Security policies for distributed systems*

Jean Quilbeuf, Georgeta Igna, Denis Bytschkow, and Harald Ruess

fortiss, Munich, Germany
{quilbeuf,igna,bytschkow,ruess}@fortiss.org

Abstract. Security properties can be expressed through security policies. A security policy specifies a maximal information flow allowed in a given system. In this paper, we focus on distributed systems communicating through message passing. Such a system implicitly defines a security policy that forbids direct information flow between processes that do not exchange messages. We show that the implicit security policy is enforced, provided that processes run in separation and the only communications allowed by the platform follow the message paths. We propose to further restrict the allowed information flow by adding filter functions that control which messages can be transmitted between processes. We prove that locally checking filter functions is a sufficient condition to ensure a global security policy. We present a case study for illustrating this approach.

1 Introduction

Today's application require a high computational power that is often achieved by using distributed systems. Such systems assemble processes provided by different parties. In order to achieve a global functionality, processes exchange messages. Some processes know sensitive data that should not be disclosed to other processes. Ensuring that sensitive data are not divulged to wrong processes is of a major importance when deploying distributed systems. For instance, in a voting system, there is a central unit that counts votes. However, the choice of one particular voter, which is sensitive data, should not be disclosed by the central unit to other voters. The latter requirement informally defines a security property.

A security property can be expressed through a security policy [4]. In general, a security policy decomposes a system into security domains, each of them having consistent level of information. For instance, one may decompose a system according to users or processes. A security policy a maximal bound on the information that can be transmitted between the security domains. Intuitively, the security policy specifies how information is allowed to flow between domains, e.g. whether a user may access the data of another user. After executing a given global sequence of actions, each security domain should not have gathered more information than what is allowed by the security policy.

* This work was carried out as part of the DMILS project www.d-mils.org. The project is partially funded by the European Commission under the 7th Framework Programme for Information and Communications Technologies.

We consider distributed systems consisting of processes communicating through asynchronous message passing. In this context, processes already provide a decomposition of the system into security domains. Therefore, a distributed system implicitly defines a security policy. This policy is constructed as follows: whenever two processes exchange messages, the policy allows information to flow from the sender of the message to the receiver. In the literature, there is a distinction between transitive and intransitive security policies. A transitive policy implies that if information may flow from process π_1 to process π_2 and then from process π_2 to process π_3, then information can also directly flow from process π_1 to process π_3. In general, the implicit security policy described above is not transitive, e.g. there might be transmission of a message from π_1 to π_2 and π_2 to π_3 even if π_1 cannot directly send a message to π_3. In such a situation, transmission of information from π_1 to π_3 is not impossible, but requires participation of π_2.

Rushby [11] and van der Meyden [9] assume that any action is possible from any global state. Assume that an action a transmits information from π_1 to π_2 and an action b transmits information from π_2 to π_3. Assume that the action b transmits the information about π_1 that π_2 has gathered. Action b can be executed before or after π_1 executed the action a, which yields different states in π_3, depending on the state of π_2. In order to capture this, Rushby and van der Meyden define the information available to a security domain in a recursive manner. For instance, the information available in π_3 after executing b contains the information available in π_2 before executing b.

We consider an asynchronous setting, where the actions are the emission or the reception of a message. Therefore, all actions are not always possible (i.e. it is not possible to receive a message that was not sent). A system defines a set of possible execution traces, which is also assumed by Mantel [8] and Balliu [2]. Furthermore, we assume that messages contain all the information, that is the information detained by a process is the sequence of messages that it received and sent so far. In this setting, sending the message b from the domain π_2 is possible only when π_2 has received the message a. When the domain π_3 receives b, it may infer, knowing the set of possible executions, that a has been transmitted. Even in that case, we consider the system secure, as the information was relayed by π_2. The system would be insecure if π_3 could directly observe the emission of a.

Enforcing the kind of security policy described so far can be achieved by low-level artefacts, such as separation kernels [10] and time triggered networks [6]. Separation kernels use time and space partitioning to ensure that two process running on the machine do not interfere. Similarly, time triggered networks can provide time partitioning and ensure isolation of the channels. A platform combining these two approaches is being built in the D-MILS project. Configuring the platform so that only processes that are supposed to exchange messages can actually communicate ensures that the security policy is respected.

In the example of the voting system, each voter receives the result of the vote from the central unit once the election is done. Since each voter sent his vote before the results are published, the implicit security policy allows the result message to contain the detail of the votes. The implicit security policy is not strong enough to ensure that no voter may know the vote of another voter. Chong and van der Meyden [**?**] introduced filter functions to strengthen a given security policy. A filter function restrict the information between two security domains, depending to the history of the actions executed. For instance, the filter function between the central unit and a voter might allow transmission of information only if (1) all voters have voted and (2) the data sent back are only the election results. The version by van der Meyden allows the filter function to replace an action by another one. In [12], Zhang considers filter functions that return a boolean, allowing or not the information to flow.

We also consider filter functions that return a boolean. We require that a filter function depend only on the local history of the domain that can issue the information. For instance, for the voting system example, the filter function depends only on the messages received and sent by the central unit. A filter function restricts the outputs of one particular process. Checking that the behavior of this particular process respects the filter functions, that is does not send messages it is not allowed to send, can be done separately. We show that it is sufficient to check that every filter function is respected to prove that the security policy is met. Components whose output is not restricted by a filter function do not need to be checked. In the voting system example, we do not need to check the behavior of each voter, it is sufficient to check the central unit.

Our approach is illustrated on a smart micro grid example. The smart micro grid negotiates with a set of prosumers, that both produce and consume energy, in order to assure that the global production or consumption of the whole system stay within predefined bounds. The security property to ensure is that no prosumer should be able to deduce the consumption or production of an other prosumer. This property is encoded by a security policy using a filter function. We use model checking techniques to prove that the smart micro grid respects the filter function.

This paper is organized as follows. In Section 2, we define our model of distributed systems, that we call *distributed machine* and the notion of security policy. In Section 3, we show that a distributed system complies with its implicit security policy and that local check of filter function is sufficient to ensure that the global system meets a filtered security policy. Our case study is detailed in Section 4. Finally, we present some related works in Section 5 and conclude in Section 6.

2 Information Flow

The information flow of a system is the information exchanged between different parts of a system. A security policy defines a maximal information flow. A system meets a security policy if no part of the system can obtain more information than allowed by the maximal information flow.

Following the work by Chong and van der Meyden [3], we model a system using the notion of machine, and the maximum information flow as a security policy that are formally defined in the next sections. We formally state the fact that a machine complies with a security policy.

2.1 Distributed Machine

In this paper, we focus on distributed systems, which are made of independent processes exchanging messages. Our definition of distributed systems complies with the machine model introduced by Rushby [11] and van der Meyden [9]. We start by formally defining processes and then explain how to compose them.

Definition 1. *A process is a tuple* $\pi = (S, s^0, A^{in}, A^{out}, \texttt{step})$*, where*

- S *is a set of local states, with* s^0 *as the initial state,*
- $A = A^{in} \cup A^{out}$ *is the set of actions composed of* receive actions A^{in} *and* send actions A^{out}*, and,*
- $\texttt{step} : S \times A \to S$ *is a partial transition function. We write* $\texttt{step}(s, a) =$ *undef when the action* a *cannot be executed from state* s*.*

We distinguish between two types of actions in a process. A send action, denoted $!_m$ models the emission of the message m. A receive action, denoted $?^{\pi}_{m}$ models the reception of the message m in the process π. The set of states and the set of actions may be infinite.

Not all actions are possible from every state. We denote by $LocalExec(\pi)$ the sequence of actions that are valid executions of π. For a sequence $\alpha \in LocalExec(\pi)$, we denote by $s^0.\alpha$ the state reached after executing the sequence α.

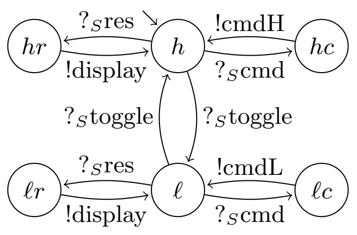


Fig. 1. Graphical representation of a process.

Example 1. In Figure 1, we represent a process, that operates as a switch for sending commands either to a high or a low level network. This example shows a simplified version of the switch component from the Starlight Interactive Link [1]. In [3], the Starlight system is taken as an example to illustrate filter functions. On the figure we prefix receive actions by ? and send actions by !. This process may receive 3 types of messages, namely res, cmd and toggle. The reception of a toggle message (from the user) switches the security mode between high (state h) and low (state l). Whenever a cmd (command) message is received (from the user), it is forwarded to a network for execution. If the process is in high security mode, the command is sent to the high security network through a cmdH message, otherwise it is sent to the low security network through a cmdL message. Whenever a res (result) message is received, it is forwarded to the user through a display message.

Formally, the set of states is $\{h, hr, hc, \ell, \ell r, \ell c\}$. The set of input actions is $\{\text{cmd}, \text{toggle}, \text{res}\}$ and the set of output actions is $\{\text{cmdL}, \text{cmdH}, \text{display}\}$. The `step` function is defined as shown on the figure, for instance $\mathtt{step}(h, \text{toggle}) = \ell$ and $\mathtt{step}(hr, \text{toggle}) = \mathit{undef}$.

In this paper, we consider *distributed machines*, which are defined as the composition of a set of processes. The processes communicate through asynchronous message passing. An action of the distributed machine is either the emission or the reception of a message. As the synchronous reception of the message might not be possible in every state of a process, we assume that each process is equipped with a buffer for storing incoming messages. The consumption of a message in the buffer corresponds to a receive action. Identifying the sender of an action allows mapping this action to a security domain, which is needed to reason about information flow. Therefore, we require that for each message defined in the set of processes, there is a unique sender.

Definition 2 (Process composability). *Given a set of processes $\{\pi_1, \ldots, \pi_n\}$, where for each i, $\pi_i = (S_i, s_i^0, A_i^{in}, A_i^{out}, \mathtt{step}_i)$, we say that they are* composable *iff for each message m, there exists a unique process π_i such that $!_m \in A_i^{out}$ and at least one process π_j such that $?_a^{\pi_j} \in A_j^{in}$. We call process π_i the* sender *of m and π_j a* receiver *of m.*

The behavior of a distributed machine is as follows. Sending a message $!_m$ is done by executing the transition labeled by $!_m$ in the sender and adding m in the buffer of each receiver of m. Receiving a message $?_m^{\pi_i}$ in π_i is done by removing an occurrence of m from the buffer of π_i and executing the corresponding transition locally. The buffer is represented by a sequence of input actions. If several receive actions are possible, only the one corresponding to the message occurring first in the buffer can be executed.

A machine as defined in [9] is a transition system, extended with a decomposition into security domains. Each of these security domains delimits a subpart of the system that is granted a given level of information. The definition of a machine contains an observation function stating what each domain can observe from the global state. In a distributed machine, each process corresponds to a security domain, which can only observe its local state and the buffer of incoming messages.

Definition 3. *Given a set of processes $\{\pi_1, \ldots, \pi_n\}$ that are composable, a* distributed machine *is a tuple $\mathcal{M} = (S, s^0, A, \mathtt{step}, D, \mathtt{dom}, \mathtt{obs})$, such that*

- *$S = (S_1 \times A_1^{in*}) \times \ldots \times (S_n \times A_n^{in*})$, and $s^0 = ((s_1^0, \epsilon), \ldots, (s_n^0, \epsilon))$. We denote by $(q_1, \ldots, q_n)$ a global state of the system, where for each i, $q_i = (s_i, \beta_i)$*

indicates the current state s_i of the process π_i and the contents β_i of the input buffer.

– $A = \bigcup_{i=1}^{n} \left(A_i^{in} \cup A_i^{out}\right)$ *The actions of the distributed are emissions and receptions of messages.*

– `step` *contains two types of transitions, that correspond to sending and receiving messages:*

- *if for a process π_i there exists $!_m \in A_i^{out}$ such that $\mathtt{step}_i(s_i, !_m) \neq$ undef then $\mathtt{step}((q_1, \ldots, q_n), !_m) = (q'_1, \ldots, q'_n)$ where*

$$q'_j = (s'_j, \beta'_j) = \begin{cases} (\mathtt{step}_j(s_j, !_m), \beta_j) & \text{if } i = j \wedge ?_m^{\pi_j} \notin A_j^{in} \\ (s_j, \beta_j.m) & \text{if } ?_m^{\pi_j} \in A_j^{in} \wedge i \neq j \\ (\mathtt{step}_j(s_j, !_m), \beta_j.m) & \text{if } i = j \wedge ?_m^{\pi_j} \in A_j^{in} \\ (s_j, \beta_j) & \text{otherwise} \end{cases}$$

- *if for a process π_i, there exists $?_m^{\pi_i} \in A_i^{in}$ such that*

$$\mathtt{step}_i(s_i, ?_m^{\pi_i}) \neq \mathit{undef} \wedge \beta_i = \alpha_1.m.\alpha_2 \wedge \forall b \in \alpha_1 \ \mathtt{step}_i(s_i, ?_b^{\pi_i}) = \mathit{undef}$$

then

$$\mathtt{step}((q_1, \ldots, (s_i, \beta_i), \ldots, q_n), ?_m^{\pi_i}) = (q_1, \ldots, (\mathtt{step}_i(s_i, ?_m^{\pi_i}), \alpha_1.\alpha_2), \ldots, q_n)$$

– $D = \{\pi_1, \ldots, \pi_n\}$,

– $\mathtt{dom} : A \to D$ *such that* $\mathtt{dom}(!_m) = \mathtt{dom}(?_m^{\pi_j}) = \pi_i$ *where π_i is the only process such that $!_m \in A_i^{out}$,*

– $\mathtt{obs} : D \times S \to \bigcup_{i=1}^{n} (S_i \times A_i^{in})$, *defined by* $\mathtt{obs}(\pi, (q_1, \ldots, q_i, \ldots, q_n)) = q_i$.

As for processes, a machine cannot execute any action from any state. We denote by $Exec(\mathcal{M})$ the valid execution of a machine. For a sequence $\alpha \in Exec(\mathcal{M})$, we denote by $s^0.\alpha$ the global state reached after executing α.

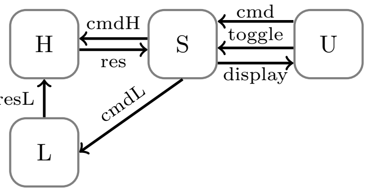


Fig. 2. A machine obtained by composing the processes $\{H, L, S, U\}$. Each arrow corresponds to a message that can be exchanged between two processes.

Example 2. Figure 2 represents a global view of the Starlight Interactive Link. There are four processes: a high-security network H, a low-security network L, the switch S detailed in Figure 1 and a user U. The user chooses between high-security and low-security mode by sending a toggle message. The user inputs commands (cmd) to the switch, which forwards them to the high- or low-security network, depending on the current mode. Upon reception of a command, the H or L process executes it and outputs the result (resL and res messages respectively). The result of a command executed in the low-security network is transmitted back to the switch through the high-security network. Upon reception of a result, the switch forwards it to the user through a display message.

We do not detail the behavior of each component. The beginning of a valid execution of the system is $!_{\text{cmd}}\ !_{\text{toggle}}\ ?^{S}_{\text{cmd}}\ !_{\text{cmdH}}\ ?^{S}_{\text{toggle}}$. Actions $?^{S}_{\text{cmd}}$, $?^{S}_{\text{toggle}}$ and $!_{\text{cmdH}}$ correspond to the transitions of process S as depicted in Figure 1.

Each process is a security domain. Each action that is sending or receiving a message is associated to the domain corresponding to the sender process. For instance, $\texttt{dom}(!_{\text{cmdL}}) = \texttt{dom}(?^{L}_{\text{cmdL}}) = L$. Each process can observe its local state and its input buffer. For instance, in the global state s reached after executing $!_{\text{cmd}}!_{\text{toggle}}$, we have $\texttt{obs}(S, s) = (h, \text{cmd toggle})$.

In this example, the security property to ensure is "The low security network has no information about the commands sent to the high security network".

2.2 Security policy

A security policy is represented as a directed graph, whose vertices are the security domains and edges may be labeled by filter functions. Intuitively, an edge between two security domains allows information to flow according to the direction of the edge. An example of security policy is depicted in Figure 3 which presents how information flows in the machine depicted in Figure 2.

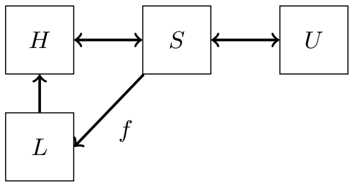

$$\begin{tikzcd}
H \arrow[r, leftrightarrow] & S \arrow[r, leftrightarrow] \arrow[dl, "f"] & U \\
L \arrow[u] & &
\end{tikzcd}$$

Fig. 3. Security policy with the domains of the machine from Figure 2.

In our model, transmission of information is expressed through actions, that modify the state of different processes. The presence of an edge between two security domains indicates that any action executed at the source of the edge may transmit information to the destination of the edge. According to Figure 3, an action executed in process S might affect both the state of H, L and U. An edge labeled by a filter function may allow or not an action at the source to affect the state of the destination, depending on the sequence of actions executed so far. In the figure, the edge between S and L is labeled by a filter function f. The intuitive role of f is to ensure that L receives only commands sent in low security mode by the user.

Definition 4. *Given a set of actions A, a filter function $f : A^* \rightarrow \{True, False\}$ returns a boolean indicating whether the last action of the sequence can be transmitted.*

We denote by $\mathcal{F}(A)$ the set of filter functions defined on the set of actions A.

In our example, the function f is true only when both:

– the last action is $!_{\text{cmdL}}$, and
– the number of toggle messages received is odd, indicating that the user is inputting commands in low security mode.

Formally, we write $f(\alpha a) = (a = !_{\text{cmdL}}) \wedge (|\alpha|_{?_{toggle}} \mod 2 = 1)$, where $|\alpha|_a$ is the number of occurrences of a in the sequence α.

Definition 5. *A* security policy $\mathcal{A}$ *defined over the set of actions A is a pair $(D, \rightsquigarrow)$ where*

– *D is a set of security domains and*
– *$\rightsquigarrow \subseteq D \times \mathcal{F}(A) \cup \{\top\} \times D$ is a set of edges, labeled by filter functions in $\mathcal{F}$ or $\top$. We require that:*
  - *$\forall \pi_i \in D.(\pi_i, \top, \pi_i) \in \rightsquigarrow$.*
  - *if $(\pi_i, f, \pi_j) \in \rightsquigarrow$, then f is in $\mathcal{F}(A_i^{in} \cup A_i^{out})$.*

The first restriction on the definition of $\rightsquigarrow$ states that each security domain can always observe actions that are associated to itself. The second restriction states that a filter function applied to the flow between a process π_i and π_j should depend only on the sequence of messages received and sent by π_i. In particular, all actions that are not actions of π_i are not taken into account.

We write $\pi_i \overset{f}{\rightsquigarrow} \pi_j$ if $(\pi_i, f, \pi_j) \in \rightsquigarrow$. An edge labeled by $\top$ imposes no restriction on the corresponding information flow. In that case, we denote $\pi_i \rightsquigarrow \pi_j$. A security policy is *transitive* if it contains only edges labeled by $\top$ and the relation $\rightsquigarrow$ is transitive. Otherwise, it is intransitive. We focus on intransitive policies, such as the one depicted in Figure 3.

There are several constructions for representing the maximal information allowed by a security policy [3]. Contrarily to the model prescribed by [3], we do not assume that any action is possible from any state. Consequently, each process knows the set of possible global executions and may infer some informations about the global state, based on its observation. In our example, when the low security network receives a command to execute, it knows that this command was sent before by the user to the switch. Such information may be computed by using knowledge with perfect recall as in [2].

The information available to a security domain after executing a given sequence of actions is obtained by purging actions not visible by the security domain. An information is not visible by a security domain if either there is no incoming arrow from the domain of the action or if the incoming arrow is labeled by a filter function that evaluates to false. Formally, we recursively define the purge function for a security domain π as $\texttt{purge}_\pi(\epsilon) = \epsilon$ and

$$\texttt{purge}_\pi(\alpha a) = \begin{cases} \texttt{purge}_\pi(\alpha) & \text{if } \texttt{dom}(a) \not\rightsquigarrow \pi \vee (\texttt{dom}(a) \overset{f}{\rightsquigarrow} \pi \wedge \neg f(\alpha a)) \\ \texttt{purge}_\pi(\alpha) a & \text{otherwise} \end{cases}$$

The purged execution sequence represents the maximal information that a process is allowed to have at a given execution point. A machine meets a security policy if for each security domain, the observation after executing a sequence α depends only on the purged sequence.

Definition 6. *A machine* $\mathcal{M} = (S, s^0, A, \texttt{step}, D, \texttt{dom}, \texttt{obs})$ *complies with the security policy* $\mathcal{A} = (D, \rightsquigarrow)$ *if:*

$$\forall \pi_i \in D, \forall \alpha, \beta \in \mathit{Exec}(\mathcal{M}), \texttt{purge}_{\pi_i}(\alpha) = \texttt{purge}_{\pi_i}(\beta) \implies \texttt{obs}_{\pi_i}(s^0.\alpha) = \texttt{obs}_{\pi_i}(s^0.\beta)$$

Finally, we remark that a distributed machine implicitly defines a security policy without filter functions. The security domains are already defined by assigning a domain to each process. If a message can be sent from a process π_1 to a process π_2, there is an arrow from the security domain π_1 to the security domain π_2.

Definition 7. *A distributed machine obtained by composing the processes* $\{\pi_1, \ldots, \pi_n\}$ *defines the* implicit security policy $(D, \rightsquigarrow)$ *where:*

- $D = \{\pi_1, \ldots, \pi_n\}$ *each process is a security domain,*
- $a \in A_i^{out} \cap A_j^{in} \implies (\pi_i, \top, \pi_j) \in \rightsquigarrow$. *If a message can be sent from* π_i *to* π_j, *there is an arrow between the corresponding security domains.*

3 Proving Security Policies for Distributed Machines

We use the classical method based on unwinding relations [5] to prove that a machine complies with a given security policy. An unwinding relation assigns to each security domain π an equivalence relation on the states of the machine that we denote by $\sim_\pi$.

We first consider the case where all edges of the security policy are labeled by $\top$, that is without filter functions. In that case, the Theorem 1 by Rushby [11] states that a machine $\mathcal{M}$ complies with a security policy if there exists a family of unwinding relations $\{\sim_\pi\}_{\pi \in D}$ such that, for any two states $s, t \in S$, any process $\pi \in D$ and any action $a \in A$, we have:

- *Output consistency.* If $s \sim_\pi t$, then $\texttt{obs}(\pi, s) = \texttt{obs}(\pi, t)$.
- *Step consistency.* If $s \sim_\pi t$ then $\forall a \in A, \texttt{step}(s, a) \sim_\pi \texttt{step}(t, a)$.
- *Local respect.* If $\texttt{dom}(a) \not\rightsquigarrow \pi$, then $s \sim_\pi \texttt{step}(s, a)$.

Note that we actually use the formulation by van der Meyden [9]. Using unwinding relations, we prove that a distributed machine complies with its implicit security policy as in Definition 7.

Proposition 1. *A distributed machine complies with its implicit security policy.*

The proof relies on the fact that the current state of a process depends only on the messages it receives and sends. The implicit security policy allows each process to observe these actions. Therefore the distributed machine complies with the implicit security policy.

Proof. We consider the unwinding relations obtained as follows: Given two states $q = (q_1, \ldots, q_n)$ and $r = (r_1, \ldots, r_n)$, we define $q \sim_{\pi_i} r$ iff $q_i = r_i$. This is clearly an equivalence relation. Recall that for each i, $q_i = (s_i, \beta_i)$ where s_i is the state of the process and β_i the input buffer. We now prove the three properties needed to establish security:

- *Output consistency* $q \sim_{\pi} r \implies q_i = r_i \implies \mathtt{obs}(\pi_i, q) = \mathtt{obs}(\pi_i, r)$.
- *Step consistency* Let $a \in A$ be an action, $q, r \in S$ be two global states and $q' = \mathtt{step}(q, a)$, $r' = \mathtt{step}(r, a)$ the states reached when executing a. We assume that $q \sim_{\pi} r$, that is $q_i = r_i$. Recall that a is either the emission of a message $!_m$ or the reception of a message $?^{\pi_i}_m$.
  If π_i is not sender nor receiver of the message m, by definition of $\mathtt{step}$, we have $q'_i = q_i$ and $r'_i = r_i$. Since $q_i = r_i$, we have $q' \sim_{\pi_i} r'$.
  If $a =!_m$ is the emission of the message m:
  - if π_i is the sender of the message, by definition of $\mathtt{step}$, $q'_i = (\mathtt{step}_i(s_i, m), \beta_i)$ and $r'_i = (\mathtt{step}_i(t_i, m), \gamma_i)$. Since $(s_i, \beta_i) = q_i = r_i = (t_i, \gamma_i)$, we have $q'_i = r'_i$ that is $q' \sim_{\pi_i} r'$.
  - if π_i is a receiver of the message, by definition of $\mathtt{step}$, $q'_i = (s_i, \beta_i m)$ and $r'_i = (t_i, \gamma_i m)$. Since $(s_i, \beta_i) = q_i = r_i = (t_i, \gamma_i)$, we have $q'_i = r'_i$ that is $q' \sim_{\pi_i} r'$.

  If $a =?^{\pi_j}_m$ is the reception of the message m:
  - if π_i is the sender of the message, by definition of $\mathtt{step}$, $q'_i = q_i$ and $r'_i = r_i$, that is $q' \sim_{\pi_i} r'$.
  - if π_i is the receiver of the message, by definition of $\mathtt{step}$, $q'_i = (\mathtt{step}_i(s_i), \alpha_1\alpha_2)$ and $r'_i = (\mathtt{step}_i(t_i), \alpha'_1\alpha'_2)$, where $(s_i, \alpha_1 m \alpha_2) = q_i = r_i = (t_i, \alpha'_1 m \alpha'_2)$. Since m does not appear in α_1 or α'_1, we have $q'_i = r'_i$ that is $q' \sim_{\pi_i} r'$.
- *Local respect* Let $a \in A$ be an action, $q \in S$ be a state, and $q' = \mathtt{step}(q, a)$. By definition of $\rightsquigarrow$, $\mathtt{dom}(a) \not\rightsquigarrow \pi_i$ only if $\mathtt{dom}(a)$ does not send any message to π. By definition of $\mathtt{step}$, if a is the emission or the reception of a message not involving π_i, then $q'_i = q_i$. Thus $q \sim_{\pi_i} \mathtt{step}(q, a)$. □

We extend the unwinding theorem to security policies with filter functions. We define an additional property, that depends only on the local state of the component. This property ensures that the transmission of a message a process π_i to a process π_j cannot take place whenever the filter function on the edge between security domains π_i and π_j evaluates to false. The transmission of the message cannot take place if either π_j is not a receiver of the message or the send action is not possible in π_i.

– *Local Filter Respect*: $\forall(\pi_i, f, \pi_j) \in\rightsquigarrow \forall\alpha \in \mathit{LocalExec}(\pi_i)$
  $\neg f(\alpha a) \implies a \notin A_j^{in} \vee \mathtt{step}(s_i^0.\alpha, a) = \mathit{undef}$

Theorem 1 states that a machine complies with a security policy if the

Theorem 1. *Let $\mathcal{M}$ be a machine, $\mathcal{A}$ be a security policy, and for each domain π $\sim_\pi$ be an equivalence relation. If the relations $\{\sim_\pi\}_{\pi\in D}$ verify output consistency, step consistency, local respect and local filter respect properties, then the machine $\mathcal{M}$ complies with the architecture $\mathcal{A}$.*

Proof. Let $\alpha, \beta \in \mathit{Exec}(\mathcal{M})$ be two executions of $\mathcal{M}$. We prove by induction on $|\alpha| + |\beta|$ that $\mathtt{purge}_\pi(\alpha) = \mathtt{purge}_\pi(\beta) \implies s^0.\alpha \sim_\pi s^0.\beta$.
The base case is $\beta = \alpha = \epsilon$, and we have $s^0 \sim_\pi s^0$.
Let us write $\alpha = \alpha' a$. We distinguish between the two following cases:

– $\mathtt{dom}(a) \not\rightsquigarrow \pi$ or $\mathtt{dom}(a) \overset{f}{\rightsquigarrow} \pi$ and $f(\alpha)$ is false. In that case, $\mathtt{purge}_\pi(\alpha a) = \mathtt{purge}_\pi(\alpha) = \mathtt{purge}_\pi(\beta)$. By applying the induction hypothesis on α' and β, we obtain $s^0.\alpha' \sim_\pi s^0.\beta$. Since $\alpha' a$ is a valid execution sequence, the local respect ensures that a is not an action that can be received by π. Thus $s^0.\alpha' \sim_\pi \mathtt{step}(s^0.\alpha', a)$ and we conclude $s^0.\alpha \sim_\pi s^0.\beta$.
– $\mathtt{dom}(a) \rightsquigarrow \pi$ or $\mathtt{dom}(a) \overset{f}{\rightsquigarrow} \pi$ and $f(\alpha)$ is true. In that case, we can assume that β also ends with a. Otherwise, by swapping α and β, one falls back in the previous case. We write $\beta = \beta' a$. By definition of the purge function we have $\mathtt{purge}_\pi(\alpha) = \mathtt{purge}_\pi(\alpha')a$ and similarly $\mathtt{purge}_\pi(\beta) = \mathtt{purge}_\pi(\beta')a$ and therefore $\mathtt{purge}_\pi(\alpha') = \mathtt{purge}_\pi(\beta')$. The induction hypothesis applied to α' and β' gives us $s^0.\alpha' \sim_\pi s^0.\beta'$. The step consistency allows us to conclude $s^0.\alpha \sim_\pi s^0.\beta$.

We proved that $\mathtt{purge}_\pi(\alpha) = \mathtt{purge}_\pi(\beta) \implies s^0.\alpha \sim_\pi s^0.\beta$. By using the output consistency, we have $s^0.\alpha \sim_\pi s^0.\beta \implies \mathtt{obs}_\pi(s^0.\alpha) = \mathtt{obs}_\pi(s^0.\beta)$, which concludes the proof. □

The above proof is valid for any machine that admits unwinding relations verifying the above properties. We already exhibited unwinding relations for the particular case of distributed systems, where the security policy has no filter functions and matches the communications between processes. In order to prove security for a security policy with the same edges but possibly labeled by filter functions, one only need to prove that local filter respect holds.

4 Case Study

The case study discussed in this paper is a simplified version of a prosumer system that is implemented in our research lab [7]. The system contains a smart micro grid (SMG) that coordinates a finite set of prosumers $Pr_1, \ldots, Pr_n$. Each prosumer can produce energy and consume energy from the local production or from the grid. Moreover, each prosumer has the possibility to store energy in batteries. Therefore, a prosumer may sell energy or buy energy from the grid.

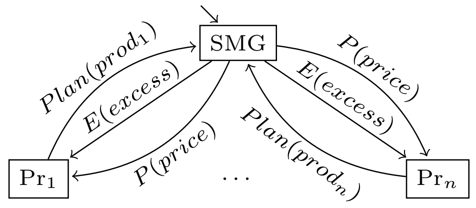


Fig. 4. Representation of the system machine

Figure 4 shows the machine of our case study. The SMG generates a price for energy and transfers it to the prosumers through action $P(price)$. The price is the same for both buying and selling energy to the grid. Based on the price received and an estimation of the local production and consumption, each prosumer computes a production plan (variable *prod* in the figure). This plan specifies the amount of energy the prosumer buys or sells to the grid. If this value is positive, then the prosumer produces energy that allows him to sell a part of it to the grid, otherwise, it buys energy from the grid. This value is sent through the *Plan* action to the SMG. When all prosumers have sent their production plans, the SMG computes the global production of the grid. The global production may be negative, in which case the prosumers consume more energy than what they produce.

The stability of the grid is assured by bounding the global production to available line capacity that is specified by an upper and and lower bounds i.e. U_B, and L_B respectively. Variable *excess* returns the amount by which the global production exceeds the bounds of the line capacity. If the global production is within the bounds the grid will be stable and the *excess* variable has the value zero, otherwise it returns the amount by which the global production exceeds the upper bounds U_B or lower bound L_B. If action $E(excess)$ transfers a nonzero value, all prosumers have to adjust their plans. After that, new plans are sent back to the SMG, which updates the excess and sends it to the prosumers. It may take more rounds to have the plans accepted by the SMG. Once the plans are validated, we assume that they become active for a finite period of time and after that a new price is generated and new plans are computed.

Figure 5 gives the security policy of our prosumer system. Since SMG is the only resource that has to be trusted, we specify on the edges from the SMG to prosumers a filter function f_{excess}. In order to formally describe the filter function, we define two additional functions. Given a finite sequence of actions

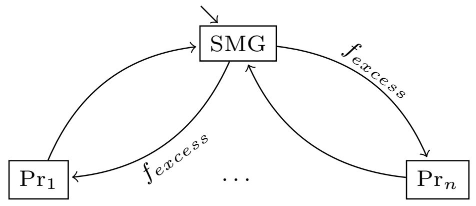


Fig. 5. Security policy of the system

$\alpha = a_1, \dots, a_m$, the suffix of size n of α, where n≤m, is:

$$\mathrm{suffix}(\alpha, n) = a_{m-n+1}, \dots, a_m.$$

Further, given a finite sequence of actions α and an action a, function $|\alpha|_a$ returns the number of occurrences of action a in α.

Then, f_{excess} is defined as follows:

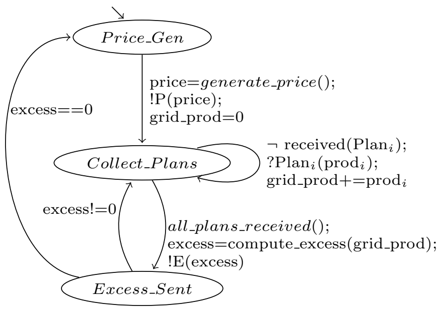
$$\begin{aligned} f_{excess}(\alpha a) = \\ & \Big(a = P(price) \wedge (\alpha = \epsilon \vee \mathrm{suffix}(\alpha, 1) = E(0))\Big) \vee & (1) \\ & \Big(a = E(excess) \wedge \bigwedge_{i=1}^{n} |\mathrm{suffix}(\alpha, i)|_{Plan(prod_i)} = 1 \\ & \wedge excess = compute\_excess(\sum_{i=1}^{n} prod_i)\Big) & (2) \end{aligned}$$

This function assures that:

(1) whenever action P is sent, it is either the initial action that the SMG does, or it comes after the emission of the message indicating that the production plans do not exceed the line capacity bounds, or
(2) whenever action E is sent, each prosumer has sent its production plan exactly once in the last n actions of the sequence α and the filter function sends the right excess value. This value is computed by function *compute_excess* defined as follows:

$$compute\_excess(\sum_{i=1}^{n} prod_i) = \begin{cases} 0, & \text{if } L_B \leq \sum_{i=1}^{n} prod_i \leq U_B \\ \sum_{i=1}^{n} prod_i - U_B, & \text{if } U_B < \sum_{i=1}^{n} prod_i \\ L_B - \sum_{i=1}^{n} prod_i, & \text{otherwise.} \end{cases}$$

The first case in the above definition holds when the sum of the plans sent does not exceed the L_B and U_B line capacity bounds. The second case arises when prosumers produce too much energy, whereas the last case occurs when the global consumption exceeds the line capacity lower bound.



(a) SMG

Fig. 6. Implementation of the SMG

In order to model the SMG and to check that it satisfies the filter function, we have built a Uppaal model. For readability we present in Figure 6 the main element of the corresponding model. At the moment we have no timing constraints in our model, but we can easily introduce them by adding deadlines for the time when the SMG waits for the prosumer plans.

The automaton in Figure 6 contains three locations: *Price_Gen*, which is the initial location and has a transition on which the energy price is generated and sent to prosumers, *Collect_Plans*, which is active for the period when plans are sent. Channel $Plan_i$ is binary, meaning that each prosumer sends its production plan on a unique channel. Guard ¬ received(Plan_i) guarantees that the SMG takes into account the first production plan each prosumer sends. When a prosumer tries to send a new plan before receiving an E action, the SMG simply ignores this action. Local variable grid_prod sums up the plans received. When the SMG has received a plan from each prosumer (i.e. function *all_plans_received* returns true), the transition to location *Excess_Sent* is taken. On this transition, the excess is computed using function *compute_excess* and sent to prosumers. When the excess is not zero, the transition between *Excess_Sent* and *Price_Gen* is taken which makes the SMG ready to receive adjusted production plans. Finally, in case the excess is zero, the transition to *Price_Gen* is taken. In order to check the filter function, we have written the following properties:

$$A[](SMG.EXCESS\_SENT) \text{ imply excess} == \\ compute\_excess(\sum_{i=1}^{n} prod_i, SMG.U_B, SMG.L_B), \tag{3}$$

and,

$$A[](!SMG.EXCESS\_SENT) \text{ imply excess==0}, \qquad (4)$$

Property (3) requires that whenever the SMG is not in location $EXCESS\_SENT$, the excess variable should have the default value i.e. zero. Property (4) checks that the excess is correctly computed when the SMG is in location $EXCESS\_SENT$. In order to express part (1) of the filter function, we should write the following property:

$$A[](SMG.price \neq 0) \text{ imply excess==0},$$

but this is ensured by Property (4), assuming that the default values for both price and excess is zero.

5 Related Works

Mantel [8] and Balliu consider systems defined by a given set of execution traces. In these systems, a given action is not always possible. Mantel considers different definition of security, using alway the same policy security. The considered policy contains only two security domains, H and L allows only information to flow from L to H.

Balliu [2] considers distributed systems where the confidentiality of channels between processes has to be ensured. The condition for security is based on knowledge with perfect recall: low security components should not be able to infer the content of a high security channel based on the history of observation on the low security channels. However, this requires to know exactly the set of traces of the system, and thus the implementation of each process. In the Starlight example, a process requesting the same command once in high security mode then once in low security mode would allow the low security network to know exactly the sequence of command executed in the high security network.

6 Conclusion

7 Acknowledgments

We thank the D-MILS project partners and fortiss *Living Lab* research group for the fruitful discussions had regarding the topic of this paper.

References

1. Anderson, M.S., North, C.J., Griffin, J.E., Milner, R.B., Yesberg, J.D., Yiu, K.K.H.: Starlight: Interactive link. In: ACSAC. pp. 55–63. IEEE Computer Society (1996)

2. Balliu, M.: A logic for information flow analysis of distributed programs. In: Riis Nielson, H., Gollmann, D. (eds.) Secure IT Systems, Lecture Notes in Computer Science, vol. 8208, pp. 84–99. Springer Berlin Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-642-41488-6_6`
3. Chong, S., van der Meyden, R.: Using architecture to reason about information security. In: Layered Assurance Workshop (2012)
4. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. pp. 11–20 (1982)
5. Goguen, J.A., Meseguer, J.: Unwinding and Inference Control. In: Proceedings of the 1984 IEEE Symposium on Security and Privacy. pp. 75–86. IEEE Computer Society (1984), `http://dx.doi.org/10.1109/SP.1984.10019`
6. Kopetz, H., Ademaj, A., Grillinger, P., Steinhammer, K.: The time-triggered ethernet (tte) design. 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), Seattle, Washington (May 2005)
7. Koss, D., Bytschkow, D., Gupta, P.K., Schätz, B., Sellmayr, F., Bauereiß, S.: Establishing a smart grid node architecture and demonstrator in an office environment using the soa approach. In: Software Engineering for the Smart Grid (SE4SG), 2012 International Workshop on. pp. 8–14. IEEE (2012)
8. Mantel, H.: Possibilistic definitions of security - an assembly kit. In: CSFW. pp. 185–199. IEEE Computer Society (2000)
9. van der Meyden, R.: What, indeed, is intransitive noninterference? In: Biskup, J., López, J. (eds.) Computer Security ESORICS 2007, Lecture Notes in Computer Science, vol. 4734, pp. 235–250. Springer Berlin Heidelberg (2007), `http://dx.doi.org/10.1007/978-3-540-74835-9_16`
10. Rushby, J.: The design and verification of secure systems. In: Eighth ACM Symposium on Operating System Principles (SOSP). pp. 12–21. Asilomar, CA (Dec 1981), (ACM *Operating Systems Review*, Vol. 15, No. 5)
11. Rushby, J.: Noninterference, transitivity, and channel-control security policies. SRI International, Computer Science Laboratory (1992)
12. Zhang, C.: Conditional information flow policies and unwinding relations. In: Bruni, R., Sassone, V. (eds.) TGC. Lecture Notes in Computer Science, vol. 7173, pp. 227–241. Springer (2011)