *addRule*$(c_c, comInfo, exps)$: Adds a new filter rule to $c_c.rules$, the rule is based on *comInfo* stating the source, the target, and the type of the message. *exps* is a list of *comInfo* stating the exceptions to this rule.

In addition to using the wild-card *any* to generalize some of the rules and *null* to specify that there are no exceptions. For example, $addRule(c_c, (e_1, c_1, any), null)$ adds a rule to the policy of c_c stating that any message (of any type) with e_1 as its source and c_1 as its target is refused, with no exceptions.

We can check whether a message satisfies a rule or not using $satisfies(m, rule)$ which is true if $(m.comInfo \neq rule.comInfo) \vee rule.exps.contains(m.comInfo)$. Notice that the equality, as well as the *contains*, both understand the use of *any*. For example, the following statements are both true:

$$(e_1, c_1, req) = (e_1, c_1, any),$$

$$[(e_1, any, any)].contains((e_1, c_1, req)).$$

The most important property (regarding our work) of *FWLL* specifies that each forwarded message does not break any rules.

$$\begin{aligned}&\textbf{prt\_FWLL\_1}: \\ &\forall c_c \in \mathcal{SC}comps, \quad \forall, \forall m \in \mathcal{M}ess, \\ &\Box[received(c_c, m) \wedge sent(c_c, m) \\ &\Rightarrow \forall rule \in c_c.rules, satisfies(m, rule)].\end{aligned} \tag{16}$$

5 SECURITY PATTERN COMPOSITION

Figure 3 demonstrates an abstraction of our combination approach. To generate a secure model, we specify the architecture elements along with the security policies. The security policies are translated into security requirements used to create the properties and assumptions of the system as well as the verification scenarios. The properties and assumptions are used to specify the pattern semantics (which security patterns are used and how).

Furthermore, the architecture, the pattern semantics and the scenarios are combined using our auto-combiner tool. The output contains the secure model, regular and attack scenarios, and observers coding the properties representing the security requirements of the architecture and the patterns. Finally, these outputs are used in a model-checker to verify that the model respects the properties in both regular and attack scenarios. The result of such verification is either a valid model or traces to help track violations.

For example, if we have the component c_1 in the architecture elements, and a security policy requiring to secure writing access to resource r_1 of c_1, the policy can be expressed using $protect(c_1, WRITE, r_1)$. When generating the secure model, we know that c_1 needs access protection, therefore we add the authorization pattern along with SAP_A and CHP. Since now c_1 has authorization pattern attached to it, each access to any of its resources should be verified. On the properties side, we add a property to verify that each access was either permitted because the resource is not protected for the specific operation (or an explicit permission is given to the source), or denied because the resource is protected (and no explicit permission is given). Finally, on the scenarios side, we add at least 2 different requests where one should be denied (lack of explicit permission) and one should be granted (explicit permission given). We can have other automatically generated scenarios (and some manual scenarios) to present different behaviors (exp. parallel requests) and make sure the properties are always respected.

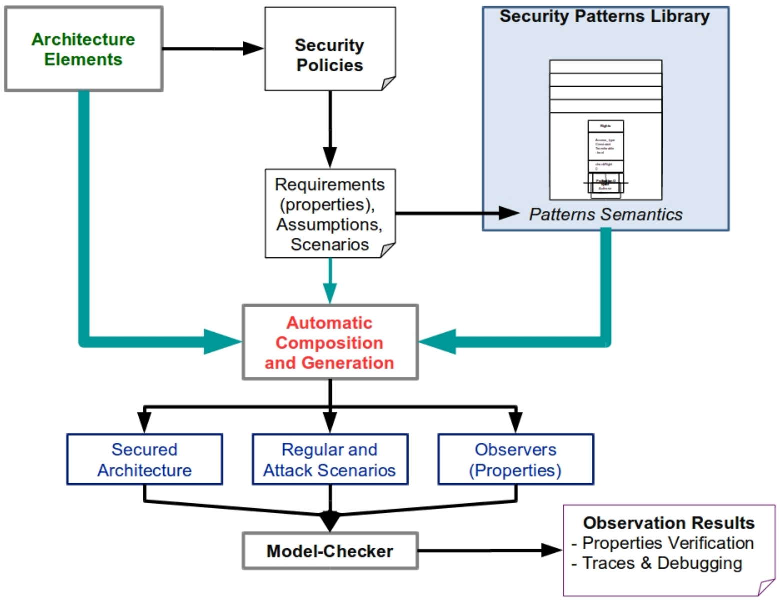

Architecture Elements
Security Policies
Security Patterns Library
Requirements (properties), Assumptions, Scenarios
Patterns Semantics
Automatic Composition and Generation
Secured Architecture
Regular and Attack Scenarios
Observers (Properties)
Model-Checker
Observation Results
- Properties Verification
- Traces & Debugging

Figure 3. Security pattern combination approach

The generated observers should consider the properties of the architecture, the properties regarding the security requirements, and the properties of the patterns used. For example, the architecture may have a specific behavior that needs to be respected, this should be translated into an observer to guarantee that the new behavior is correct. A security requirement regarding a protection of a resource also needs an observer to validate that the protection was indeed applied. Finally, the security patterns themselves have specific properties that should be respected.

In many cases, combining these different types of properties is a complicated task that can result in conflicts. This is not the same conflict we mentioned about multiple patterns used together: this conflict is about the requirements of different elements. For instance, the behavior of the architecture can be completely changed by applying a security policy. Consider that we have a model, and it should treat any message by forwarding it to a specific area. If the security policy limits access to this area, message treatment would now depend on the message itself.

Satisfying multiple requirements from different elements was a challenging part of our approach. However, to limit the automatic decision making, we consider the following order of property importance:

- The properties of the patterns are the most important, if these properties are not respected, the generated model can no longer be considered secure. Even if all other properties are respected or not, if the pattern is not functioning correctly, then any grantee given by the pattern can no longer be considered true.
- The security requirements are next, however, in implementation, we can combine these properties with the adjacent pattern properties. For example, the observer regarding a secure access requirement can be integrated in an $AUTH$ observer.
- Finally, the modified behavior of the model can have specific requirements that cannot be directly met due to the security measures. However, the new behavior is a combination between the original behavior, the patterns, and the requirements, which can be verified.

These are our own choices to resolve conflicts between properties. However, there can be other unresolved conflicts between properties which can be shown as properties violations which can be traced. To resolve these conflicts, either the security requirements need to be reduced or some services (model requirements) should be changed. The choice depends on each implementation, it also depends on the decision makers who decide which is more important, security or service. This question is not related to our work and is out of the scope of this paper. In our case, conflicts are resolved using the properties priorities stated above. Unresolvable conflicts did not appear during our experimentations.

6 VERIFICATION APPROACH

After specifying the architecture and the properties, we need to verify that these properties are respected in the secure architecture. In the case of complex systems, we cannot visually verify the code or the sequence diagram to ensure it respects its properties. A more suited solution is to use model checking tools to formally prove the correctness of a system. This approach relies on formal logic to obtain a proof that is absolute and undeniable.

To be able to formally represent and verify our model we need 3 essential elements (Figure 4).

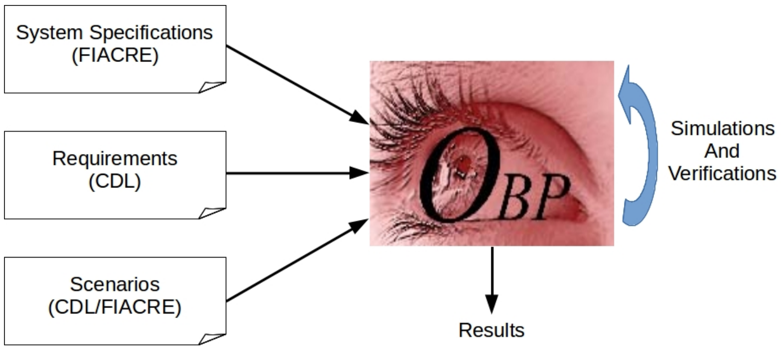


Figure 4. OBP explorer

We first need a language to model our SCADA system, the threat (attacker), and the security patterns. For this, we use the Intermediate Format for the Embedded Distributed Component Architectures (FIACRE) [4] which is a formally defined language that defines the behavior of a system considering time aspects. For the moment we are interested in the different possible behaviors of the system (with no consideration of time aspects).

We also need a way to formally present our properties that would ensure the fulfillment of the system requirements. We use the Context Description Language (CDL) [8] which, in addition to formalizing properties, can formalize scenarios to interact with the system. One of the main advantages of using CDL is the reduction in state explosion [9]. The scenarios can be either implemented using FIACRE or CDL, each implementation has its own advantages, in this article we have an example of each.

Finally, we need a model explorer to explore our model, and verify our properties. We use the Observer-Based Prover (OBP) explorer[1].

To use our model checking tools, we transform the different versions of our architecture (with or without each pattern) into a FIACRE code. We transform our scenarios and properties into CDL scenarios and CDL observers.

CDL uses a specific syntax to express properties which can be implemented using three different methods. While multiple methods can be used for the same purpose, each is more suitable for a specific situation:

[1] OBP documentation, tutorial and available tool: `http://www.obpcdl.org`

Invariant: Suitable in cases where only one configuration is concerned at a time. Exp. $pre_1 \Rightarrow pre_2$ means that at any configuration, if pre_1 is *true* then pre_2 should also be *true*.

Automaton: Suitable in cases where multiple configurations are concerned, it can detect events in the transitions from one configuration to another. Exp. $eve_1 \Rightarrow pre_2 \wedge eve_2$ means that if in a transition eve_1 happens, then pre_2 is *true* before the transition, and eve_2 happens in the same transition.

Liveness: Suitable when something leads to another. Exp. $pre_1 \Rightarrow \Diamond pre_2$, if pre_1 is *true*, pre_2 is eventually *true* in any following sequence.

For example, prt_CHP_3 uses the invariant method as follows:

$$\begin{aligned} &predicate\ p_1\ is\ accomplished(c_i, job),\\ &predicate\ p_2\ is\ checked(c_i, job) \wedge conform(c_i.policy, job),\\ &assert\ not(p_1 \wedge \neg p_2). \end{aligned} \tag{17}$$

This means that any configuration where p_1 is true and p_2 is false should be detected as a violation.

To observe prt_ARCH_1, we create the following automaton:

- $start \rightarrow e.Received(any) \rightarrow s1$ (1),
- $s1 \rightarrow e.state = Idle \rightarrow start$ (2),
- $s1 \rightarrow e.state \neq Idle \rightarrow reject$ (3).

The observer is at *start*, when e receives any message during a transition from one configuration to another, the observer goes to $s1$. During the same transition, the observer can make multiple shifts, from $s1$, if the state of e in the original configuration is *Idle*, the observer goes back to *start*. Since it has already detected the receiving event, it does not detect it again during this same transition.

It is important for the observer to go back to *start* in the case of correct behavior so it can detect the next time e receives a message. From $s1$, if the state of e in the original configuration is not *Idle*, the observer goes to *reject* so it can be easily traced. It stays in the *reject* state for the remainder of the sequence.

Figure 5 represents the property automaton, with p a priority to check one route before the other ($p1$ is tested before $p2$).

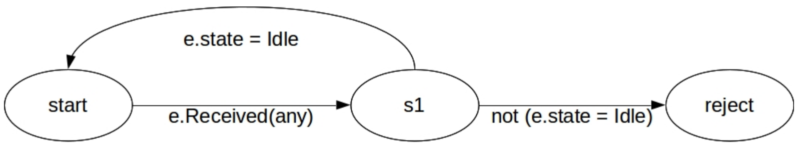


Figure 5. Initial property automaton

Finally, to observe *prt_SAP_C_2*, we use the liveness method as follows:

$$[\,]\ (|\ c_i@Received \land \{c_i.mess.source, c_i.mess.target\} \nsubseteq c_i.comps\ |$$

$$=><>|\ c_i@Sending \land c_i.mess = neg(c_i, c_i.mess))\ |).$$

If at some point c_i is at *Received*, and either the source of the message or its target is not in $c_i.comps$, then, eventually, c_i should be at *Sending* with a negative response. Since we already verify elsewhere that if c_i is at *Sending* it always sends a message, the combination leads to the property being correctly verified.

In the same fashion, we create observers for all of the other properties. Some of the more complicated properties require the use of more than one method at the same time.

7 EXPERIMENTATION

7.1 Process

In our experimentations, we tested multiple approaches on how to include the security patterns in the architecture. In this document we describe one of these approaches in which we use multiple states and transitions to apply the patterns. This modifies the automata (and the behavior) of both the communication component and the access component.

Figure 6 demonstrates the automaton of the secure communication component. From the *Received* state, in addition to verifying whether the message can be forwarded or not (basic verifications), we also verify if it is signed correctly. If this is the case, we go to *SAP* state which is responsible for testing the availability of the target. If the target is not available, a negative response is prepared and sent. If the target is available, *CHP* is called to verify the conformity of the message with the security policy. If the message conforms, it is forwarded, if not, *TrigAct* is called to make sure the correct countermeasure is applied, which is, in our case, sending a negative response.

Figure 7 demonstrates the automaton of the secure access component. From the *Received* state, in addition to verifying whether the target is the component itself, we also verify if the message is signed correctly. If this is the case, we go to *Respond* state which is responsible for preparing the correct response. In order to prepare the response, the resource needs to be accessed, and *SAP* is called to verify the availability of the resource. If it is not available, a negative response is prepared and sent. If the resource is available, *CHP* is called to test the conformity of the access demand with the security policy. If the demand conforms, the access is correctly achieved and a positive response is sent. If not, *TrigAct* is called to prepare a negative response as our chosen countermeasure.

The properties and scenarios can also be generated automatically. Each property generates an observer for each instance where it is necessary. For example, if we

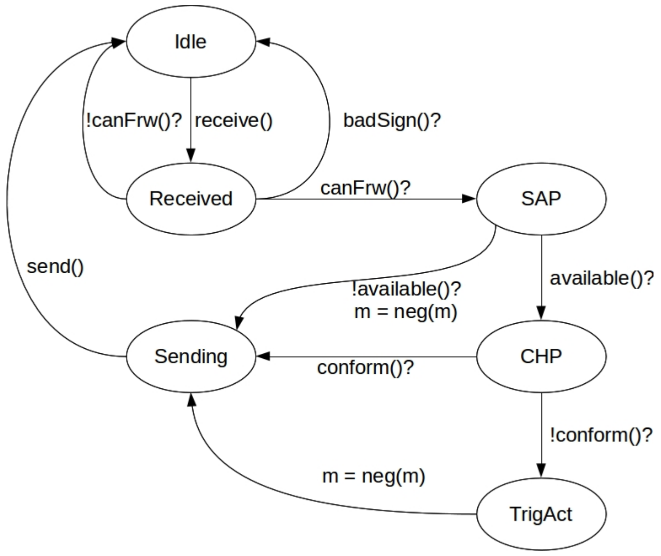


Figure 6. Secure communication component

have two access components c_a_1 and c_a_2, applying *prt_SAP_A_1* generates two observers *prt_SAP_A_1_*c_a_1 and *prt_SAP_A_1_*c_a_2.

This is applied differently depending on the property itself. For example, consider that the components above are both in an area secured by c_c_1. Applying the property *prt_SAP_C_1.a* creates an observer on c_c_1 itself to detect whether or not this happens. However, it should also verify the reaction of the components in the secure zone.

This means that the secure components should verify the signature correctly, which is observed using *prt_SAP_C_1.a_*c_a_1 and *prt_SAP_C_1.a_*c_a_1.

In this document we are more interested in validating our approach than verifying the generated models. Therefore, we use manually generated scenarios designed to verify certain aspects of the approach as well as to measure the complexity of the generated model.

We have multiple options to vary the scenarios:

- *Option_1*: Whether or not to have a stable system with an initial behavior. Which means, the system, without any external stimulation, already exchanges messages.
- *Option_2*: Whether or not to have normal interactions coming from the environment ($CLT1$ and $CLT2$ can send requests).
- *Option_3*: Whether or not to have attacking interactions coming from the environment (simply, an entity that is not recognized by the system, exp. $ATT1$).

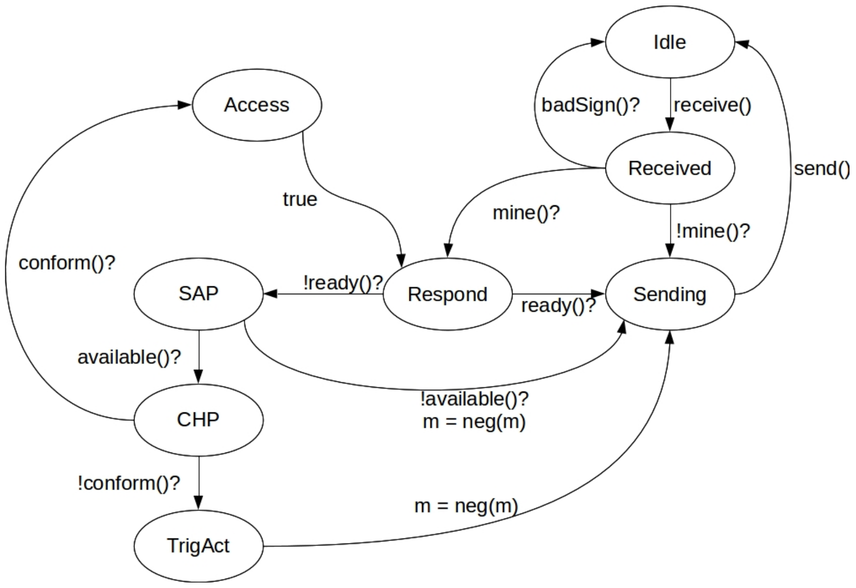


Figure 7. Secure access component

- *Option_4*: Whether or not to have attacking interactions from inside the system, we consider an unidentified entity (exp. $ATT2$) sending requests directly to $LC1$ without passing through GC and *Network*.

We can apply these options together. Consider that we have a system with inter-communications (*Option_1*), this would result in multiple possible configurations. We call this a stable system since each configuration can be reached from any other configuration. At any point (no matter which configuration the system is at), a client may send a valid request (*Option_2*) to which the response should be positive. An unidentified entity can also send a request (*Option_3*) which should result in a negative response. Finally, a request (*Option_4*) can be sent directly to an internal component (a component in a secure area), and since such requests were not signed by the *Firewall*, they should be ignored.

These scenarios are sufficient to verify all the different properties of the architecture, the security policy, and the patterns. However, the resulting configurations are too complicated for visual verification. Therefore, we start by simulating only one request using one of the options above, we alternate between requests and options to visually verify that the automatically generated model behaves as expected and the properties are verified correctly. Additionally, we add mistakes to the model to make sure the properties can detect these mistakes. Finally we can apply the full verification which can conclude that the secure model respects the properties.

7.2 Case Study

Consider we want to secure the model in Figure 8, it has the following components:

- GCS_1 and GCS_2: Access controllers which can access resources as well as forward messages. For the sake of our example, their access capabilities will not be necessary since there are no requests targeting them.
- NET_1 and NET_2: Communication controllers which can only forward requests. Both have a firewall when security is applied. NET_1 would protect every other component other than GCS_1, and NET_2 would protect every component of type device DEV_i.
- Four PLC_i each responsible for the corresponding device DEV_i. They ensure indirect access, each request to a PLC_i is treated at PLC_i before being forwarded to the device DEV_i. All PLC_i have secure access when security is applied.
- Four DEV_i which respond to the indirect access requests. Note that they do not apply security measures other then verifying messages signatures. Access rights verifications are run at PLC_i level.

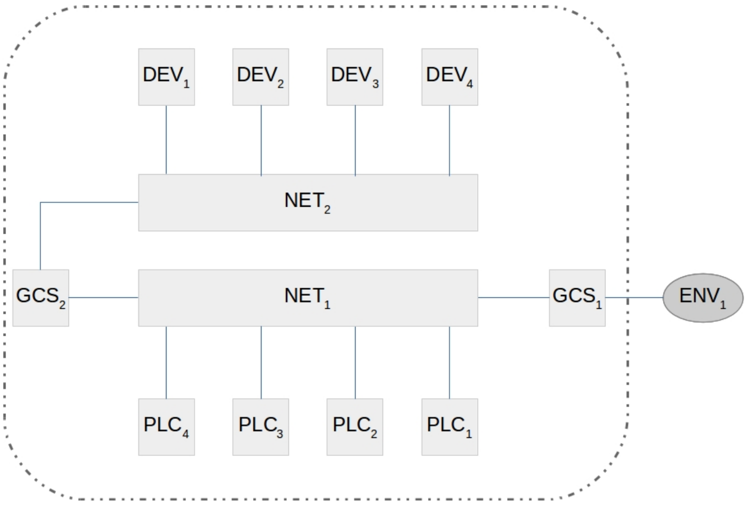


Figure 8. Simple SCADA model

For example, a request from CLT_i to PLC_i would go through GCS_1, NET_1, PLC_i, NET_1, GCS_2, NET_2, and DEV_i. The response would take the reverse route exactly. However, the request can be refused at NET_1 and NET_2 in the case of target unavailability or filter violation, or at PLC_i in the case of resource unavailability or access rights violation.

We apply a security policy on this model regarding message filtering and resource access. The policy details are:

- $\mathit{protect}(\mathit{GCS}_i, \mathit{any}, \mathit{any})$: Protect all resources of any GCS for any operation (we use i to apply the rule for any component of this type).
- $\mathit{protect}(\mathit{PLC}_i, \mathit{any}, \mathit{any})$: Protect all resources of any PLC for any operation.
- $\mathit{protect}(NET_1, \{\mathit{PLC}_i, \mathit{GCS}_2, NET_2, DEV_i\})$: Apply a firewall on NET_1 protecting all *PLCs* and *Devs*, and NET_2 and GCS_2.
- $\mathit{protect}(NET_2, \{DEV_i\})$: Apply a firewall on NET_2 protecting all *Devs.*
- $\mathit{addRule}(NET_2, (\mathit{any}, \{DEV_i\}, \mathit{any}), \{(\{\mathit{PLC}_i\}, \mathit{any}, \mathit{any})\})$: Add a filter rule to NET_2 refusing any message targeting DEV unless the source of this message is a *PLC.*

We create multiple scenarios which we simulate on both the secure and the insecure model. For verification purposes, we consider four different types of actors included in these scenarios:

- CLT_i: A legit client to which the response should be positive.
- SPM_i: A known non-legit entity whose requests should be stoppable at Firewalls. This is implemented using $\mathit{addRule}(NET_i, (\{\mathit{SPM}_i\}, \mathit{any}, \mathit{any}), \{\})$
- ATT_i: An unknown non-legit entity whose requests may pass through the Firewalls but has no access rights resulting in negative responses directly from the PLC_i (without forwarding the request to the device DEV_i).
- PEN_i: A penetration entity which would send requests directly from inside the system, either to PLC_i without passing through NET_1 or to DEV_i without passing through NET_2.

In the case of the insecure model, apart from PEN_i, we expect that the behavior of the other actor types is always the same. Since there is no way of filtering messages or applying access rights, these actor types are all the same to the system and their messages always result in positive responses. PEN_i requests would also have positive responses, however the route of the request is shorter since it is directly inserted inside the system.

When security is applied, the behavior is expected to differ for each actor type. CLT_i should stay the same since it is legit, the route of the request is the same, however the number of resulting configurations is expected to increase because of the states and transitions related to security. SPM_i requests are directly refused at NET_1 resulting in a negative response caused by the applied filter. ATT_i routes are longer than SPM_i and shorter than CLT_i since they are refused at PLC_i level. Finally, PEN_i messages are refused directly by the receiving component without calling *SAP*, *CHP*, nor *TrigAct*, since they are not signed and this is verified at *Received* state. This results in the shortest route for the PEN_i requests.

7.3 Property Verification

First, we investigated the insecure and the secure models by observing their behavior when stimulated by different possible actors, separated, or combined. During these observations we confirmed our previous expectation about both models. We also confirmed that the properties of the patterns, the security policy, and the initial model are all respected in the secure model. This is true not only for the properties mentioned in this paper but also for all the properties of the patterns.

As explained before, the properties are applied for each component concerned. For simplicity, we do not go into the details of each predicate and event used, nor how they are written: we are more interested in the format of the properties. As explained before, the properties are applied for some or all of the components depending on the property, some are only in the secure model while others are used in both models. In the following, we present some of the properties verified using our approach. Each property is written for only one component, other applications undergo slight modifications such as the name of the component.

- *prt_ARCH_1*: Applied on each and every component in both the insecure and the secure models. The following is its application on GCS_1:

$$\begin{array}{l}\textit{property prt\_ARCH\_1\_GCS}_1 \textit{ is}\\ \textit{start} -- // \textit{recv\_GCS}_1\textit{\_ANY} / -> \textit{wait};\\ \textit{wait} -- / \textit{pre\_GCS}_1\textit{\_Idle} // -> \textit{start};\\ \textit{wait} -- / \textit{not pre\_GCS1\_Idle} // -> \textit{reject}.\end{array} \tag{18}$$

  recv_GCS$_1$*_ANY* is detected when GCS_1 receives any message. At this point, if GCS_1 was at *Idle* (*pre_GCS*$_1$*_Idle*), the property is respected and the observer goes to *start* to wait for the next time GCS_1 receives a message. If not, the property is rejected and the configuration where this violation happened can be easily traced. Notice that, at any configuration, the observer is either at *start* or *reject*, this is important so that the property is verified each time GCS_1 receives a message.

- *prt_SAP_C_2_NET*$_2$ (Figure 9): Applied on any component applying *SAP_C* which are NET_1 and NET_2. For example, its application on NET_2 is as follows:

$$\begin{array}{l}\textit{property prt\_SAP\_C\_2\_NET}_2 \textit{ is}\\ \textit{start} --// \textit{eve\_NET}_2\textit{\_SAP\_true} / -> \textit{at\_Sap};\\ \textit{at\_Sap} --/ \textit{not pre\_NET}_2\textit{\_Available} /\textit{eve\_NET}_2\textit{\_Sending\_true} / -> \textit{ver};\\ \textit{at\_Sap} --/ \textit{not pre\_NET}_2\textit{\_Available} /\textit{eve\_NET}_2\textit{\_SAP\_false} / -> \textit{reject};\\ \textit{at\_Sap} --/ \textit{pre\_NET}_2\textit{\_Available} /\textit{eve\_NET}_2\textit{\_SAP\_false} / -> \textit{start};\\ \textit{ver} --/ \textit{pre\_NET}_2\textit{\_NAK} /\textit{send\_NET}_2\textit{\_ANY} / -> \textit{start};\\ \textit{ver} --/ \textit{not pre\_NET}_2\textit{\_NAK} /\textit{send\_NET}_2\textit{\_ANY} / -> \textit{reject}\end{array} \tag{19}$$

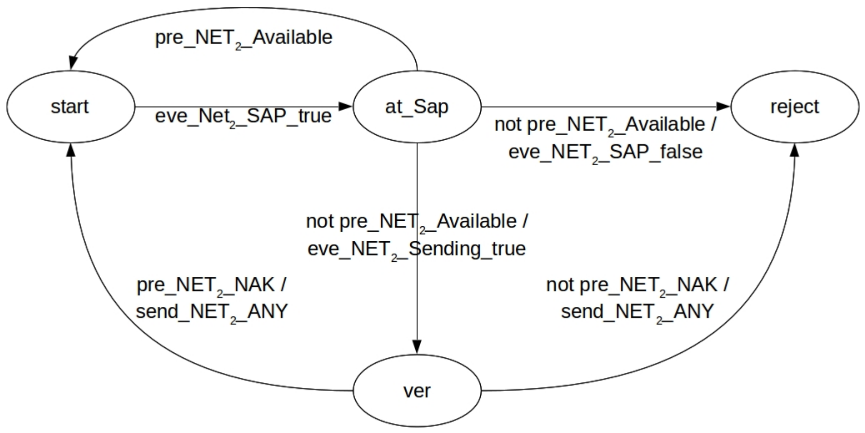

start
at_Sap
reject
ver
pre_NET₂_Available
eve_Net₂_SAP_true
not pre_NET₂_Available / eve_NET₂_SAP_false
not pre_NET₂_Available / eve_NET₂_Sending_true
pre_NET₂_NAK / send_NET₂_ANY
not pre_NET₂_NAK / send_NET₂_ANY

Figure 9. Application of Property_SAP_C_2 on Net2

This means that, if NET_2 goes to SAP state ($eve\_NET2\_SAP\_true$), the observer goes to *at_Sap*. If the target is not available (*not* $pre\_NET2\_Available$) and NET_2 goes to *Sending*, the observer goes to *ver*.

If the target is not available, and NET_2 changes its state (other than going to *Sending*), the property is rejected. However, if the target is available, the observer goes back to *start*. This is because this property is only interested in cases where the target is not available.

From *ver*, if NET_2 has a NAK message and sends it, the property is respected, and the observer goes back to start. However, if the message is not a NAK message, the observer goes to *reject* the moment this message is sent (not before).

Notice that in this example, we can have configurations where the observer is not at *start* nor *reject*, however it would still work each time SAP state is visited. This is because, from SAP, if NET_2 changes its state, the observer goes to *start*, *reject*, or *ver* if NET_2 went to *Sending*.

Moreover, from *Sending*, we already verify that the component cannot leave without sending a message. NET_2 either has $pre\_NET_2\_NAK$ true or false, so it either goes to *start* or *reject*. In short, at any point where NET_2 can go to SAP, the observer is either at *start* or *reject*.

- prt_CHP_2: Applied on any component applying CHP which is every component apart from DEV_i devices. For example, on PLC_1:

$$\begin{aligned}&\textit{property } prt\_CHP\_2\_PLC_1 \textit{ is}\\&[](\mid pre\_PLC_1\_CHP \wedge \neg pre\_PLC_1\_conform\\&=><>\mid pre\_PLC_1\_TrigAct)\mid)\end{aligned} \tag{20}$$

The property verifies that if, at any configuration, PLC_1 is at *CHP* and the request is not conform to the policy, eventually PLC_1 should go to *TrigAct*.

- prt_ARCH_4: Applied on any component, its application on DEV_1:

$$\begin{aligned}&\textit{property } prt\_ARCH\_4\_DEV_1 \textit{ is}\\&\textit{assert not}(pre\_DEV_1\_Idle \textit{ and } \neg DEV_1.mess == MESS\_NULL).\end{aligned} \tag{21}$$

Table 1 demonstrates for each property on which component this property is applied. The application depends on the situation of the component, its characteristics, and its type. In this table, when the component type is placed instead of the component, it means the property is applicable on each component of this type.

For example, $prt\_SAP\_C\_2$ is applied on any component of type *NET*, which are NET_1 and NET_2. Also, ENV_a means all entities in the environment.

Property	**Type**	**Applied on**	**Components**
prt_ARCH_1	safety	all components	GCS, NET, PLC, DEV
prt_ARCH_2	safety	all components	GCS, NET, PLC, DEV
prt_ARCH_3	safety	all components	GCS, NET, PLC, DEV
prt_ARCH_4	safety	all components	GCS, NET, PLC, DEV
$prt\_SAP\_C\_1.a$	authenticity	comps behind a firewall	GCS_2, NET_2, PLC, DEV
$prt\_SAP\_C\_1.b$	authenticity	comps behind a firewall	GCS_2, NET_2, PLC, DEV
$prt\_SAP\_C\_2$	availability	secure communication comps	NET
$prt\_SAP\_C\_3$	confidentiality	all entities	GCS, NET, PLC, DEV, ENV
$prt\_SAP\_C\_4$	integrity	all entities	GCS, NET, PLC, DEV, ENV
$prt\_SAP\_A\_1$	authenticity	secure access comps	GCS, PLC
$prt\_SAP\_A\_2$	availability	secure access comps	GCS, PLC
prt_CHP_1	availability	secure comps	GCS, NET, PLC
prt_CHP_2	availability	secure comps	GCS, NET, PLC
prt_CHP_3	authenticity	secure comps	GCS, NET, PLC
prt_AUTH_1	authenticity	secure access comps	GCS, PLC
prt_FWLL_1	authenticity	secure communication comps	NET

Table 1. Properties application

7.4 Complexity Measurements

We use dedicated scenarios based on the simple actors of type CLT_i. Here we are interested in analyzing the complexity of our approach regarding the resulting configurations. Therefore, scenarios should be normalized in a way that each request takes (by itself) exactly the same amount of transitions to be fulfilled. When combined, multiple requests from the same actor or multiple actors would have a great impact on the resulting configurations. The main objective here is to confirm that the impact of applying the security patterns is limited and predictable.

For the complexity measurements, we do not use the other actor types since we have already confirmed that, when used on the secure model, their routes are less important in terms of complexity. We have multiple scenarios noted A_i_j where i is the number of actors (CLT_i) included in the scenario and j is the number of messages sent by this actor. Multiple actors can send requests in parallel, however, each actor waits for a response before sending its following request. We have a maximum of four requests per actor, which are $(READ, res_1)$, $(WRITE, res_1)$, $(READ, res_2)$, and $(WRITE, res_2)$, We also have a maximum of four actors each sending requests to the corresponding PLC_i. Each received message cannot cause more than one message sent (either forward the message or send a response or ignore). Therefore, we know in advance that it is impossible for our model to have more than 4 messages in its channels all at once. We fixed the size of all channels to 5 and observed that no channel was ever full, this helped in the normalization between the scenarios so they would not be affected by the size of the channels.

Table 2 contains the number of configurations, the number of transitions, and the depth (D), for each scenario and for both secure and insecure models.

Figure 10 demonstrates the ratios between secure and insecure models in terms of number of configurations, number of transitions, and depth. We compute these ratios for different scenarios with different number of actors and number of messages per actor. The depth ratio is completely stable. This is because the added number of transitions to respond to a request is stable for legit requests. Both configuration and transition ratios are stable in the case of one actor. These ratios increase by around 0.3 points each time we add a new actor. However, their increase due to messages per actor increase is much less significant.

These results show that using the security patterns, the complexity of our model increases but in a stable and predictable fashion. Note that, in cases where both legit requests and non-legit requests are used, the complexity may decrease since most non-legit requests are treated with less transitions in the case of the secure model than the insecure one.

8 CONCLUSION

Current SCADA systems have a shortage in security measures, while the interest of attackers in these systems is constantly increasing. We do not limit our work to the SCADA domain, however, we are interested in considering it in our simulations since it represents a challenge when it comes to information security. The reflections developed in this work are generic and could be adapted to other types of architectures.

	Insecure Model			Secure Model		
Scn	**nb. confs**	**nb. trans**	**D**	**nb. confs**	**nb. trans**	**D**
A_1_1	49	48	48	63	62	62
A_1_2	97	96	96	125	124	124
A_1_3	145	144	144	187	186	186
A_1_4	193	192	192	249	248	248
A_2_1	2 344	4 415	96	3 686	6 977	124
A_2_2	11 207	21 424	192	17 755	34 028	248
A_2_3	26 590	51 027	288	42 208	81 153	372
A_2_4	48 493	93 224	384	77 045	148 352	496
A_3_1	110 137	299 913	144	204 621	560 051	186
A_3_2	1 394 467	3 871 044	288	2 623 843	7 301 556	372
A_3_3	5 396 320	15 039 747	432	10 183 726	28 435 959	558
A_3_4	13 605 189	37 982 760	576	25 707 157	71 886 920	744
A_4_1	5 114 765	17 939 618	192	10 953 437	38 624 884	248
A_4_2	180 143 913	646 742 856	384	392 578 705	1 412 820 112	496
A_4_3	1 150 191 955	4 145 224 977	576	2 515 317 972	9 082 222 251	744

Table 2. Complexity measurements

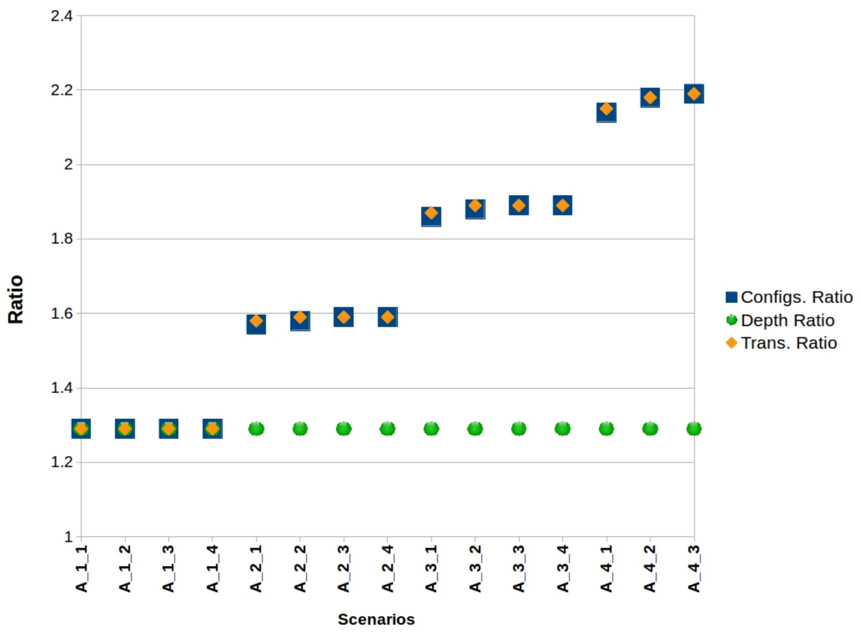


Figure 10. Ratios between secure and insecure models

[12] proposes to use security patterns to strengthen the security of SCADA architectures. Security patterns allow us to implement current best found solutions to resolve specific security issues. Our goal was to provide rules to generate a secure model based on an insecure one, some security requirements, and some security patterns to satisfy these requirements. These rules can be used to automatically generate a secure model based on the insecure one.

We also looked into tools to formally verify validate the generated secure model using model checking techniques. This validation goes through the verification of the properties of the model, the properties of the security patterns, and the properties related to the security requirements.

Our experiments have strengthened our ability to drive, by this method and technique, a process of integration of the security patterns and formal validation of the generated model. But given the large number of properties and scenarios involved in this process, and to be taken into account in the verification process, the approach only makes sense if we are able to generate these sets of properties and scenarios.

During the experiments, we limited ourselves to four security patterns. But a similar approach (formalization, composition, verification) could be used for the integration of other patterns described in the literature. We also focused on sets of properties that could be extended for each pattern.

Currently we are working on improving the combination and generation rules. In our work, we opted for implementation choices which can be studied and optimized further. In addition, we are updating our library to include more patterns. Finally, we seek to include an attack library which, in addition to having generic attack scenarios, allows us to have scenarios representing specific attacks and threats.

Acknowledgement

Partially supported by the Research Grant of the French Government Defense Procurement and Technology Agency – Information Department (DGA-MI).

REFERENCES

[1] Alencar, P.—Cowan, D.—Dong, J.—Lucena, C.: A Pattern-Based Approach to Structural Design Composition. Proceedings of the Twenty-Third Annual International Computer Software and Applications Conference (COMPSAC '99), IEEE, 1999, pp. 160–165, doi: 10.1109/CMPSAC.1999.812694.

[2] Alencar, P. S. C.—Cowan, D. D.—Lucena, C. J. P.: A Formal Approach to Architectural Design Patterns. In: Gaudel, M. C., Woodcock, J. (Eds.): Industrial Benefit and Advances in Formal Methods (FME '96). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1051, 1996, pp. 576–594, doi: 10.1007/3-540-60973-3_108.

[3] Avalle, M.—Pironti, A.—Sisto, R.: Formal Verification of Security Protocol Implementations: A Survey. Formal Aspects of Computing, Vol. 26, 2014, No. 1, pp. 99–123, doi: 10.1007/s00165-012-0269-9.

[4] Berthomieu, B.—Bodeveix, J.-P.—Farail, P.—Filali, M.—Garavel, H.—Gaufillet, P.—Lang, F.—Vernadat, F.: Fiacre: An Intermediate Language for Model Verification in the Topcased Environment. 4th European Congress ERTS Embedded Real Time Software (ERTS 2008), Toulouse, France, 2008, 8 pp.

[5] Byres, E.—Lowe, J.: The Myths and Facts Behind Cyber Security Risks for Industrial Control Systems. Proceedings of the VDE Kongress, Vol. 116, 2004, pp. 213–218.

[6] Clarke, E. M.—Grumberg, O.—Peled, D.: Model Checking. MIT Press, 1999.

[7] Delessy-Gassant, N.—Fernandez, E. B.—Rajput, S.—Larrondo-Petrie, M. M.: Patterns for Application Firewalls. Proceedings of the Conference on Pattern Languages of Programs (PLoP), 2004.

[8] Dhaussy, P.—Boniol, F.—Roger, J.-C.—Leroux, L.: Improving Model Checking with Context Modelling. Advances in Software Engineering, Vol. 2012, 2012, Art. No. 547157, 13 pp., doi: 10.1155/2012/547157.

[9] Dhaussy, P.—Roger, J.-C.—Boniol, F.: Reducing State Explosion with Context Modeling for Model-Checking. Proceedings of the 2011 IEEE 13th International Symposium on High-Assurance Systems Engineering (HASE '11), 2011, pp. 130–137, doi: 10.1109/HASE.2011.24.

[10] Dong, J.—Alencar, P. S. C.—Cowan, D. D.: Ensuring Structure and Behavior Correctness in Design Composition. Proceedings of the Seventh IEEE International Conference and Workshopon on the Engineering of Computer-Based Systems (ECBS 2000), 2000, pp. 279–287, doi: 10.1109/ECBS.2000.839887.

[11] Dong, J.—Peng, T.—Zhao, Y.: Model Checking Security Pattern Compositions. Proceedings of the Seventh International Conference on Quality Software (QSIC '07), IEEE, 2007, pp. 80–89, doi: 10.1109/QSIC.2007.4385483.

[12] Fernandez, E. B.—Larrondo-Petrie, M. M.: Designing Secure SCADA Systems Using Security Patterns. Proceedings of the 2010 43rd Hawaii International Conference on System Sciences (HICSS '10), IEEE, 2010, pp. 1–8, doi: 10.1109/HICSS.2010.139.

[13] Fernandez, E. B.—Larrondo-Petrie, M. M.—Seliya, N.—Delessy-Gassant, N.—Herzberg, A.: A Pattern Language for Firewalls. 2003.

[14] Fernandez, E. B.—Pan, R.: A Pattern Language for Security Models. Proceedings of the PLoP Conference, Vol. 1, 2001.

[15] Fovino, I. N.—Coletta, A.—Carcano, A.—Masera, M.: Critical State-Based Filtering System for Securing SCADA Network Protocols. IEEE Transactions on Industrial Electronics, Vol. 59, 2012, No. 10, pp. 3943–3950, doi: 10.1109/TIE.2011.2181132.

[16] Fovino, I. N.—Coletta, A.—Masera, M.: Taxonomy of Security Solutions for the SCADA Sector. Project ESCORTS Deliverable, Vol. 2, 2010.

[17] Hafiz, M.—Adamczyk, P.—Johnson, R. E.: Growing a Pattern Language (for Security). Proceedings of the ACM International Symposium on New Ideas, New

Paradigms, and Reflections on Programming and Software (Onward! 2012), 2012, pp. 139–158, doi: 10.1145/2384592.2384607.

[18] Igure, V. M.—Laughter, S. A.—Williams, R. D.: Security Issues in SCADA Networks. Computers and Security, Vol. 25, 2006, No. 7, pp. 498–506, doi: 10.1016/j.cose.2006.03.001.

[19] Krutz, R. L.: Securing SCADA Systems. John Wiley and Sons, 2005.

[20] Viega, J.—McGraw, G.: Building Secure Software: How to Avoid Security Problems the Right Way. 1st Edition. Addison-Wesley Professional, 2001.

[21] Schumacher, M.—Fernandez-Buglioni, E.—Hybertson, D.—Buschmann, F.—Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. John Wiley and Sons, 2013.

[22] Taibi, T.—Ngo, D. C. L.: Formal Specification of Design Pattern Combination Using BPSL. Information and Software Technology, Vol. 45, 2003, No. 3, pp. 157–170, doi: 10.1016/S0950-5849(02)000195-7.

[23] Wang, Y.: sSCADA: Securing SCADA Infrastructure Communications. International Journal of Communication Networks and Distributed Systems, Vol. 6, 2011, No. 1, pp. 59–78, doi: 10.1504/IJCNDS.2011.037328.

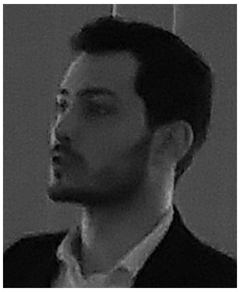
[24] Wassermann, R.—Cheng, B. H.: Security Patterns. Proceedings of the PLoP Conference, Michigan State University, 2003, Citeseer.

[25] Yoder, J.—Barcalow, J.: Architectural Patterns for Enabling Application Security. PLoP '97 Conference, Urbana, Vol. 51, 1998, Art. No. 61801.

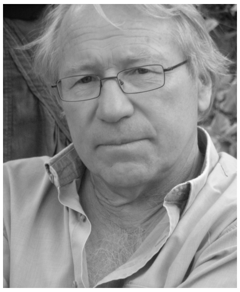
[26] Yoshioka, N.—Washizaki, H.—Maruyama, K.: A Survey on Security Patterns. Progress in Informatics, Vol. 5, 2008, pp. 35–47, doi: 10.2201/NiiPi.2008.5.5.

[27] Zhu, B.—Joseph, A.—Sastry, S.: A Taxonomy of Cyber Attacks on SCADA Systems. Proceedings of the 2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing (iThings/CPSCom 2011), IEEE, 2011, pp. 380–388, doi: 10.1109/iThings/CPSCom.2011.34.

[28] Zhu, B.—Sastry, S.: SCADA-Specific Intrusion Detection/Prevention Systems: A Survey and Taxonomy. Proceedings of the 1st Workshop on Secure Control Systems (SCS), 2010.



Fadi Obeid is consultant in cyber security, working on different cyber security subjects, for multiple clients. He studied cyber security in the University of Limoges, France, and obtained his master's degree in 2013. He started his Ph.D. at ENSTA Bretagne in 2014 where he worked on model checking security patterns implementations. In parallel to his Ph.D. at ENSTA Bretagne, he created a new encryption protocol dedicated to the low vocabulary of SCADA and IoT communications. After finishing his Ph.D. in 2018, he started working as Consultant for Alter Solutions. He has a polyvalent profile, including model checking, cryptography, side channel analysis, functional security.



Philippe Dhaussy is Professor at CNRS Lab-STICC within ENSTA Bretagne. His expertise and his research interests include model-driven software engineering, formal validation for real time systems and embedded software design. He has an engineer degree in computer science from ISEN (French Institute of Electronics and Computer Science) in 1978 and received his Ph.D. in 1994 at Telecom Bretagne (France) and his HDR in 2014. From 1980 to 1991, he had been software engineer and technical coordinator in consulting companies (Atlantide group), mainly in real-time system developments. He joined ENSTA-Bretagne in 1996 as Professor. He has over 100 publications in the areas of software engineering and computer science. He co-supervised ten Ph.D. students and he was and still is involved in several research projects as a work package coordinator.