# Model Checking Security Pattern Combination

Fadi Obeid
Université Bretagne Loire
Laboratoire Lab-STICC
UMR CNRS 6285
ENSTA Bretagne, Brest
Email: fadi.obeid@ensta-bretagne.org

Philippe Dhaussy
Université Bretagne Loire
Laboratoire Lab-STICC
UMR CNRS 6285
ENSTA Bretagne, Brest
Email: philippe.dhaussy@ensta-bretagne.fr

***Abstract*—A security pattern is a reusable solution for a specific security issue. Based on an insecure model, and using a combination of security patterns, we can generate a model respecting some security requirements constituting a security policy. The resulting model needs to fulfill the security requirements without affecting the original functionalities and services. The security patterns need to be consistent with each others, as well as the model, they also need to cover the whole security spectrum resulting in completeness. We can use model checking techniques in order to insure the correct functionality, as well as the consistency and completeness of the generated model. In this paper, we describe our approach to combine an architectural model with security patterns to generate a secure model. This model is later verified using model checking techniques to validate the properties of the model itself as well as the used patterns. Finally, using an experimental use case, we demonstrate the possible spatial complexity of our approach.**

***Keywords:*** **Security Patterns, Formal Verification, Model Checking**

## I. Introduction

A security pattern is a reusable solution to a recurring security problem. It provides detailed guidelines for implementing an architectural solution for a specific security problem. Security patterns should be considered as methodological tools to describe technical solutions related to security in a specific context. They impose decisions that must be taken into consideration when designing architectures. They facilitate communication between experts and non-experts.

Many security patterns have been proposed in the literature [17]. Associated with these patterns, some authors have proposed methodologies for their implementation and integration into software architecture models. The description of the patterns includes a description of their requirements which must be respected when integrated into an architecture.

In this work, we study methods and concepts for generating models that comply with specific security policies and security requirements. These models are verified later using model checking techniques. In order to achieve this objective, we define the meta model of the insecure architectures, and create a library of security patterns. Based on the model, the library, and a security policy, we generate a secure model following specific combination rules. Finally, the result is used as an input of a model checking in order to validate the properties of the model, the patterns, and the security policy.

Partially supported by Research Grant from Brest Metropolis as well as the French Government Defense Procurement and Technology Agency - Information Department (DGA-MI).

Although we work on an abstract level, the application part of our work considers supervisory control and data acquisition (SCADA) systems. These systems control many critical infrastructures, such as nuclear facilities and chemical plants. The SCADA environment, due to its limitations and constraints, is lacking satisfactory security measures.

This paper is organized as follows: In section II we provide references to related work on security patterns, formal verification, and SCADA. We describe our meta model in section III. This meta model is inspired by SCADA architectures. In section IV we define the library for the used security patterns. In section V we explain our approach to combine the patterns with the model. We demonstrate our verification tools in section VI. Finally, we provide an experiment to test our approach in section VII, this experimentation provides a brief complexity analysis as well.

## II. Related Work

A survey on security patterns can be found in [24]. [23] and [11] introduce many security patterns. A full description of integrating security patterns into systems is found in [17]. [14] presents a pattern language unifying and classifying all published security patterns at the time. Many security patterns are explained with their formal constraints in [21], enabling the verification of these patterns

A survey on formal verification of security protocol implementations is found in [4], focusing on the automatic verification of models close to the real implementation. Many research efforts were made on model checking of design patterns [3], [8], [2], [19]. [9] is directly related to our work since it verifies security pattern combinations. In this paper, the authors explain how wrongly combining security patterns may result in several errors.

Many studies [16], [15], [25], [20], [12] address the security issues in SCADA. These studies provide theoretical solutions, security guidelines, and different approaches to improve security in SCADA. [13] and [26] propose SCADA-specific security solutions and SCADA-specific IDS respectively. [10] is directly related to the application part of our work since the authors of this article propose using security patterns to design secure SCADA systems.

## III. META-MODEL

In this article, we consider the reader to already have basic knowledge about a SCADA architecture [5]. For the sake of our work, let's consider only the following components in a SCADA: (1) A $PLC$ which sends commands and receives information about the status of its section. (2) An $Actuator$ which executes commands. (3) A $Sensor$ which sends data regarding a physical condition such as temperature, etc.

In addition, we have communication devices responsible for separating the sections, and also, forwarding messages between components and sections. Finally, we have external clients (relative to the architecture) sending requests to internal components, these requests are in the form of commands to modify something, or demands of information about a section.

If we abstract this architecture, to a high level, we end up with two types of components: (1) communication components, responsible for communicating messages and separating sections. (2) access components which provide the services for reading or writing resources.

For example, an actuator executes a command by writing a resource (change the level of a valve for example). A sensor sends data by reading a resource. Finally, we have the external clients which send requests and receive responses. These requests are either to write a resource (modify something), or read a resource (demand information).

Figure 1 demonstrates our meta model, including: The communication $ComComp$ and access $AccComp$ components, and the clients $Client$. Messages are exchanged using $FIFO$ channels. And finally, the message divided into two parts: (1) The communication info part $comInfo$ which is visible to anyone. (2) The data part $Data$ which is only visible to the target of the message. Finally, both parts are authentic, meaning that a component cannot modify any part of the message without modifying its source. In other words, we considered an already implemented signature based communication, with the $Data$ part being encrypted.

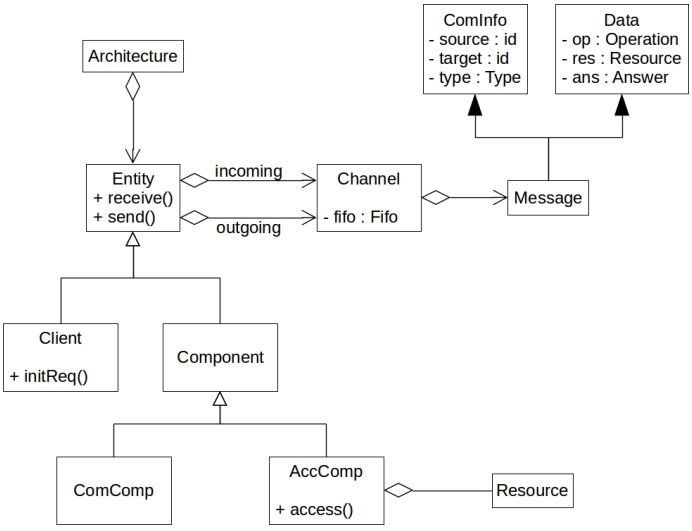


Figure 1. Meta Model

Figure 2 demonstrates the minimal automate for each entity. The client $Client$ initiates a requests $initReq()$, sends it $send()$, and waits for a response in the $WaitResp$ state. Once a response is received $receive()$, the client goes back to its initial state $Idle$ so it can initiate a new request. The communication component $ComComp$ receives a message $receive()$ and goes to the state $Received$. If the message can be forwarded $canFrw()$ the component goes to $Sending$, sends the message $send()$ and goes back to $Idle$ to receive other messages. If not $!canFrw()$, the message is ignored and the component goes back to $Idle$. In the same fashion, the access component $AccComp$ has its own minimal required automate to provide access to resources.

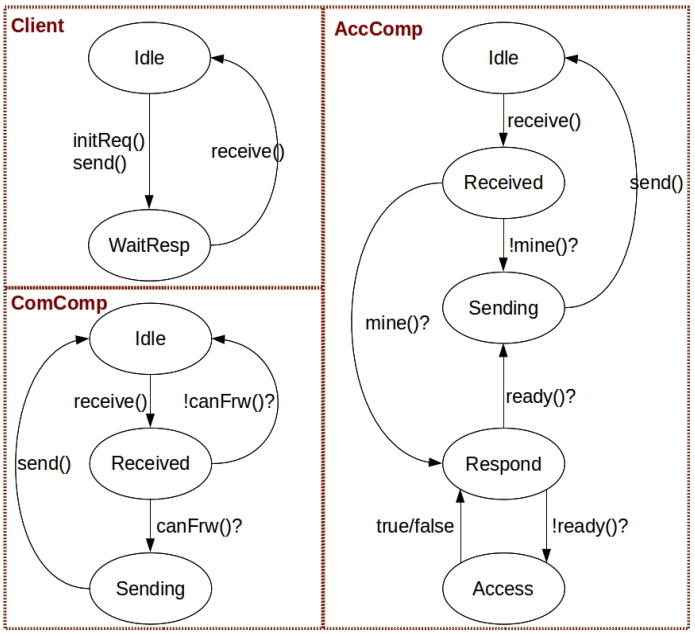


Figure 2. Minimal Entities Automates

## IV. SECURITY PATTERNS

In general, securing an architecture involves the integration of several security patterns. The management of the entire security policy is therefore shared between several patterns.

We focus on four patterns that we can extend later on the same methodological bases to include additional security measures. These patterns work together (fig. 3) to achieve a fully secure access:

- The $SingleAccessPoint\ (SAP)\ pattern$ unifies the access ports to mediate security checks through a single access point.
- The $CheckPoint\ (CHP)\ pattern$ is dedicated to checking whether the security policy is respected or not, and applying countermeasures.
- The $Authorization\ (AUTH)\ pattern$ implements security measures for access rights.
- The $FireWall\ (FWLL)\ pattern$ implements security measures for restricting and filtering messages.

Figure 3 demonstrates how these patterns work together to secure a component. Whereas, $SecPol$ is $AUTH$ in the case of an access component $AccComp$ and $FWLL$ in the case of a communication component $ComComp$. In case of a policy violation, the corresponding counter measure is called. In this article, $SAP$ and $CHP$ are always used when $AUTH$ or $FWLL$ are used. Therefore, the composition $\{SAP, CHP, AUTH\}$ can be implemented on an access component to secure access to the its resources. The composition $\{SAP, CHP, FWLL\}$ is used on a communication component to secure communication between its associated components (in a supposed secure zone) and other entities.

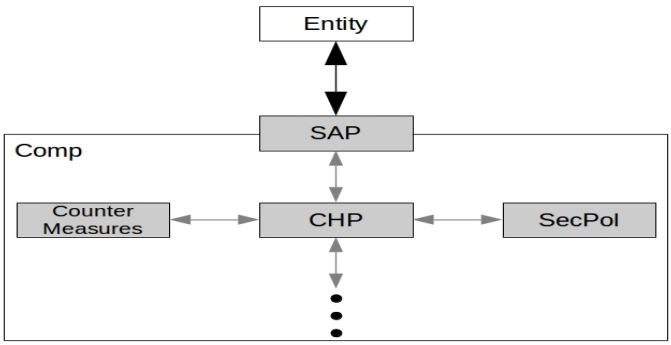


Figure 3. Specified Patterns

Each of these patterns is defined by its name, description, functionality, and security properties. Other aspects such as structure and behavior are also needed, but not included in this article. In the following, we describe the $SAP$ pattern, followed by a formalization of some of its security properties.

### A. Single Access Point

The $SAP$ introduced by [22] aims to implement a single access point to improve control and monitoring of inputs. $SAP$ calls other patterns such as $CHP$ to perform checks on information passing through this point.

In addition, $SAP$ has some basic controls. In the case of our meta-model, we can consider the $SAP$ in the case of communication components and access components. In the case of an access component, the $SAP$ checks whether the demanded resource in a request is available, if not, it can directly send a negative response. In the case of a communication component, the $SAP$ checks whether the demanded target is available, if not, it can directly send a negative response.

The application of this pattern therefore prevents external entities from directly accessing resources or communicating with targets. The single access point is an appropriate place to possibly capture a log of the historical ($Log$) access. This data can be useful for checking the sequencing of accesses according to their rights.

We identify two types of $SAP$:

- $SAP\_C$ unifying the access point to accessing a secure zone associated with a secure communication component.
- $SAP\_A$ unifying the access point to accessing resources in a secure access component

The functionalities of $SAP$ are:

- Control: If $SAP$ receives a message, it is controlled:
  - In the case of $SAP\_C$, the control verifies if the requested target is available.
  - In the case of $SAP\_A$, the control verifies if the requested resource is available.
- Response: In the case of target (or resource in the case of $SAP\_A$) unavailability, a negative response is sent.

### B. Formal Properties

A security policy is described by security objectives that are a set of security properties that can be defined in a formal form. These properties are grouped into three large classes [1]: Confidentiality, Integrity, and Availability. Each property represents the conditions that the system must meet. An incorrect definition, or the partial application of a policy, can cause the system to be in an unsafe state. While security patterns fulfill the security requirements of a security policy, they also have their own requirements that need to be fulfilled.

In the case of $SAP$, a classification of requirements is proposed in [21]. We consider some of these requirements in the case of a communication component integrating $SAP$.

Firstly, we consider the following formal notions:

- $\mathcal{M}ess$: All possible messages.
- $\mathcal{E}nts$: All entities (components and clients).
- $\mathcal{SC}comps$: All secure communication components.
- $c.comps$: all components associated with the secure communication component $c$.
- $sent(c, m)$: $true$ if component $c$ sent message $m$.
- $received(c, m)$: $true$ if component $c$ received $m$.
- $controlled(c, m)$: $true$ if component $c$ controlled (at $SAP$ level) message $m$.
- $neg(c, m)$: The negative response to the request in message $m$, produced by component $c$ (using $c$ as a source).

The properties are then formalized as follow:

- Authenticity: If a message from outside a secure zone, is received inside the secure zone, it should have been controlled by the secure communication component securing this zone.

  **prt_SAP_C_1:**

  $$\forall\, m \in \mathcal{M}ess,\ \forall\, c_c \in \mathcal{SC}comps,\ \forall\, c \in c_c.comps,\ received\ (c,\ m) \wedge m.source \notin c_c.comps \Rightarrow controlled\ (c_c,\ m)$$

- Availability: If a message is received by a secure component, and is targeting the secure zone (its source is not inside the secure zone), and the target is not available (its target is not inside the secure zone), then, the secure component should send a negative response to this message.

  **prt_SAP_C_2:**

  $$\forall\, m \in \mathcal{M}ess,\ \forall\, c_c \in \mathcal{SC}comps,\ received\ (c_c,\ m) \wedge \{m.source, m.target\} \nsubseteq c_c.comps \Rightarrow \Diamond\ sent\ (c_c,\ neg\ (c_c,\ m))$$

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

  **prt_SAP_C_3:**

  $\forall\, m \in \mathcal{M}ess, \forall\, c_c \in \mathcal{SC}comps, \forall\, e \in \mathcal{E}nts, received\ (e,\ m) \wedge \{m.source, m.target\} \subseteq c_c.comps \Rightarrow e \in c_c.comps$

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

  **prt_SAP_C_4:**

  $\forall\, m \in \mathcal{M}ess, \forall\, c_c \in \mathcal{SC}comps, \forall\, e \in \mathcal{E}nts, sent\ (e,\ m) \wedge \{m.source, m.target\} \subseteq c_c.comps \Rightarrow e \in c_c.comps$

In our study, we formalized dozens of properties for the four studied security patterns. In our experiments, we instantiate all these properties on the cases of modeled architectures in which the security mechanisms are integrated. Then, during the verification process of model-checking, we translate all instances of properties into invariants, observers, or LTL formulas.

## V. Security Pattern Composition

Figure 4 demonstrates our composition approach. To generate a secure architecture, we need to specify the architecture elements based on the already discussed meta-model. These elements are accompanied by a security policy defining the security requirements to be fulfilled. Based on these requirements, we can define the patterns semantics (which patterns to use and where). We also extract some scenarios to stimulate the generated architecture.

Finally, we can compose the architecture, with the patterns semantics, to generate a secure architecture. We use the assumptions and scenarios with the architecture to generate regular and attack scenarios. Finally, based on the requirements, we generate invariants and observers to validate the secure architecture. The generated tuple (Secure architecture, scenarios, properties) is then used as the input of our model checker which would verify the properties and validate the secure architecture.

We formally describe the security policy which is important in order to apply the correct patterns in the correct places. In this article, we provide a simple, yet sufficient, formal description of the security policy.

Consider we have the access components $c_1$ and $c_2$, the communication component $c_3$, the client $e_1$ and finally the resource $r_1$ of $c_1$. A policy can be described as follows:

- $protect(c_1, READ, r_1)$ provokes $AUTH$ pattern on $c_1$ to protect reading resource $r_1$.
- $protect(c_3, \{c_1, c_2\})$ provokes $FWLL$ pattern on $c_3$ considering a secure zone containing $c_1$ and $c_2$.
- $allow(e_1, c_1, READ, r_1)$ conditions the policy of $c_1$ to allow $e_1$ to read resource $r_1$.

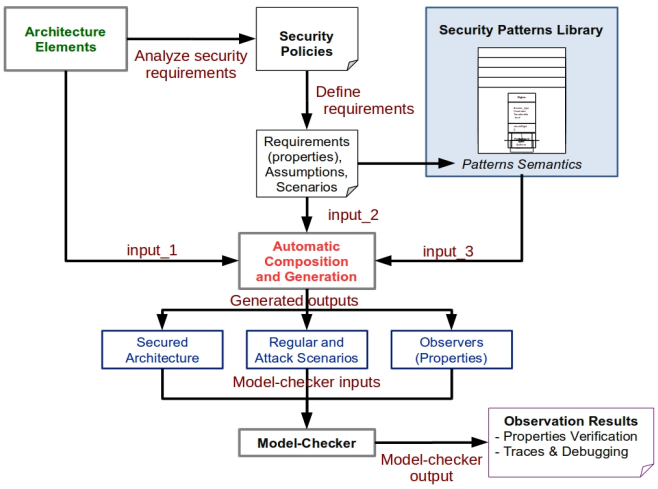


Figure 4. Security Pattern Composition Approach

- $rule(c_3, comInfo = (ANY, c_1, ANY), except = \{(e_1, c_1, Req)\})$ conditions the policy of $c_3$ to prohibit messages with any source, if the target is $c_1$, and no matter the type of the message. With the exception of requests from $e_1$ to $c_1$. $ANY$ is a wild-card used to include any variable of a type.

When a pattern is needed for a component, this component is transformed based on its minimal automate.

Figure 5 demonstrates how the minimal automate of the communication component $ComComp$ (defined in figure 2) is modified with this transformation. From the $Received$ state, the component verifies the signature. If the signature is correct, $SAP$ is called to verify the availability of the target. If the target is not available $!available()$, a negative response is prepared $m = neg(m)$. If the target is available $available()$, $CHP$ is called to verify the conformity of the message with the security policy. In case the message is conform $conform()$, it is forwarded, if not $!conform()$, $TrigAct$ is called to apply the correct countermeasure. For simplicity, the countermeasure here is preparing and sending a negative response. We have formalized transformation rules to automatically generate the new automata based on the original insecure ones.

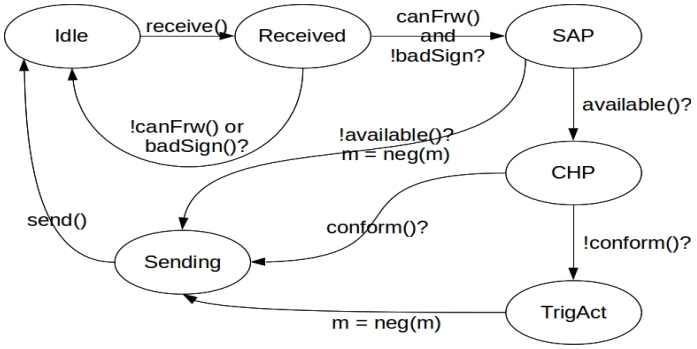


Figure 5. Secure Communication Component

## VI. Verification Tools

We carry out verifications using the OBP tool [1]. Figure 6 gives an overview of this tool.

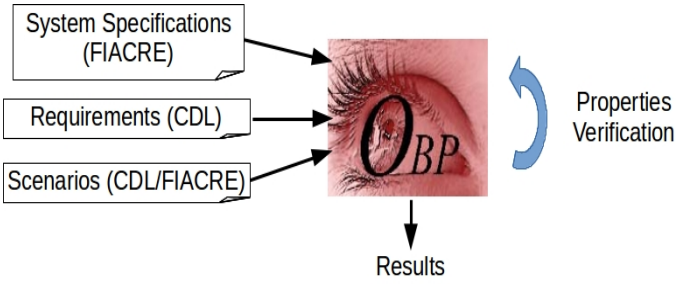


Figure 6. OBP Verification Tool

To carry out verifications, we generate the models in FIACRE [2] which specifies the behavior of our architecture as well as its interactions with the environment. In figure 7 we have an excerpt of FIACRE which is the part of the code where $SAP$ verifies the availability of the targeted component before sending a negative response or calling $CHP$. $CHP$ verifies the access rights before approving to forward the message or calling a countermeasure ($TrigAct$).

```
from SAP

    if (not (availableComponent(id,mess.comm.target))) then
        mess := getRespMess(id,mess,false);
        /*@comp_UnavailableComponent*/
        to Sending
    end;
    /*@comp_AvailableComponent*/
    to CHP

from CHP

    if (canPass(id,mess.comm)) then
        canP := true;
        /*@comp_CanPass*/
        to Sending
    else
        /*@comp_CannotPass*/
        to TrigAct
    end
```

Figure 7. FIACRE Example

We formalize the security requirements using CDL [7] which is associated with the OBP tool. In figure 8 we show the $SAP\_C\_1$ property applied as an observer on a component named *comp*. The observer is triggered once *comp* receives a message. If the source of the message is friendly (in the same secure zone), the observer goes back to its initial state for future verifications when another message is received. If not, the observer checks whether the message was controlled or not, if it was controlled, the observer goes back to its initial state, if not, the property is rejected.

[1]OBP was developed in the reception team at Ensta Bretagne, it is accessible for free on http://www.obpcdl.org.

[2]Language defined in the framework of the TopCased project (http://www.topcased.org).

```
property prt_SAP_C1_comp1 is
{
    start           -- / / eve_comp1_Received_true      /   -> at_Received;

    at_Received     -- / pre_comp1_FriendlySource       / / -> start;
    at_Received     -- / not pre_comp1_FriendlySource   / / -> wasItControlled;

    wasItControlled -- / pre_com1_Controlled            / / -> start;
    wasItControlled -- / not pre_Comp1_Controlled       / / -> reject
}
```

Figure 8. CDL Example

Normal scenarios, as well as attack scenarios can be described in either CDL or FIACRE. Describing scenarios in CDL allows to take advantage of the work on the reduction of the complexity during the explorations of the models [6], which is a well-known problematic aspect of model checking.

## VII. Experimentation

As an illustration, we consider an architecture in which security mechanisms have been integrated (fig. 9). For example, this architecture corresponds to an abstract model of a vetronic vehicle architecture [18] or a SCADA architecture.

### A. Architecture Description

The architecture (fig. 9) is composed of 4 types of processes: $GCS$: access component, but also can forward messages without controlling nor modifying them. $NET$: communication component which controls and forwards messages. $PLC$: indirect access component, they apply access rights. $DEV$: Access components affected by requests sent to a $PLC$, $DEV$ cannot apply security measures.

A request is sent by an environment client to $PLC_i$ which verifies the access rights and sends the message to $DEV_i$. The response of $DEV_i$ is sent to $PLC_i$ which sends the final response to the environment. The principle is that, $DEV_i$ cannot implement security measures, so we forbid it to communicate with other components than $PLC_i$. To do so, $NET_2$ filters communications that are not between $DEV_i$ and $PLC_i$ (inbound or outbound).

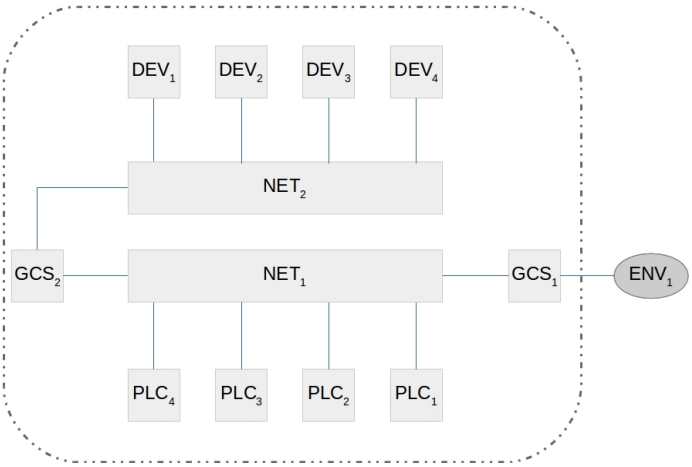


Figure 9. Secure Architecture

### B. Scenarios and Results

During our experiments and during properties verification, we submit our architecture to regular and attack scenarios of various forms. Some attacks are stopped by $NET$ components (does not pass message filtering). Other attacks are stopped by $PLC$ components (do not have sufficient access rights). The environment has four types of actors (clients): $CLT$: has authorized access to some $PLC$ resources. $SPM$: violates firewall rules. $ATT$: violates access rights. $PEN$: same as $ATT$, but, can use internal communications directly.

The resulting behavior, was as expected. Depending on the type of the client, the following happens: $CLT$: The requests went through $PLC$ and $DEV$, and the final response was positive ($ans = ACK$). $SPM$: $NET\_1$ responds with $ans = NAK$ $ATT$: $PLC_i$ responds with $ans = NAK$ $PEN$: No response, nor forward, due to signature violation, the component that received the message, ignored it.

### C. Verification

If we consider the property $prt\_SAP\_C\_2$ which is applicable to all components of the type $NET$ integrating a $SAP$. An instance of this property on $NET\_1$ in $CDL$:

**propertyLTL** $prt\_SAP\_C\_2\_NET\_1$ **is**
$\{ [\,] ($
$| \; pre\_NET\_1\_received \; \wedge \; not \; pre\_NET\_1\_available \; |$
$=> <>$
$| \; pre\_NET\_1\_sent \; \wedge \; NET\_1.mess.ans = NAK \; |) \}$

With the predicates: $pre\_NET\_1\_received$ is true if $NET\_1$ has received a message, and $pre\_NET\_1\_sent$ is true if $NET\_1$ has sent a message. $pre\_NET\_1\_available$ is true if the message target in $NET\_1$ is available, and $NET\_1.mess.ans$ is the answer ($ACK$ or $NAK$) in the message prepared in $NET\_1$.

We variate the actors composition to include multiple attacks ($SPM, ATT, PEN$) and multiple regular clients $CLT$. We notice that no actual property was violated. However, a special observer was violated detecting the behavior of $PEN$ when a message is installed directly behind a component of type $NET$. We also verified the different properties in a cyclic application, where clients and attacks were implemented using $FIACRE$ so they would continuously send requests.

### D. Complexity Analysis

To analyze the complexity of our approach, we generated specific scenarios with a number of clients $CLT$ and messages per client. In the following, we illustrate scenarios of the form $A\_i\_j$ with $i$ clients of type $CLT$ sending requests simultaneously. Each client sends $j$ requests, where each request waits for the response of the previously sent request from the same client. In other words, we have $i$ parallel clients sending $j$ sequential messages each. Each of these messages pass from $ENV_1$ to $GCS_1$ to $NET_1$ before arriving to the targeted $PLC$. The $PLC$ forwards the command after verifying it to the targeted $DEV$ through $NET_1$, $GCS_2$, and $NET_2$. The response is sent from $DEV$ to $PLC$ and back from $PLC$ to $ENV_1$. In these specific scenarios, all the messages are legit, so each message takes exactly the same amount of transitions from the request creation to the reception of the response. When multiple messages are transiting in parallel, we notice the combination between all the possible configurations resulting in a more important difference between the behaviors of the secure and insecure models.

Table I demonstrates the results of using multiple actors and multiple messages per actor. These results are separated between the insecure and the secure architecture model. Each contains the number of resulting configurations in the exploration of $OBP$, the number of transitions, and the depth. The number of configurations is the number of different possible states of the whole system. The number of transitions is the number of possible transitions for the system from one configuration to another. Finally, the depth, is the length of the sequence of transitions taken by the system from its original configuration (before the environment starts interacting with it) to its final configuration (after all requests has been sent and responses received). The depth is usually dynamic depending on the different possible routes that can be taken, but, in this example, all routes have the same length, which is intentionally done to normalize the possibilities for a better comparison between secure and insecure model.

Table I
EXPERIMENTATION RESULTS

| | Insecure Model | | | Secure Model | | |
|---|---|---|---|---|---|---|
| **Scenario** | **nb. confs** | **nb. trans** | **Depth** | **nb. confs** | **nb. trans** | **Depth** |
| 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 |

Figure 10 demonstrates the ratios of configurations, transitions, and depth, between the insecure and secure model. We notice that the ratio of the depth is a constant, meaning that even if the route is longer in the case of a secure model, the difference is stable. Both the configurations and transitions ratios change slightly when changing the number of messages. However, the change is more interesting when the number of parallel actors is changed.

Finally, these results are based on regular scenarios with legit clients, these clients always have their requests accepted and going all the way to $DEV$ and back. Using attack scenarios based on the other actors ($SPM, ATT, PEN$) results in fewer configurations, transitions, and depth, in the secure model than the insecure one. This is due to the fact that the requests of $ATT$ have a negative response directly from the $PLC$. $SPM$ have even less configurations, transitions, and depth, since their requests receive a direct negative response from $NET\_1$. $PEN$ go even lower with their requests being ignored by the receiving component.

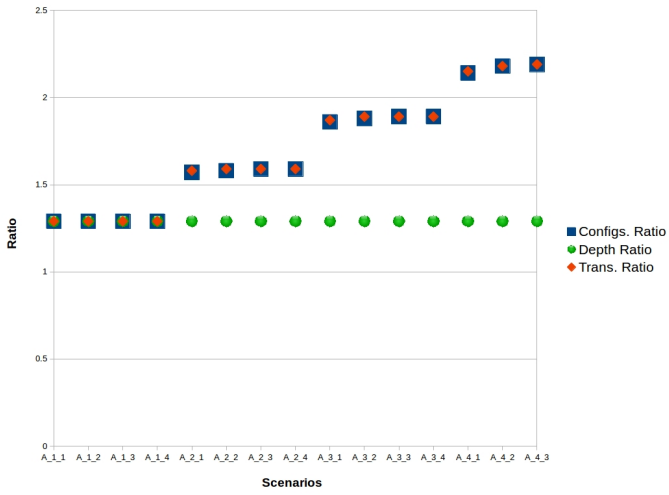


Figure 10. Ratios Between Secure and Insecure

## VIII. Conclusion

Our goal was to provide rules to generate the secure architecture, and tools to formally validate that the resulting architecture complies with a given security policy.

In this work, we studied how to generate a secure architecture based on an insecure one, a security policy, and a library of security patterns. The resulting model is verified using model checking to validate the requirements of the architecture, the security policy, and the patterns as well. We have placed ourselves in a framework of SCADA architectures, however, the reflections developed in this work are of a generic nature and could be adapted to other types of architectures.

During our work, we were confronted with this vast problematic where many parameters have to be taken into account to answer the security needs of the architecture. We have opted for implementation choices that may not be the best possible ones. Further work needs to be done to optimize these implementations and to identify the right criteria for comparison between the different strategies.

During the experiments, we restricted ourselves to the limited number of four patterns. But a similar approach could be undertaken for the integration of other patterns described in the literature. For a given security policy, the question is: Which method of integration of the patterns do we defend? What is the impact on complexity and robustness against attacks? What are the compromises to find?

The perspectives of this work concern several axes to be taken into account. They concern the taking into account of security policies which can be complex and defined in a dynamic way. These policies must be based on an indispensable formalization from which an architectural model generation process can be considered. A wider choice of security mechanisms will be needed as well as the automation of the process of integration and composition of more numerous and different patterns. Finally, in addition to simulated use cases, we are working on a use case involving $Vetronics$.

## References


[1] T. R. D. 5200.28-STD. *Trusted Computer System Evaluation Criteria*. Department of Defense, 1985.

[2] P. Alencar, D. Cowan, J. Dong, and C. Lucena. A pattern-based approach to structural design composition. In *Computer Software and Applications Conference, 1999. COMPSAC'99. Proceedings. The Twenty-Third Annual International*, pages 160–165. IEEE, 1999.

[3] P. S. Alencar, D. D. Cowan, and C. J. P. d. Lucena. A formal approach to architectural design patterns. In *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 576–594. Springer, 1996.

[4] M. Avalle, A. Pironti, and R. Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26(1):99–123, 2014.

[5] S. A. Boyer. *SCADA: supervisory control and data acquisition*. International Society of Automation, 2009.

[6] L. L. R. Ciprian Teodorov, Philippe Dhaussy. Environment-driven reachability for timed systems : Safety verification of an aircraft landing gear system. *Int. Software Tools for Technology Transfer (STTT)*, 2016.

[7] P. Dhaussy, F. Boniol, J.-C. Roger, and L. Leroux. Improving model checking with context modelling. *Advances in Software Engineering*, 2012.

[8] J. Dong, P. S. Alencar, and D. D. Cowan. Ensuring structure and behavior correctness in design composition. In *Engineering of Computer Based Systems, 2000.(ECBS 2000) Proceedings. Seventh IEEE International Conference and Workshopon the*, pages 279–287. IEEE, 2000.

[9] J. Dong, T. Peng, and Y. Zhao. Model checking security pattern compositions. In *Quality Software, 2007. QSIC'07. Seventh International Conference on*, pages 80–89. IEEE, 2007.

[10] E. B. Fernandez and M. M. Larrondo-Petrie. Designing secure scada systems using security patterns. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–8. IEEE, 2010.

[11] E. B. Fernandez and R. Pan. A pattern language for security models. In *proceedings of PLOP*, volume 1, 2001.

[12] I. N. Fovino, A. Coletta, A. Carcano, and M. Masera. Critical state-based filtering system for securing scada network protocols. *Industrial Electronics, IEEE Transactions on*, 59(10):3943–3950, 2012.

[13] I. N. Fovino, A. Coletta, and M. Masera. Taxonomy of security solutions for the scada sector. *Project ESCORTS Deliverable*, 2, 2010.

[14] M. Hafiz, P. Adamczyk, and R. E. Johnson. Growing a pattern language (for security). In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 139–158. ACM, 2012.

[15] V. M. Igure, S. A. Laughter, and R. D. Williams. Security issues in scada networks. *Computers & Security*, 25(7):498–506, 2006.

[16] R. L. Krutz. *Securing SCADA systems*. John Wiley & Sons, 2005.

[17] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.

[18] E. Stipidis. Vetronics system integration. pages 401 – 415, April 2006.

[19] T. Taibi and D. C. L. Ngo. Formal specification of design pattern combination using bpsl. *Information and Software Technology*, 45(3):157–170, 2003.

[20] Y. Wang. sscada: securing scada infrastructure communications. *International Journal of Communication Networks and Distributed Systems*, 6(1):59–78, 2010.

[21] R. Wassermann and B. H. Cheng. Security patterns. In *Michigan State University, PLoP Conf*. Citeseer, 2003.

[22] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. *Urbana*, 51:61801, 1997.

[23] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. *Urbana*, 51:61801, 1998.

[24] N. Yoshioka, H. Washizaki, and K. Maruyama. A survey on security patterns. *Progress in informatics*, 5(5):35–47, 2008.

[25] B. Zhu, A. Joseph, and S. Sastry. A taxonomy of cyber attacks on scada systems. In *Internet of things (iThings/CPSCom), 2011 international conference on and 4th international conference on cyber, physical and social computing*, pages 380–388. IEEE, 2011.

[26] B. Zhu and S. Sastry. Scada-specific intrusion detection/prevention systems: a survey and taxonomy. In *Proceedings of the 1st Workshop on Secure Control Systems (SCS)*, 2010.