A comprehensive approach to the automatic refinement and verification of access control policies

Manuel Cheminod[b], Luca Durante[b], Lucia Seno[b], Fulvio Valenza[a,b,*], Adriano Valenzano[b]

[a] *Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy*
[b] *CNR-IEIIT, c.so Duca degli Abruzzi 24, Torino I-10129, Italy*

Abstract

Access control is one of the building blocks of network security and is often managed by network administrators through the definition of sets of high-level policies meant to regulate network behavior (policy-based management). In this scenario, policy refinement and verification are important processes that have to be dealt with carefully, possibly relaying on computer-aided automated software tools.

This paper presents a comprehensive approach for access control policy refinement, verification and, in case errors are detected in the policy implementation, their fixing. The proposed methodology is based on a twofold model able to describe both policies and system configurations and allows, by suitably processing the model, to either propose a system configuration that correctly enforces the policies, or determine whether a specific implementation matches the policy specification also providing hints on how possible anomalies can be fixed. Results on the average complexity of the solution confirm its feasibility in terms of computation time, even for complex networked systems consisting of several hundred nodes.

Keywords: Access Control, Policy-Based Network Management, Policy Refinement, Policy Verification.

1. Introduction

Access control is a major building block of network security [1], regulating access of legitimate users to resources of a system. Access control by itself cannot prevent the occurrence of cyber-attacks, however effective protection schemes cannot abstract from both the definition of how access to system resources shall take place as well as the correct enforcement of the desired behavior in the system.

As target networked systems can be very complex, administrators often rely on policy-based paradigms for access control management (i.e., policy-based management, PBM [2]). Policies are technology-independent rules which define the desired network behavior from a high-level perspective. As such, they allow to separate the two problems of specification (i.e., definition of the desired network behavior) and implementation (i.e., actual enforcement of the desired behavior in the system) making network management easier and more flexible.

Access control policies regulate the access of users to the resources of a networked system by defining "*who is allowed to do what on what*". The application of PBM to access control has been receiving increasing attention by the scientific community since many years, and research on this topic has been focusing on three main domains, namely *policy analysis* (PA), *policy verification* (PV) and *policy refinement* (PR).

[*] Corresponding author

Email addresses: `manuel.cheminod@ieiit.cnr.it` (Manuel Cheminod), `luca.durante@ieiit.cnr.it` (Luca Durante), `lucia.seno@ieiit.cnr.it` (Lucia Seno), `fulvio.valenza@polito.it` (Fulvio Valenza), `adriano.valenzano@ieiit.cnr.it` (Adriano Valenzano)

Policy analysis deals with the fulfillment of specific properties by a set of policies [3]. The goal of PA is the detection of specification inconsistencies, arising when two or more policies are conflicting. PA only concerns the specification domain (i.e., the policies) and does not consider the implementation domain, i.e., how policies are actually enforced in the system.

Policy refinement bridges the gap between specification and implementation. Indeed, a PR process translates high-level policies into low-level system configurations [4]. PR is a critical task that, if not carefully performed, may lead to either incorrect or sub-optimal implementations, thus affecting important network performance indexes such as throughput and even jeopardizing the overall system security. When PR is performed by hand the risk of errors increases. The Verizon Data Breach Investigation Report points out that 14% of breaches in 2017 were caused by human errors in network configuration [5]. For this reason, computer-aided automatic or semi-automatic tools are definitely needed to assist administrators in the translation of access control policies.

Finally, PV deals with checking whether a set of policies is correctly enforced in a system or, in other words, whether the system implementation actually matches the policy semantics and no anomalies are present. As an example, policies may state that "*administrators can login in all nodes*", but a mismatched firewall configuration may result in cutting off some server. PV can also be used to validate a hand-made refinement. Note that, sometimes, performing a full refinement of access control policies is not convenient. In particular, when policies change in large networks, administrators verify whether the new policies are already correctly enforced with the current system configuration and, only if the result is negative, they introduce the required modifications. In this case the verification process plays a crucial role.

In this paper we present a comprehensive approach to automatic access control (AC) policy refinement and verification. Our proposal builds on both the twofold model introduced in [6], which enables the description of high-level AC policies and low-level implementation details of a target networked system, and some promising results described in [7].

While the solution in [6] was only meant for a posteriori policy verification, the technique shown in this paper extends [6] by focusing on credential assignment as a means to enforce AC policies. This is leveraged to solve the following problems, commonly tackled by security administrators:

1. Is there a user credential assignment which allows the implemented system to satisfy the policies? (*policy refinement*);

2. Can a partial user credential assignment be completed so that the system behavior fulfills the policy specification? (*constrained policy refinement*);

3. Does a specific user credential assignment make the system behavior match the policy specification? (*policy verification*);

4. How should the credential assignment be modified to fix anomalies (i.e., discrepancies between specification and implementation)? (*anomaly fixing*).

In practice, questions above refer to three different situations, that is:

1. no credential has been granted yet;

2. some credentials have already been assigned (constrained credentials);

3. all credentials have been assigned.

Rough connections to the state of a system development are then: under design, partially implemented and fully implemented.

The paper is structured as follows: Sect. 2 summarizes the characteristics of the twofold model employed for policy and system description. Sect. 3 and 4, respectively, formally present the proposed approach (and how to solve the considered problems) and provide conceptual workflows describing how administrators can be assisted in managing access control for their network in different circumstances. Sect. 5 describes some optimizations introduced in the actual implementation of the proposed solution and Sect. 6 deals with the automated tool and its performance. Finally, Sect. 7 considers some related works and Sect. 8 draws some conclusions and provides hints on future work.

2. Background

The twofold model presented in [6] enables the description of both a set of high-level access control policies (through a *Specification Model* $\mathcal{S}$) and the fine-grained details of a target networked system (through an *Implementation Model* $\mathcal{I}$). Model $\mathcal{I}$, in particular, stores information about the system structure, components and configurations, i.e., nodes and their interconnections, communication protocols, physical places where nodes are placed (e.g. rooms and cabinets), doors between adjacent spaces, software applications running on nodes, available and exploited access control mechanisms, credentials owned by users, etc.

By suitably processing models $\mathcal{S}$ and $\mathcal{I}$, three sets of pairs, S^+, S^- and I can be computed such that

$$S^+, S^-, I \subseteq Users \times Actions \tag{1}$$

where $Users$ and $Actions$ are sets identifying, respectively, all users u possibly interacting with the system and all actions α that can be performed in the system.[1] In the following, we use u and α to mean a generic element of set $Users$ and $Actions$, respectively.

Sets S^+ and S^- derive from $\mathcal{S}$ and define, respectively, *allowed* and *denied* users actions. Basically, whenever a pair $(u, \alpha) \in S^+$ it means that, according to the policies, user u is allowed to perform action α on the system. Conversely, if $(u, \alpha) \in S^-$, according to the policies, user u must not be able to perform action α on the system.

In the following, $S^+_{\tilde{u}}$ and $S^-_{\tilde{u}}$ are the sets of allowed and denied actions specified for user $\tilde{u} \in Users$, i.e.

$$S^{+/-}_{\tilde{u}} := \{(u, \alpha) \in S^{+/-} \mid u = \tilde{u}\} \tag{2}$$

Clearly, $S^{+/-} = \bigcup_{u \in Users} S^{+/-}_u$. Moreover, we assume $S^+ \cap S^- = \emptyset$ and, consequently, $S^+_u \cap S^-_u = \emptyset \ \forall \ u \in Users$, (i.e., policies are supposed to be coherent and, as such, conflict free).

The *Implementation* set I is derived from $\mathcal{I}$, and pairs belonging to I describe all actions that users are actually allowed to perform on the system, as determined by its current implementation.

Set I is only a portion of the information obtained by processing model $\mathcal{I}$. Indeed, by suitably combining $\mathcal{I}$ with a set of purposely defined inference rules, a collection of automata A_u can be computed (one automaton for each user $u \in Users$), such that

$$A_u = (Q_u, \Sigma_u, \delta_u, q^0_u) \tag{3}$$

Equation (3) adopts the traditional notation for Deterministic Finite State Automata (DFSA) [8] where Q_u is the set of states q_u, $q^0_u \in Q_u$ is the initial state, $\Sigma_u \subseteq Actions$ is the set of events, i.e. user's actions, and δ_u is the transition function of the automaton. Overall each automaton A_u stores the sequences of actions α allowed to user u, given a specific system implementation described by $\mathcal{I}$, and how her/his state varies accordingly.

Inference rules describe the way user/system interactions take place (e.g., how a user can move, stepping a door, between adjacent rooms or how s/he can access a file if s/he is logged on its host and has the required privi-

[1] In [6], *Actions* is the set of all pairs obtained by combining elements of sets *Operations* and *Objects*, describing all objects of the system and all operations that can be performed on at least one object in the system.

leges and/or credentials). Each automaton A_u is computed by considering the interaction of user u with the system and applying all inference rules to register actions s/he is able to perform and how this affects her/his state (i.e., the room s/he is in, access to local resources s/he gains).

Automata A_u are used to derive sets I_u. Each I_u consists of all pairs (u, α) obtained by combining user u with all α appearing in at least one transition of A_u. The implementation set I can then be obtained as

$$I = \bigcup_{u \in Users} I_u$$

The problem of policy verification can formally be assessed by comparing sets S^+, S^- and I. In particular, by defining the anomaly sets containing all actions respectively allowed and denied to users in violation of the policies as

$$\bar{S}^+ := S^+ \setminus I \qquad \bar{S}^- := S^- \cap I \tag{4}$$

we obtain that the implementation satisfies the policy specification if and only if

$$\bar{S}^+ = \emptyset \quad \wedge \quad \bar{S}^- = \emptyset \tag{5}$$

Note that for the implementation to match the specification, $I = S^+$ is not a necessary condition. Indeed, the high-level specification of policies may put into evidence only a subset of actions that are possible in the systems. In other words I may contain pairs not explicitly considered in the policies defined by the network administrators. As an example, I may contain a pair (u, α) where $\alpha = (login, PC)$. This kind of actions is useful to describe how access to some of the system resources actually takes place (e.g., a user, for accessing a specific software application running on PC, may first need to login on the PC itself). Often, these actions are not taken into account by policies, which in turn, do not have any knowledge of where applications are actually installed. Policies may explicitly state that u shall be able to access a specific software application without mentioning logging in on the PC hosting the application itself. If not otherwise specified in the denied action set, this means that for a correct implementation it is not important whether user u is able to login on PC or not.

Since automata are built on a per-user basis, an easy way to check the correctness of the policy implementation is by computing (4) and checking (5) $\forall\ u \in Users$ individually, i.e.:

$$\bar{S}_u^+ := S_u^+ \setminus I_u \qquad \bar{S}_u^- := S_u^- \cap I_u \tag{6}$$

$$\bar{S}_u^+ = \emptyset \quad \wedge \quad \bar{S}_u^- = \emptyset \tag{7}$$

3. Formal description of the approach

Approach in [6] only focused on policy verification and, in particular, on computing sets $\bar{S}^+$ and $\bar{S}^-$ and checking (5), to possibly highlight the need for changes in the current implementation. However, such a technique was unable to provide hints on how these system modifications had to be carried out, leaving this critical task completely to the network administrators.

This work extends the scope of [6] by including both policy refinement and verification in the proposed solution and by automatically suggesting fixings for the detected anomalies. Modifications to the implementation (reflecting on $\mathcal{I}$), may, in principle, involve different aspects of the target system (e.g., network topology, node configurations, etc.). Of course, some changes are very complex and may affect the system functionality. In this paper we take into account only changes pertaining to user credential assignments, as they are easy to manage and often help with fixing a large number of detected anomalies.

The main idea is determining relations among actions α and credentials needed to perform them, i.e., what sequences of actions and, consequently, what sets of credentials allow users to gain privileges enabling the execution of a specific action. This information is already present in $\mathcal{I}$ [6], as it is necessary to check whether an inference rule can fire, leading to a state transition in (3). In the following C is the set of all credentials in $\mathcal{I}$, and set $C_u \subseteq C \; \forall \; u \in Users$ stores the credentials owned by user u. When a transition can fire in A_u (3), the user's credential $c \in C_u$ allowing the corresponding action is known, even though it was not included in the transition label in [6] because of its restricted goal.

By contrast, credentials are also used here to fix anomalies detected through sets (6), and this imposes managing c allowing α explicitly in the corresponding automaton transition labels. More formally, from now on, each transition of any automaton A_u is associated with an event $e \in \Sigma_u \subseteq Actions \times (C \cup \{\varepsilon\})$. Thus, in general, event e has the form (α, c), i.e. it is a pair combining an action and a credential requested to perform that action (if any). Symbol ε means "no credential", i.e., whenever a transition is labeled with $e = (\alpha, \varepsilon)$, no specific credential is necessary to carry out α. All algorithms presented in [6] to build automata A_u still work properly here, considering events as described above.

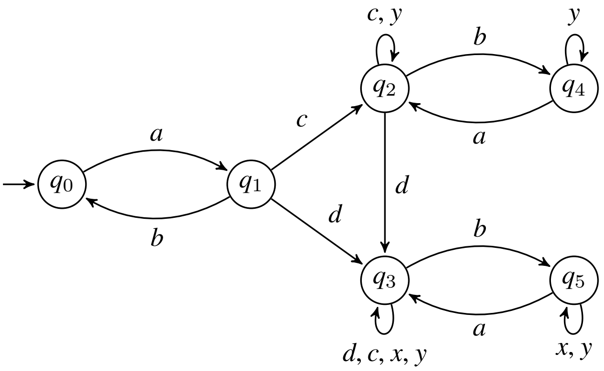
To minimize the computational effort, we compute a single automaton A_u referring to a hypothetical super-user $u = sup$. Superuser sup is defined as the user who is assigned all credentials existing in the system, i.e. $C_{sup} = C$. Automaton $A_{sup} = (Q_{sup}, \Sigma_{sup}, \delta_{sup}, q^0_{sup})$ is built exactly in the same way as automata A_u. While each automaton A_u describes the interactions between the system and a specific user u, automaton A_{sup} represents the most powerful interaction allowed by the system characteristics. Indeed, as it models the interaction achievable by owning all possible credentials, A_{sup} generates a maximum language consisting of all sequences of actions any user can perform in the system. As such, A_{sup} describes the system global behavior and is useful to examine dependencies among actions avoiding the computation of as many A_us as $u \in U$. Information about any user $u \in U$ can easily be derived directly from A_{sup} whenever needed.

The following sections deal with the computation, from automaton A_{sup}, of expressions specifying whether a user is enabled to carry out an action depending on the credentials s/he has. Moreover, we also discuss how these expressions can help in tackling the refinement, constrained refinement and verification problems. In more detail, starting with a generic DFSA, section 3.1 shows how to compute a collection of sets of enabling events $\forall \; e \in \Sigma$, such that e can be impeded by disabling at least one event in each set. Section 3.2, instead, makes use of sets of enabling events for automaton A_{sup} to determine $\forall \; e \in \Sigma_{sup}$ a boolean function (enabling function) expressing what credentials are needed to perform the action guarded by e. Finally, sections 3.3, 3.4 and 3.5 apply enabling functions to policy refinement, constrained refinement and verification respectively.

3.1. Enabling events

Let us consider the simple automaton $G = (Q, \Sigma, \delta, q_0)$ shown in Fig. 1, where events $e \in \Sigma$ are (α, c) pairs mentioned above. G is a super-user automaton computed for an example system where:

- two adjacent rooms (A and B) are connected through a door which can be opened with key k_{AB};
- two system nodes (host H_1 and server S_1), connected through the network infrastructure, are located in B;
- two different passwords (*user* and *admin*) can be used to sign in on H_1;
- whenever a user is logged in as an administrator on H_1, s/he can perform the *backup* action (e.g., invoke a backup service), while simple users are not allowed to do so;
- any user logged in H_1 can access, remotely through the network, a database hosted by S_1.



Short label	Complete event label
a	$(enter,\ A),\ k_{AB}$
b	$(enter,\ B),\ k_{AB}$
c	$(login,\ H_1),\ pw\_uh_1$
d	$(login,\ H_1),\ pw\_ah_1$
x	$(backup,\ H_1),\ \varepsilon$
y	$(access,\ DB),\ pw\_db$

Figure 1: Example automaton G

For sake of simplicity short labels have been used for edges in G, i.e., $\Sigma = \{a, b, c, d, x, y\}$, while corresponding (complete) labels obtained through the computation of the automaton from $\mathcal{I}$ are listed in the table below the automaton, e.g. event a stands for a pair $(\alpha,\ c)$ where action α is $(enter,\ A)$ whereas credential c is k_{AB}. $\mathcal{L}(G)$ is the language generated by automaton G.

Definition 1. *Given an event set Σ and a string $s \in \Sigma^*$, we define operator $\mathcal{E}(\cdot) : \Sigma^* \rightarrow 2^{\Sigma}$ as:*

$$\mathcal{E}(s) = \{e \in \Sigma \mid \exists\ t, v \in \Sigma^* : s = tev\}$$

Given a string s of events of Σ, operator $\mathcal{E}(\cdot)$ returns the set of all events in s. For instance, if we consider strings $s_1 = accyby$, $s_2 = aba$ and $s_3 = \varepsilon$ belonging to $\mathcal{L}(G)$ in Fig. 1 and apply $\mathcal{E}(\cdot)$ to each one of them, we obtain $\mathcal{E}(s_1) = \{a, c, y, b\}$, $\mathcal{E}(s_2) = \{a, b\}$ and $\mathcal{E}(s_3) = \{\varepsilon\}$, respectively.

Definition 2. *Given an automaton $G = (Q, \Sigma, \delta, q_0)$ and an event $e \in \Sigma$, a set of events $E \subseteq \Sigma \setminus \{e\}$ enables event e in G ($E \rhd e$) as:*

$$E \rhd e := \begin{cases} \exists\, s \in \Sigma^* : \begin{cases} se \in \mathcal{L}(G) \\ \wedge \\ \mathcal{E}(s) = E \end{cases} \\ \wedge \\ \nexists\, s' \in \Sigma^* : \begin{cases} s'e \in \mathcal{L}(G) \\ \wedge \\ \mathcal{E}(s') \subset E \end{cases} \end{cases} \tag{8}$$

Basically, a set of enabling events for an event e is the smallest set of events whose combined occurrence enables event e.

For instance, considering G in Fig. 1, we have $\{a\} \rhd b$, $\{a\} \rhd c$, $\{a\} \rhd d$, and $\{a, d\} \rhd y$, while $\{a, d, x\} \not\rhd y$ being $\{a, d\} \subset \{a, d, x\}$.

Definition 3. *Given an automaton $G = (Q, \Sigma, \delta, q_0)$ and an event $e \in \Sigma$, we define operator $En(\cdot) : \Sigma \rightarrow 2^{\Sigma}$ as:*

$$En(e) = \{E \subseteq \Sigma \ \mid\ E \rhd e \ in\ G\} \tag{9}$$

Given a generic automaton modeling a system, any event e in its set Σ is characterized by a (possibly empty) set of sets of enabling events $En(e)$. From the definitions above it follows that for e to occur, all events in at least one of sets $E \in En(e)$ must happen and, consequently, to prevent e from happening at least one element in each set $E \in En(e)$ must be disabled. Moreover, from (8), any pair of sets $E_1, E_2 \in En(e)$ may share some elements but $E_1 \subset E_2$ is never true. Note that the set of enabling events for $e \in \Sigma$ may be $En(e) = \{\{\varepsilon\}\}$ meaning that e can occur directly in the initial state of G (i.e., it is not enabled by other events). Moreover, $En(e) = \{\{\varepsilon\}\}$ is different from $En(e) = \emptyset$, as the latter means that $e \in \Sigma$ can never happen in G (i.e., no state having an outgoing transition labeled as e is reachable from q_0).

The computation of $En(e), \forall e \in \Sigma$ for G in Fig. 1 leads to:

$$\begin{array}{rclr}
En(a) & = & \{\{\varepsilon\}\} & En(b) = \{\{a\}\} \\
En(c) & = & \{\{a\}\} & En(d) = \{\{a\}\} \\
En(x) & = & \{\{a, d\}\} & En(y) = \{\{a, c\}, \{a, d\}\}
\end{array}$$

3.2. Enabling functions

The following definitions apply to automata having the characteristics of A_{sup}. In particular, given an event $e = (\alpha, c) \in \Sigma$, we define $C(\cdot) : \Sigma \rightarrow (C \cup \{\varepsilon\})$ and $\mathcal{A}(\cdot) : \Sigma \rightarrow Actions$ such that:

$$C(e) = c \qquad\qquad \mathcal{A}(e) = \alpha$$

Each event $e = (\alpha, c)$, can be disabled for user u either directly (by denying credential c to u), or indirectly (by disabling at least one event in any set belonging to $E \in En(e)$ for u). Note that events e where $C(e) = \varepsilon$ can be disabled only indirectly.

In this condition, an event e can happen only if a user owns credential c for e together with the credentials for all events in at least one of the sets in $En(e)$. This can be described by means of a boolean expression where a variable is associated to each credential, whenever the value of the variable is 0 it means that the credential is not provided to the user, while, on the contrary, when the variable assumes value 1 the user is assigned the corresponding credential. A sum of variable products is then specified for each $E \in En(e)$.

As an example, let us consider event y in Fig. 1: $En(y) = \{\{a, c\}, \{a, d\}\}$. The corresponding action ($access$, DB) can be performed only by users who own the credential for y together with those allowing (a `and` c) `or` (a `and` d), i.e.

$$pw\_dB \cdot k_{AB} \cdot pw\_uh_1 + pw\_dB \cdot k_{AB} \cdot pw\_ah_1 \quad (10)$$

Any assignment of variables making expression (10) true corresponds to a set of credentials enabling u to perform action ($access$, DB) in the example.

More in general, for each $e = (\alpha, c) \in \Sigma_{sup}$, given its set of sets of enabling events $En(e)$, we build a boolean function $F(\alpha)$ (*enabling function*) as sketched above.

Definition 4. *Given the set C of all credentials in $\mathcal{I}$, we define a set of boolean variables*

$$V = \{v_c \mid c \in C\} \qquad (11)$$

where v_c is the variable associated to credential c.

For the example automaton in Fig. 1, we have that set $V = \{pw\_db,\ k_{AB},\ pw\_uh_1,\ pw\_ah_1\}$, where each variable has been named according to the associated credential.

Definition 5. *Given an automaton $A_{sup} = (Q_{sup}, \Sigma_{sup}, \delta_{sup}, q^0_{sup})$ and an event $e \in \Sigma_{sup}$, $f(e)$ is the boolean expression*

$$f(e) := v_c \cdot \left(\sum_{E \in En(e)} \prod_{\tilde{e} \in E} v_{C(\tilde{e})} \right) \qquad (12)$$

For instance, $f(y)$ for system G in Fig. 1 is equation (10). Event $e = (\alpha, c)$ can occur only if the user has credential c, i.e. $f(e)$ cannot be true if v_c is false. In addition, the user must own all credentials requested for all events in at least one $E \in En(e)$, i.e. the corresponding variable product must be true.

In the example system G, action ($login, H_1$) appears in both events c and d, and credentials pw_uh_1 and pw_ah_1 are respectively needed to enable them. This situation, which happens frequently in real systems, has to be handled explicitly as policies deal with actions irrespectively of the credentials needed to carry them out.

Definition 5 can then be extended to define how action α can be performed whatever the credential requested for it:

Definition 6. *Given an automaton A_{sup} and an action α such that $(\alpha, c) \in \Sigma_{sup}$*

$$F(\alpha) := \sum_{c \in C \ \mid \ (\alpha, c) \in \Sigma_{sup}} f(\alpha, c) \qquad (13)$$

Once again let us consider action $(login, H_1)$: from (12) we have:

$$f(login, H_1, pw\_uh_1) := pw\_uh_1 \cdot k_{AB}$$

$$f(login, H_1, pw\_ah_1) := pw\_ah_1 \cdot k_{AB}$$

and by applying (13) we obtain:

$$F(login, H_1) := pw\_uh_1 \cdot k_{AB} + pw\_ah_1 \cdot k_{AB}$$

Basic definitions above are useful to address the *policy refinement*, *constrained refinement*, and *verification and fixing* problems. Refinement means finding a user's credential assignment allowing u to carry out all actions in S_u^+, and preventing u from performing any $\alpha \in S_u^-$. Constrained refinement is similar and consists of completing a partial credential assignment already provided by network administrators, while verification and (possibly) fixing is when a complete, already available assignment has to be checked for correctness.

Some actions α belonging to *Actions* might not appear in some $e \in \Sigma_{sup}$. When this happens, i.e. α is not present in A_{sup}, $F(\alpha) = 0$ by definition:

Definition 7. *Given an automaton A_{sup} and an action $\alpha \mid \nexists(\alpha, c) \in \Sigma_{sup}$*

$$F(\alpha) := 0 \tag{14}$$

3.3. Policy refinement

Our solution for policy refinement is meant to determine a user credential assignment such that the system implementation matches the policy specification, i.e., (5) holds true. The process works on a per-user basis and tries to select values for variables of V, such that each user is enabled to perform all allowed actions (i.e., all $(u, \alpha) \in S_u^+$) while is prevented from performing all denied actions (i.e., all $(u, \alpha) \in S_u^-$).

From (13) and (14) u is able to perform all allowed actions if and only if:

$$F(\alpha) = 1 \quad \forall\ \alpha \mid (u, \alpha) \in S_u^+$$

Similarly, the system prevents u from performing any of the denied actions if and only if:

$$\overline{F(\alpha)} = 1 \quad \forall\ \alpha \mid (u, \alpha) \in S_u^-$$

By combining conditions above, a suitable credential assignment for u is such that

$$\left(\prod_{\alpha \in S_u^+} F(\alpha)\right) \cdot \left(\prod_{\alpha \in S_u^-} \overline{F(\alpha)}\right) = 1 \tag{15}$$

Consequently we can assert that, in our case, solving the refinement process means *finding, for each user u, an assignment of values for variables in V such that* (15) *holds.*

Note that such an assignment is *partial* with respect to V, since values are only found for variables explicitly appearing in (15). Variables not appearing in (15) can be safely set to 0 without affecting the policy implementation for user u. Thus a *complete* assignment $\sigma_u : V \to \{0, 1\}$, is obtained by simply assigning default values to variables not present in (15).

When no solution is found, there is no way for the current system implementation $\mathcal{I}$ to satisfy the policies by acting on the credential assignment only, and the problem is marked as *unsat*, i.e. no $\sigma_u(\cdot)$ is defined. In this case, administrators have either to change other elements of the data model (i.e., system topology, firewall and service configurations and so on) or relax some policy constraints if this is compatible with the system security requirements.

3.4. Policy constrained refinement

Sometimes administrators have already identified a partial credential assignment for user u, i.e., a function $\sigma_u^c : V_u \to \{0, 1\}$ such that any $v_c \in V_u \subset V$ is bound to an unchangeable value $\sigma_u^c(v_c)$. The refinement process should then complete the assignment so as to make (5) true.

Solving the constrained refinement problem means *finding for each user u an assignment for $\{v_c\} = V \setminus V_u$, such that* (15) *holds when each $v_c \in V_u$ is replaced by $\sigma_u(v_c)$.*

Even in this case, the solution satisfying (15), if any, can be used to build $\sigma_u : V \longrightarrow \{0, 1\}$, by joining it to $\sigma_u^c(\cdot)$, and setting all variables which are not bound by σ_u^c and do not appear in (15) to the default value (0).

3.5. Policy verification

A complete credential assignment for user u is a function $\sigma_u^V : V \longrightarrow \{0, 1\}$ where each $v_c \in V$ is bound to either 0 or 1. Thus, solving the verification problem means *checking, for each user u, whether a credential assignment $\sigma_u^V(\cdot)$ enables a correct implementation of policies in the system.*

In particular, sets $\bar{S}_u^+$ and $\bar{S}_u^-$ in (6) can be computed by means of the enabling functions as:

$$\begin{aligned} \bar{S}_u^+ &= \{(u, \alpha) \mid F(\alpha)|_{\sigma_u^V} = 0 \ \wedge \ (u, \alpha) \in S_u^+\} \\ \bar{S}_u^- &= \{(u, \alpha) \mid F(\alpha)|_{\sigma_u^V} = 1 \ \wedge \ (u, \alpha) \in S_u^-\} \end{aligned} \tag{16}$$

where notation $F(\alpha)|_{\sigma_u^V}$ stands for "$F(\alpha)$ where each variable v_c is replaced by $\sigma_u^V(v_c)$".

Three additional formal definitions are used in the following sections:

Definition 8. *Given an action α, $V_\alpha \subseteq V$ is the set of all boolean variables v_c appearing in the enabling function $F(\alpha)$*

$$V_\alpha = \{v_c \mid v_c \textit{ is a variable in } F(\alpha)\} \tag{17}$$

Definition 9. *Given an automaton A_{sup} and a user u, $\check{K}_u$ is the set of all actions:*

$$\check{K}_u := \left\{\mathcal{A}(e) \mid e \in \Sigma_{sup},\ (u, \mathcal{A}(e)) \notin \bar{S}_u^+ \cup \bar{S}_u^-\right\} \tag{18}$$

In practice, $\check{K}_u$ is the set of actions which, allowed or denied under the current system configuration, are not identified as anomalies.

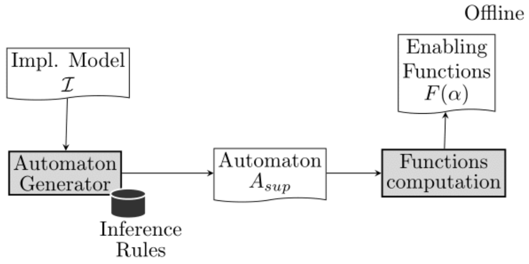


Figure 2: Workflow: offline phase

Definition 10. *Given $\check{K}_u$, $\check{V}_u$ is the set of all variables:*

$$\check{V}_u := \bigcup_{\alpha \in \check{K}_u} V_\alpha \tag{19}$$

Basically, $\check{V}_u$ is the set of variables v_c associated to credentials enabling actions in $\check{K}_u$.

4. Workflow

This section shows how the approach formally presented in Sect. 3 is put at work, while most significant details about the implementation of our technique are provided in Sects. 5 and 6.

From a practical point of view, we can distinguish two different operating phases, which are respectively referred to as *offline* and *online*. The *offline* phase (see Fig. 2) aims at modeling and analyzing the whole system and, consequently, is computation intensive and can be rather time-consuming. Fortunately, it is seldom carried out in practice, since changes in the system characteristics such as i.e., host locations, network topology, installed services etc. do not occur frequently with respect to modifications of access control policies. The *online* phase (see Fig. 3), instead, should be performed every time administrators configure or change some access control policies for any given user/group of users, or whenever the correctness of configurations has to be rechecked.

More specifically, the *offline* process (Fig. 2) is responsible for computing the set of actions whose execution enables action α. Starting from $\mathcal{I}$ and a set of predefined inference rules automaton A_{sup} is generated and stored into

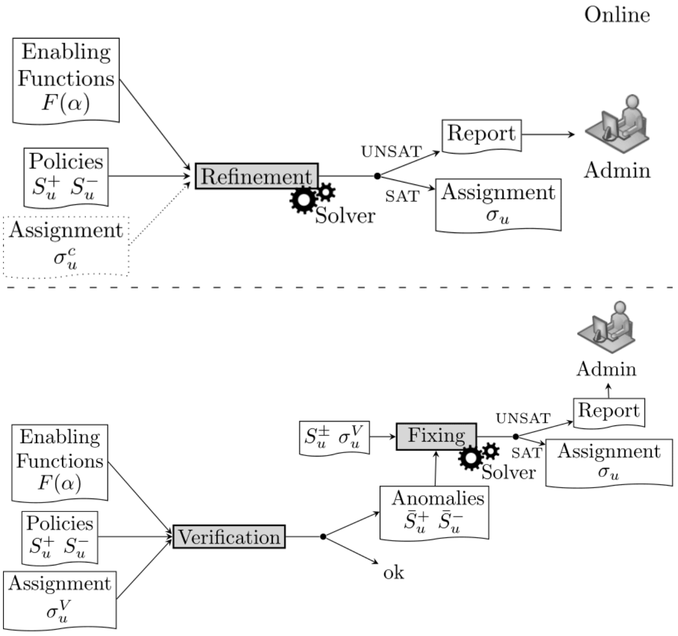


Figure 3: Workflow: online phase

a database (automaton generator block in Fig. 2). The second step of the *offline* phase (functions computation) consists of visiting A_{sup} to compute the enabling functions $F(\alpha)$ in (13) for all possible actions in the system.

The *online* phase works on a per-user basis and includes two distinct sub-processes, i.e. *refinement* and *verification and fixing* in Fig. 3.

4.1. Refinement process

Refinement aims at identifying a credential assignment $\sigma_u(\cdot)$ for any user u (or group of users), as requested to correctly enforce the policies. Note that this problem may have one, many or no solution at all.

By assuming that all solutions are equivalent, a Satisfiability Modulo Theories (SMT) solver [9] is used to find either a correct assignment or a proof that no satisfactory assignment exists. In details, the refinement process (RP) tries to solve equation (15) by taking both the enabling (boolean) functions $F(\alpha)$ and the user policies (S_u^+ and S_u^-) as inputs. RP returns either a set of credentials $\sigma_u(\cdot)$, that correctly enforce the policies, or a counter-example for non-satisfiability. The counter-example is then reported to the administrators along with the set of conflicting policies. It is up to the administrators analyzing results and possibly introduce changes in either the polices or the data model (that is the system).

When dealing with constrained refinement, RP takes the administrator-defined credential assignment $\sigma_u^c(\cdot)$ as additional input (dotted block in Fig 3) and binds the relevant credentials to the predefined values. Of course, the larger the number of bounded variables the smaller the solution space and, consequently, the higher the likelihood of unsatisfiability.

Typically, the problem of constrained refinement arises when administrators directly assign some credentials for practical or business-related reasons, e.g., employees are usually directly assigned the password for accessing their workstations or administrators are assigned credentials for logging in as superusers on some hosts. In all these cases credential assignment is already partially defined.

4.2. Verification and Fixing

Given a credentials assignment $\sigma_u^V(\cdot)$ for each user u, possible anomalies are detected by computing (16) so as to produce sets $\bar{S}_u^+$ and $\bar{S}_u^-$.

Moreover, when anomalies are detected, a possible fixing (if any) is automatically suggested to administrators. The fixing computation is based on a constrained refinement, where the credential assignment for actions not belonging to $\bar{S}_u^+ \cup \bar{S}_u^-$ is left unchanged. Credentials only concerning actions in the anomaly sets (i.e., not belonging to set $\check{V}_u$ in (19)) are left unbound, and a refinement is then searched. Roughly speaking, the fixing process (FP) tries to leave as many credentials as possible unassigned, and puts back into the game only those variables that are directly involved in anomalies, i.e. credentials enabling actions belonging to the anomaly sets. In case no solution is found, administrators can take different actions such as modifying the initial credential assignment or exploiting other characteristics of $\mathcal{I}$. Options available to administrators when the automatic fixing fails are not in the scope of this paper.

5. Offline phase optimization

Some properties of A_{sup} can be leveraged to tackle the problem of computing $F(\alpha)$ and to solve it efficiently.

The computation of an enabling function $F(\alpha)$ relies on the identification of sets of enabling events $En(e)$. Finding $En(e)$, in a generic automaton G, means finding all those paths[2] p such that $pe \in \mathcal{L}(G)$ and $\mathcal{E}(p) \in En(e)$.

This leads to an enumeration problem whose complexity depends on the size and structure of G. Unfortunately, given the number of events $|\Sigma|$, the number of paths to the target event e can be as large as $|\Sigma|!$. Actually, the worst case condition occurs when e is enabled by a combination of all the other events disjointedly. The computation complexity in the worst case is $O(|\Sigma|!)$, so an optimization strategy is needed to make the problem manageable. The main idea we followed is splitting the problem into two simpler parts (*divide et impera* approach), solve them individually and then recombine the partial results to efficiently obtain $En(e)$ for each $e \in \Sigma$.

5.1. Divide et impera

Given its characteristics, A_{sup} can be split into two simpler automata A^r and A^L ($A^r \ll A_{sup}$ and $A^L \ll A_{sup}$), where $A^r = (Q^r, \Sigma^r, \delta^r, q_0^r)$ and $A^L = (Q^L, \Sigma^L, \delta^L, q_0^L)$. Their combination by means of the standard parallel operation [8], returns A_{sup} ($A^r || A^L = A_{sup}$). This enables the computation of two enabling sets of events for A^r and A^L (respectively $En^r(e)$ and $En^L(e)$ in the following) which can be combined to build $En(e)$.

Automata A^r and A^L are built by considering complementary aspects of the system. Indeed, A^r takes into account both the physical interactions of a user with the nodes in the system and her/his movements between rooms. Instead A^L only deals with aspects concerning the acquisition and exploitation of logical accesses to the devices/services. In building A^L we assume that the user can reach, moving from the initial state, all locations that have been explored while building A^r (see Axiom 1 in the following).

In general, actions $\alpha \in Actions$ belong to three disjoint groups: θ_{I}, θ_{II} and θ_{III}. Actions in θ_{I} depend on the user physical position and can change it. Actions in θ_{II} depend on the user's position and, possibly, let her/him gain access to a device. Finally, actions in θ_{III} depend on the set of logical accesses already acquired by the user and, possibly, let her/him enlarge it with a new element.

The construction of A^r combines only α in θ_{I} and θ_{II}, while A^L takes into account θ_{II} and θ_{III}. Events that are in both Σ^r and Σ^L belong to θ_{II}, and A^r describes how sequences of θ_{I} actions can enable a subset of θ_{II}. Similarly, A^L shows how the subset of θ_{II} can trigger sequences in θ_{III}.

For example, G in Fig. 1 can be obtained by the parallel composition of the two automata A^r and A^L shown in Fig. 4 and Fig. 5 respectively. In this case $Actions = \{a, b, c, d, x, y\}$, $\theta_{\mathrm{I}} = \{a, b\}$, $\theta_{\mathrm{II}} = \{c, d\}$ and $\theta_{\mathrm{III}} = \{x, y\}$.

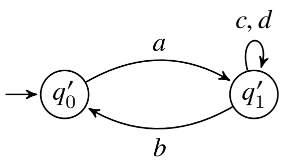


Figure 4: Example A^r

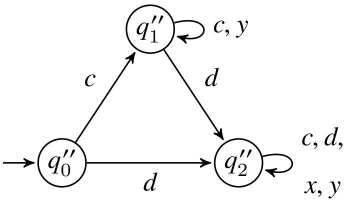


Figure 5: Example A^L

Let $\Gamma(q)$ be the set of events enabled in state q: $\Gamma(q) = \{e \in \Sigma \ \mid \delta(q, e) \text{ is defined}\}$. For instance, $\Gamma(q_0'') = \{c, d\}$ and $\Gamma(q_1'') = \{c, y, d\}$ in Fig. 5.

The complete and formal definition of the procedures used to build A, A^r and A^L has been provided in [6]. Some useful properties, referred in the following as axioms, stem from the characteristics of the data model and the inference rules. Axioms, allow to derive three new properties needed to build $En(e)$ from $En^r(\cdot)$ and $En^L(\cdot)$ in a very

[2] Since a string $s \in \Sigma^*$ is a sequence of events, every path p in G corresponds to a string in $\mathcal{L}(G)$ and we may refer to p as either a path or a string indifferently.

efficient way.

Axiom 1. *All actions $\alpha \in \theta_{II}$ that are allowed in A^r belong to A^L as well and, in particular, are enabled in the A^L initial state q_0^L.*

$$\forall e \in \Sigma^r \cap \Sigma^L, e \in \Gamma(q_0^L), En^L(e) = \{\{\varepsilon\}\}$$

Axiom 2. *Given a state $q \in Q^L$, each event $e \in \Gamma(q)$ is also enabled in any state q' reachable from q $(e \in \Gamma(q'))$.*

Axiom 3. *Any action in θ_{III} is enabled by a single logical access (and not by a simultaneous combination of multiple logical accesses acquired by the user).*

From Axiom 2 and Axiom 3 we have:

Property 1. *Given any two events e_1, e_2 in some set E in the enabling set $En(e)$ of a target event e, either e_1 is in the enabling set of e_2 or e_2 is in the enabling set of e_1.*

$$E \in En(e), \{e_1, e_2\} \subseteq E \Rightarrow$$
$$\exists E' \in En(e_2), e_1 \in E' \;\oplus\; \exists E' \in En(e_1), e_2 \in E'$$

From Axiom 1 and Property 1 we have:

Property 2. *For any event e in Σ^L and any $E \in En^L(e)$ two distinct events $\{e_1, e_2\} \subseteq E$ both belonging to $\Sigma^r \cap \Sigma^L$ can never exist.*

Moreover, in building A^L we exploit only actions belonging to θ_{II} and θ_{III}. Remembering that θ_{III} events only depend on logical accesses conditions already gained by the user, and the initial set of such conditions for any user is empty, any logical access in A^L can be acquired only by means of a θ_{II} action, that is events in $\Sigma^r \cap \Sigma^L$. As such, any sequence of events leading to $e \in \Sigma^L \setminus \Sigma^r$ starts with some specific action $\tilde{e} \in \Sigma^r \cap \Sigma^L$. Since all $e \in \Sigma^r \cap \Sigma^L$ are included in $\Gamma(q_0^L)$, we can conclude that:

Property 3. *For any event e in A^L, there is exactly one event $\tilde{e}$ in each $E \in En^L(e)$, such that it also belongs to Σ^r:*

$$\forall e, \forall E \in En^L(e), \exists! \tilde{e} \in E \mid \tilde{e} \in \Sigma^r \cap \Sigma^L \tag{20}$$

Because of the above properties, a simple procedure allows to combine $En^r(\cdot)$ and $En^L(\cdot)$ to build $En(e)$ for each e in $\Sigma^r \cup \Sigma^L$.

Given $e \in \Sigma^r$, $En(e)$ is simply defined by $En^r(e)$. In fact, if $e \in \Sigma^r \setminus \Sigma^L$ clearly $En^L(e) = \emptyset$, while if $e \in \Sigma^r \cap \Sigma^L$, $En^L(e) = \{\{\varepsilon\}\}$ for Property 1.

Instead, for every event $e \in \Sigma^L \setminus \Sigma^r$, Property 3 asserts that there is one event $\tilde{e}$ in any $E \in En^L(e)$ which is enabled in the initial state of A^L by the sets of events identified in $En^r(\tilde{e})$. Thus for any E included in some $En^L(e)$ we define $\mathcal{X}(E)$ as the "extended E", that is the set of sets built from the combination of all sets in $En^r(\tilde{e})$ and E itself.

$$\mathcal{X}(E) = \{E \cup E' \mid \forall E' \in En^r(\tilde{e}), \tilde{e} \in E \cap \Sigma^r\} \tag{21}$$

In different words, any set of events E is replaced by a set of sets of events that combines both events in E and all events in $En^r(\tilde{e})$ (that is, the enabling events of $\tilde{e}$).

Finally, we build $En(e)$ for any $e \in \Sigma^r \cup \Sigma^L$ as:

$$En(e) = \begin{cases} En^r(e), e \in \Sigma^r \\ \bigcup_{i=1}^{|En^L(e)|} \mathcal{X}(E), E \in En^L(e), e \in \Sigma^L \setminus \Sigma^r \end{cases} \tag{22}$$

By applying this procedure to the example in Figs. 4 and 5, we compute sets $En^r(\cdot)$ and $En^L(\cdot)$ for A^r and A^L and then combine the obtained results to build $En(y)$. $En^r(\cdot)$ and $En^L(\cdot)$ are the following:

$$\begin{array}{lcll} En^r(a) & = & \{\{\varepsilon\}\} & En^L(c) = \{\{\varepsilon\}\} \\ En^r(b) & = & \{\{a\}\} & En^L(d) = \{\{\varepsilon\}\} \\ En^r(c) & = & \{\{a\}\} & En^L(x) = \{\{d\}\} \\ En^r(d) & = & \{\{a\}\} & En^L(y) = \{\{c\}, \{d\}\} \end{array}$$

Then, since $y \in \Sigma^L \setminus \Sigma^r$, by (22), we have to compute $\mathcal{X}(E_1)$ and $\mathcal{X}(E_2)$, where $En^L(y) = \{E_1, E_2\}$ and $E_1 = \{c\}$, $E_2 = \{d\}$.

For $\mathcal{X}(E_1)$, $\tilde{e} \in E_1 \bigcap \Sigma^r$ and $\tilde{e} = c$. Since $En^r(c) = \{\{a\}\}$, then $\mathcal{X}(E_1) = \{\{c\} \cup \{a\}\} = \{\{c, a\}\}$. Similarly, for E_2 we have $\tilde{e} \in E_2 \bigcap \Sigma^r$, $\tilde{e} = d$ and $En^r(d) = \{\{a\}\}$, then $\mathcal{X}(E_2) = \{\{d\} \cup \{a\}\} = \{\{d, a\}\}$. Finally, $En(y) = \{\mathcal{X}(E_1) \cup \mathcal{X}(E_2)\} = \{\{c, a\}, \{d, a\}\}$

5.2. En^L computation optimization

The *divide et impera* technique greatly facilitates the En computation, because the enumeration of all paths in both A^r and A^L is simpler (the size of both automata is considerably smaller than A_{sup}). However, the worst case complexity is still $O(|\Sigma^r!|)$ and $O(|\Sigma^L!|)$ for the computation of En^r and En^L respectively.

A further improvement in evaluating En can be obtained by observing that, in general, $A^r \ll A^L$, or A^r is much smaller than A^L in terms of number of states and transitions. In a typical system, in fact, the number of rooms and actions requiring the physical proximity of a user to a device is significantly smaller then the number of actions that can be performed remotely ($|\theta_{\mathrm{I}}| + |\theta_{\mathrm{II}}| \ll |\theta_{\mathrm{II}}| + |\theta_{\mathrm{III}}|$). Thus making the enumeration of paths in A^L faster brings significant advantages.

A main consequence of properties introduced in Sect. 5.1 is that, given a target event e and a set of events E enabling two events a and b ($E \in En(a)$ and $E \in En(b)$), $E \cup \{a, b\} \notin En(e)$ is always true. This means that we can neglect further sequences including both a and b, so dramatically reducing the space of states in the automaton exploration.

A solution, which is able to exploit these properties, is shown by Algorithm 1. The algorithm takes the automaton A^L and the target event e as inputs and produces $En^L(e)$ of (9) as a result. The solution is based on a breadth-first approach where open states are accumulated in a queue T. A state t in the queue is defined as a triple (q, p, cut) where q is a state in Q, p is the path from q_0 to q and cut is the set of events to be excluded from $\Gamma(q)$.

Algorithm 1 Enumeration of $En^L(e)$ for A^L

```
Input: $A^L = (Q, \Sigma, \delta, q_0), e$
Output: $En^L(e)$
$T = \{\}; En^L(e) = \{\};$
$t = (q_0, \epsilon, \{\})$
enqueue$(t, T)$
while $T \neq \{\}$ do
    $(q, p, cut) =$ dequeue$(T)$
    if $e \in \Gamma(q)$ then
        $En^L(e) = En^L(e) \cup \mathcal{E}(p)$
    else
        for all $e' \in \Gamma(q) \setminus cut$ do
            if $\delta(q, e') \neq q$ then
                $q' = \delta(q, e')$
                $cut' = cut \cup \Gamma(q)$
                $t = (q', pe', cut')$
                enqueue$(t', T)$
            end if
        end for
    end if
end while
```

The exploration of A^L starts from state $(q_0, \varepsilon, \{\})$, where the path to q_0 is empty (i.e. ε) as well as the set cut. $(q_0, \varepsilon, \{\})$ is added to T and, as long as the queue is not empty, a state (q, p, cut) is extracted from T and processed.

If the target event e is in $\Gamma(q)$ this means that $pe \in \mathcal{L}(A^L)$. In this case, the exploration of the current branch is concluded and $\mathcal{E}(p)$ is stored in $En^L(e)$. Otherwise, for all events $e' \in \Gamma(q)$ that are not in cut and are not loops ($\delta(q, e') \neq q$), a new state $t' = (q', pe', cut')$ is generated, where $q' = \delta(q, e')$ and cut' includes all events in cut plus all those enabled in state q.

In summary, at each step of the A^L exploration, Algorithm 1 builds a path p and stores a set of events cut that are no longer considered in the current branch, since they have already been taken into account in some previous step. This set grows with every step, thus reducing significantly the number of events that have still to be evaluated in the following iterations.

The total number of paths depends on the number of new events available at each exploration step. Of course, the best case is when $\Gamma(q_0) = \Sigma$. $|\Sigma|$ one event-long paths are then built since, for each state $q' = \delta(q_0, e')$, the set cut is equal to $\Gamma(q_0)$.

Instead, the worst case occurs when new events become available and have to be considered at each step of the exploration procedure. In particular, the structure of the automaton with n states is such that: 1. the set Σ is partitioned in $\Sigma_1, \Sigma_2, \cdots, \Sigma_{n-1}$ subsets, i.e. $|\Sigma| = \sum_{i=1}^{n-1} |\Sigma_i|$ and 2. for each $e \in \Sigma_i$, e labels a transition from q_{i-1} to q_i. In this condition the number of paths that lead to the final state q_n is equal to $\prod_{i=1}^{n-1} |\Sigma_i|$. The maximum value is obtained when the product of all partition sizes is maximum and is approximately equal to $3^{|\Sigma|/3}$ [10]. Thus, the complexity of Algorithm 1, in the worst case, is $O(3^{|\Sigma|/3})$.

Algorithm 1 is suitable for exploring A^L and is able to build $En^L(e)$ for each $e \in \Sigma$. It is worth remembering that the same approach cannot be followed to build $En^r(e)$ as properties of A^L do not hold for A^r. Fortunately, A^r is much smaller than A^L in practice, so that $En^r(e)$ can be constructed by complete enumeration.

6. Implementation and performance

The methodology introduced in the previous sections has been implemented in a prototype software tool. The *Functions computation* block in Fig. 2 carries out the operations in Algorithm 1 by means of a purposely developed Java application, able to read the description of A^L and A^r ($A^r || A^L = A_{sup}$) and build the enabling functions $F(\alpha)$. Blocks in the *online* phase (Fig. 3), have been implemented by two custom Java-based modules, one devoted to refinement and fixing and the other to verification.

In particular, the refinement/fixing modules also include the "Z3" off-the-shelf SMT solver [9]. In this case, module inputs are the $F(\alpha)$ formulas, that is boolean expressions as needed by the Z3 library.

To provide some preliminary performance evaluation of the proposed approach, some tests were run using a number of synthetically generated scenarios, so as to have a rough estimation of processing times with different sized systems. Test scenarios were generated starting with a small real use case, which is able to stress main aspects of the solution, and by replicating the basic (partial) scheme so as to create several more complex systems of increasing size.

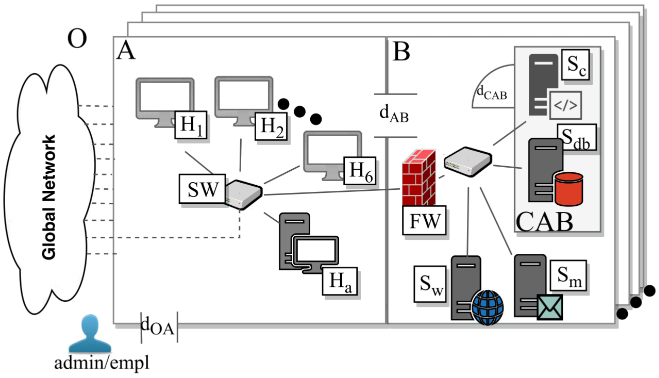


Figure 6: Basic system adopted for tests.

6.1. Basic target system

Fig. 6, shows a number of office areas interconnected through a global network. The number of nodes and high-level policies was progressively incremented by both replicating the basic area and introducing minor adaptations.

Fig. 6 also shows some details about the basic building block structure, that is a simplification of a real use case. A and B are two adjacent rooms and B includes a cabinet CAB. Access to the A and B areas is controlled through specific credentials (i.e. physical keys, badges, pin codes, and so on). Room A hosts some general-purpose user workstations ($H_i, i = 1, 2, ...6$), while a special node (H_a) is reserved to administrators for network management activities. Services offered by each H_i are *login* (accessible either physically or remotely) and *backup*, while H_a provides an additional (reserved) functionality for remote management of workstations.

Nodes in room B are machines for corporate-level services (i.e., web pages S_w, e-mail S_m, database S_{db} and code repository S_c). Several service functions can be invoked remotely and access to them is guarded by firewall FW. Actually, FW is configured so as to enable accesses to S_{db} and S_c from H_a only. Moreover, critical servers are physically confined in cabinet CAB. O is the physical environment external to the system, which also includes (part of) the

system					*automaton*		*measurements*	
rep	rooms	devices	$\lvert$Actions$\rvert$	$\lvert$C$\rvert$	states	transitions	functions (s)	ref. (100%) (s)
1	4	11	51	42	7	126	0.01	0.01
2	7	22	101	83	12	448	0.04	0.02
3	10	33	151	124	19	1260	0.26	0.04
4	13	44	201	165	30	3248	0.93	0.08
5	16	55	251	206	49	7980	2.94	0.14
6	19	66	301	247	84	18984	6.95	0.19
7	22	77	351	288	151	44100	14.06	0.32
8	25	88	401	329	282	100576	26.38	0.42
9	28	99	451	370	541	226044	57.01	0.52
10	31	110	501	411	1056	502040	94.80	0.77

Table 1: Performance measurements

global interconnection network.

At the beginning any user is assumed to be in O, that is outside the system. In this example we consider two different kinds of users: an administrator *adm* and a generic employee *empl*, characterized by different permissions. Both *adm* and *empl* can access S_w and S_m, but only the administrator has total control over S_c and S_{db}. In practice, 50 possible actions are defined for each basic office area, and the administrator should be able to perform each one of them, while a smaller set is available to normal users ($S^-_{empl} \neq \emptyset$).

6.2. Performance evaluation

Though the adoption of our methodology relies on all blocks in Figs. 2 and 3, in this paper we focus on the *Functions computation* and *Refinement/fixing* elements. Indeed, the experimental evaluation of the time necessary for policy verification showed that, even though it linearly depends on the number of policies, it is almost negligible. As an example, in the worst case, the verification of five hundred policies has proven to take less than 1 ms. The interested reader can find more details about the *Automaton generator* block and its implementation in [6].

Moreover, when measuring the time needed to carry out the refinement process, performance figures for *empl* and *adm* were very similar. Indeed, the complexity of the refinement process performed by the "Z3" solver depends on the number of formulas included in equation (15), which, in turn, corresponds to the number of policies defined for the user (cardinality of S_u+ and S^-_u). In our case, it is the same for both users *empl* and *adm*. For these reasons, and to stress the refinement module capabilities, we reported the performance measures for user *empl* also varying the actual number of policies (S^+_{empl}, S^-_{empl}) at each test. The ratio $(|S^+|+|S^-|)/|Actions|$ was changed by defining four policy sets involving 25%, 50%, 75%, and 100% of possible actions respectively.

All tests were performed on a machine equipped with an Intel i7-67000 processor running at 3.40 GHz, 16 GB RAM, and the MS-Windows 10 operating system. Several experiments were carried out to evaluate the *Functions computation* and *Refinement* modules, with progressively larger target systems and policy sets, based on the composition of the basic elements discussed above.

A summary of the obtained results is reported in Tab. 1. The first column of the table shows the number of replicas of the basic blocks. The next four columns respectively contain the number of rooms, devices, total actions and credentials for the analyzed scenarios, followed by the total number of states and transitions for both A^r and A^L. Finally, the two rightmost columns report the total time (in seconds) necessary for the computation of functions $F(\alpha)$ and for performing the refinement process with the set of policy covering 100% of actions.

Similar data are also shown in Figs. 7 and 8, where the computation time, measured varying the ratio $(|S^+| + |S^-|)/|Actions|$ between 25% and 100%, is plotted versus the number of actions.

It is worth noting that, despite the time for computing the enabling functions grows non-linearly given the size of the test case, the feasibility of the approach is confirmed as even with a rather complex scenario including 500 actions, the prototype implementation is able to perform its task in less than 3 minutes on the test machine. In addition,

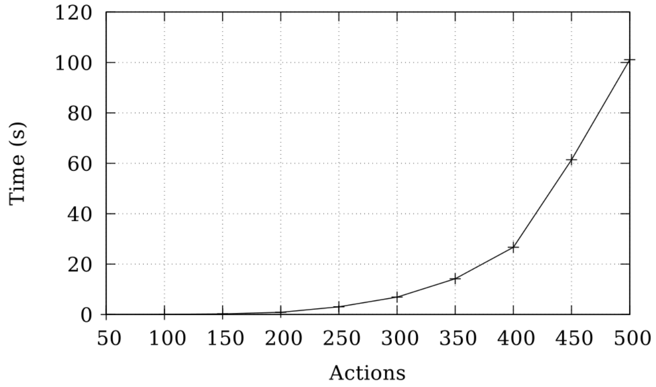


Figure 7: Performance of the *Functions computation* module.

the *Refinement* procedure is much faster (by two orders of magnitudes) and takes less than one second for the largest test case and the full set of policies.

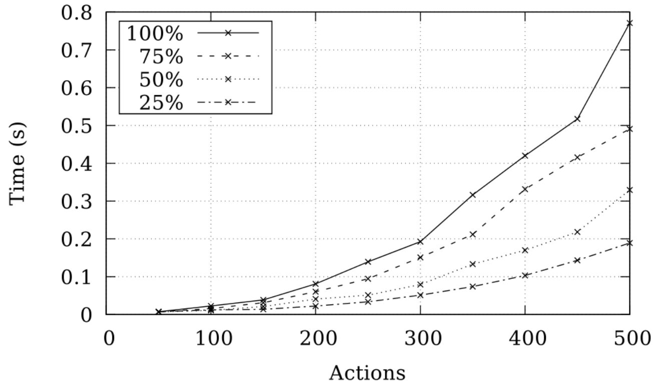


Figure 8: Performance of the *Refinement* module.

Note that the *Functions computation* is carried out only once for the entire system and is not dependent on the number of users. Conversely, the *Refinement* procedure has to be performed for each user. However, the result of the refinement procedure for one user (i.e., the credential assignment computed for that user) does not affect the refinement procedure for a different user in any way. This means that the processing time for refinement is linearly dependent on the number of users.

6.3. Worst case performance

We also checked the *Functions computation* module performance in the worst case scenario as discussed in Section 5.2. In particular, several automata were generated with increasing number of actions and a structure able to maximize the number of paths computed for a target event e, which is enabled in the final state of each automaton.

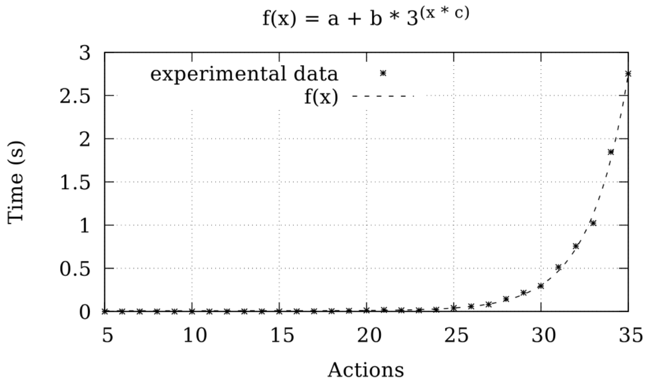


Figure 9: Performance of the *Functions computation* module in the worst case scenario.

Experimental results are shown in Fig. 9, where the number of actions x varies in the range 5 to 35. The curve $f(x) = a + b * 3^{(x*c)}$, which best fits the experimental data, is also plotted in the figure, with $a = 9015.639, b = 0.470$ and $c = 0.405$. The measured values are in good agreement with the worst-case asymptotic behavior identified in Section 5.2.

7. Related Works

Several works can be found in the literature concerning policy analysis, refinement and verification. Our contribution mainly focuses on the last two areas.

7.1. Policy Analysis

The main contributions in the area of policy analysis mostly deal with *anomaly analysis* and *policy evaluation.*

Anomaly analysis[3] looks for incorrect policy specifications that administrators may introduce in the network. It includes checks for potential errors, conflicts and sub-optimizations affecting either a single policy or a set of security policies [3]. Several works have been proposed to

[3] Sometimes anomaly analysis is also referred to as either conflict analysis or policy validation.

perform anomaly analysis for access control policies, that leverage different techniques such as model checking [11], binary decision diagrams (BDDs) [12], graph theory [13], DFSA [14], First Order Logic (FOL) [15] and geometrical model [16].

Policy evaluation, instead, checks whether a request is satisfied/unsatisfied by a set of policies [17]. A typical example of policy evaluation is the *change-impact analysis*, used to verify the effective impact of inserting, removing and/or modifying a policy. This means comparing original and modified policies. Relevant works on change-impact analysis are [18] and [19], that use BDDs to represent access control policies and perform the analysis for their modifications.

7.2. Policy Verification

Besides systematic literature reviews [20, 21], the number of papers dealing with the verification of access control policies is not large, especially when it is compared to the amount of literature dealing with policy analysis. Notable works in this area are [22, 23, 24, 25, 26].

In [22, 23], authors propose the automatic verification of access control policies against a set of properties. Verification is carried out by translating properties into a boolean satisfiability problem to be managed by means of a SAT solver. Differently from our approach, [22] and [23] model the target system in terms of access control policies and check if properties are violated by policies. Our solution, instead, builds on a fine-grained model of the real system where policies have to be verified. The SAT solver is used for refinement and fixing. In [26] authors propose a complete methodology based on BDD for the specification, verification and enforcement of access control policies. Differently from our approach, authors do not propose any methodology to fix the identified anomalies. Other tools such as NP-View and PolicyGlobe, respectively described in [24] and [25], allow for checking global policy implementations by processing information about the network topology and configuration. Unfortunately, this is limited to systems where policy enforcement mechanisms are assumed to be available. This reduces the tools usefulness as, for instance, they can produce meaningful results only for systems consisting of nodes running SE-Linux.

Note that, at the moment our approach does not allow to model and check the correct implementation of policies that require counting the number of times a certain action is performed by one or multiple users (similar, e.g., to history-based policies [27, 28, 29]). Indeed, our approach is meant to be used for a priori system verification/configuration and not as a runtime tool for access control. As such, while the model is able to capture action dependencies determined by system characteristics and configuration, it does not describe either multiple user interactions nor the flowing of time.

7.3. Policy Refinement

Although refinement is one of most ambitious goals in policy-based management when access control is considered, relatively few works have been published in the last decade, with particular focus in this area. Refinement of access control policies was originally introduced in [30, 31, 32], but little or no development has appeared since then.

More recently, with the advent of both the software defined networking (SDN) and network function virtualization (NFV) paradigms, interest in policy refinement techniques has revived [33, 34, 35]. It is worth noting that while other proposals aim at defining high-level specification languages, which are able to describe access control policies and translate them into low-level configuration commands, our solution exploits refinement techniques to search all sets of credentials that satisfy the policies automatically. If no solution is found, changes in either the policy specification or the system structure become the only viable alternatives.

8. Conclusions and Future work

In this paper, a comprehensive approach has been presented to solve some common problems faced by network administrators when dealing with access control through policy-based management. The proposed approach can be exploited to:

1. find a user credential assignment that correctly enforces a set of access control policies (*policy refinement*);
2. complete a partial user credential assignment so that results correctly enforce the specified policies (*constrained policy refinement*);
3. check whether a specific user credential assignment makes the system behavior match the policy specification (*policy verification*). If this is not the case, the approach can be used to introduce automatic changes to fix the highlighted discrepancies between the actual and desired behavior (i.e. *anomaly fixing*).

Our technique is based on the definition of models for both the access control policies and the target networked system (in terms of topology, s/w and h/w resources and how they are accessed by users). The system model is then processed to deduce what sequences of actions are needed to perform a specific operation. The resulting information is used to determine the minimal sets of credentials (if any) that shall be owned by users to perform all and only the actions assigned to them by the policies.

The feasibility of the approach has been verified by means of a prototype s/w tool, which is based on Java and the Z3 library, an off-the-shelf SMT solver. The tool performance was tested in a number of realistic network scenarios and its scalability also checked (the tool is able to provide solutions in a very short time even for networks consisting of several hundred nodes and target resources).

In the future, we plan to extend the expressiveness of the model to take into account further types of network devices, such as VPN gateways and intrusion detection systems (IDSs). Furthermore, we intend to apply this methodology so as to consider not only user credential assignments but also other aspects of network configuration (e.g., firewall rules). Finally, it is our intention to perform some empirical assessment to evaluate the ability of the model to capture the nature of real networks and the effectiveness of the approach in helping administrators managing access control in their networks.

References

[1] P. Samarati, S. C. de Vimercati, Access Control: Policies, Models, and Mechanisms, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, Ch. 3, pp. 137–196. doi:10.1007/3-540-45608-2_3.
URL https://doi.org/10.1007/3-540-45608-2_3

[2] D. Verma, Simplifying network administration using policy-based management, IEEE Network 16 (2) (2002) 20–26. doi:10.1109/65.993219.
URL http://ieeexplore.ieee.org/document/993219/

[3] F. Valenza, C. Basile, D. Canavese, A. Lioy, Classification and Analysis of Communication Protection Policy Anomalies, IEEE/ACM Transactions on Networking (2017) 1–14doi:10.1109/TNET.2017.2708096.
URL http://ieeexplore.ieee.org/document/7967691/

[4] J. D. Moffett, M. S. Sloman, Policy hierarchies for distributed systems management, IEEE Journal on Selected Areas in Communications 11 (9) (1993) 1404–1414. doi:10.1109/49.257932.

[5] Verizon, Data Breach Investigations Report (2017).

[6] M. Cheminod, L. Durante, L. Seno, A. Valenzano, Semiautomated Verification of Access Control Implementation in Industrial Networked Systems, IEEE Transactions on Industrial Informatics 11 (6) (2015) 1388–1399. doi:10.1109/TII.2015.2489181.
URL http://ieeexplore.ieee.org/document/7295628/

[7] M. Cheminod, L. Durante, L. Seno, F. Valenza, A. Valenzano, Automated fixing of access policy implementation in Industrial Networked Systems, in: 2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS), IEEE, 2017, pp. 1–9. doi:10.1109/WFCS.2017.7991947.
URL http://ieeexplore.ieee.org/document/7991947/

[8] C. G. Cassandras, S. Lafortune, Introduction to Discrete Event

Systems, Second Edition, Springer Science+Business Media, LLC, 2008.

[9] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.

[10] G. Birkhoff, Lattice theory, Vol. 25, American Mathematical Soc., 1940.

[11] K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, S. Chapin, Automatic error finding in access-control policies, in: Proceedings of the 18th ACM conference on Computer and communications security - CCS '11, ACM Press, 2011, p. 163. doi:10.1145/2046707.2046727.

[12] H. Hu, G.-J. Ahn, K. Kulkarni, Discovery and Resolution of Anomalies in Web Access Control Policies, IEEE Transactions on Dependable and Secure Computing 10 (6) (2013) 341–354. doi:10.1109/TDSC.2013.18.

[13] M. Koch, L. V. Mancini, F. Parisi-Presicce, Conflict Detection and Resolution in Access Control Policy Specifications, in: Proc. of the 5th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS), Vol. 2303 of LNCS, 2002, pp. 223–238. doi:10.1007/3-540-45931-6_16.
URL https://doi.org/10.1007/3-540-45931-6_16

[14] F. B. Schneider, Enforceable security policies, ACM Trans. Inf. Syst. Secur. 3 (1) (2000) 30–50. doi:10.1145/353323.353382.

[15] M. Cheminod, L. Durante, F. Valenza, A. Valenzano, Toward attribute-based access control policy in industrial networked systems, in: Proceedings of the 14th IEEE International Workshop on Factory Communication Systems (WFCS 18), 2018, pp. 1–9. doi:10.1109/WFCS.2018.8402339.

[16] C. Basile, D. Canavese, C. Pitscheider, A. Lioy, F. Valenza, Assessing network authorization policies via reachability analysis, Computers and Electrical Engineering 64 (2017) 110 – 131. doi:10.1016/j.compeleceng.2017.02.019.

[17] A. X. Liu, F. Chen, J. Hwang, T. Xie, Xengine: A fast and scalable xacml policy evaluation engine, SIGMETRICS Perform. Eval. Rev. 36 (1) (2008) 265–276. doi:10.1145/1384529.1375488.

[18] K. Fisler, S. Krishnamurthi, L. Meyerovich, M. Tschantz, Verification and change-impact analysis of access-control policies, in: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., IEEE, 2005, pp. 196–205. doi:10.1109/ICSE.2005.1553562.
URL http://ieeexplore.ieee.org/document/1553562/

[19] A. X. Liu, F. Chen, J. Hwang, T. Xie, Designing Fast and Scalable XACML Policy Evaluation Engines, IEEE Transactions on Computers 60 (12) (2011) 1802–1817. doi:10.1109/TC.2010.274.

[20] R. A. Shaikh, K. Adi, L. Logrippo, A Data Classification Method for Inconsistency and Incompleteness Detection in Access Control Policy Sets, International Journal of Information Security 16 (1) (2017) 91–113. doi:10.1007/s10207-016-0317-1.
URL http://link.springer.com/10.1007/s10207-016-0317-1

[21] M. Aqib, R. A. Shaikh, Analysis and Comparison of Access Control Policies Validation Mechanisms, I.J. Computer Network and Information Security 1 (1) (2015) 54–69. doi:10.5815/ijcnis.2015.01.08.

[22] G. Hughes, T. Bultan, Automated verification of access control policies using a SAT solver, International Journal on Software Tools for Technology Transfer 10 (6) (2008) 503–520. doi:10.1007/s10009-008-0087-9.
URL http://link.springer.com/10.1007/s10009-008-0087-9

[23] P. Bera, S. K. Ghosh, P. Dasgupta, Policy Based Security Analysis in Enterprise Networks: A Formal Approach, IEEE Transactions on Network and Service Management 7 (4) (2010) 231–243. doi:10.1109/TNSM.2010.1012.0365.
URL http://ieeexplore.ieee.org/document/5668979/

[24] D. M. Nicol, W. H. Sanders, S. Singh, M. Seri, Usable Global Network Access Policy for Process Control Systems, IEEE Security Privacy 6 (6) (2008) 30–36. doi:10.1109/MSP.2008.159.

[25] H. Okhravi, R. H. Kagin, D. M. Nicol, PolicyGlobe: A Framework for Integrating Network and Operating System Security Policies, in: Proc. of the 2nd ACM Wksp. on Assurable and usable security configuration (SafeConfig), 2009, pp. 53–62. doi:10.1145/1655062.1655074.

[26] A. Cau, H. Janicke, B. Moszkowski, Verification and enforcement of access control policies, Formal Methods in System Design 43 (3) (2013) 450–492. doi:10.1007/s10703-013-0187-3.

[27] M. Abadi, C. Fournet, Access control based on execution history, in: Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA, 2003, pp. 107–121.

[28] P. W. L. Fong, Access control by tracking shallow execution history, in: IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004, 2004, pp. 43–55. doi:10.1109/SECPRI.2004.1301314.

[29] J. Lobo, J. Ma, A. Russo, E. Lupu, S. Calo, M. Sloman, Refinement of History-Based Policies, Springer Berlin Heidelberg, 2011, Ch. 15, pp. 280–299. doi:10.1007/978-3-642-20832-4_18.

[30] A. Bandara, E. Lupu, J. Moffett, A. Russo, A goal-based approach to policy refinement, in: Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004., IEEE, 2004, pp. 229–239. doi:10.1109/POLICY.2004.1309175.

URL http://ieeexplore.ieee.org/document/1309175/

[31] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, G. Pavlou, A methodological approach toward the refinement problem in policy-based management systems, IEEE Communications Magazine 44 (10) (2006) 60–68. doi:10.1109/MCOM.2006.1710414.
URL http://ieeexplore.ieee.org/document/1710414/

[32] R. Craven, J. Lobo, E. Lupu, A. Russo, M. Sloman, Decomposition techniques for policy refinement, in: 2010 International Conference on Network and Service Management, IEEE, 2010, pp. 72–79. doi:10.1109/CNSM.2010.5691331.
URL http://ieeexplore.ieee.org/document/5691331/

[33] C. C. Machado, L. Z. Granville, A. Schaeffer-Filho, J. A. Wickboldt, Towards SLA Policy Refinement for QoS Management in Software-Defined Networking, in: 2014 IEEE 28th International Conference on Advanced Information Networking and Applications, IEEE, 2014, pp. 397–404. doi:10.1109/AINA.2014.148.
URL http://ieeexplore.ieee.org/document/6838692/

[34] C. Basile, A. Lioy, C. Pitscheider, F. Valenza, M. Vallini, A novel approach for integrating security policy enforcement with dynamic network virtualization, in: Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft), IEEE, 2015, pp. 1–5. doi:10.1109/NETSOFT.2015.7116152.
URL http://ieeexplore.ieee.org/document/7116152/

[35] F. Valenza, T. Su, S. Spinoso, A. Lioy, R. Sisto, M. Vallini, A formal approach for network security policy validation, Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA) 8 (1) (2017) 79–100.

Manuel Cheminodreceived the M.S. in 2005 and the Ph.D in Computer Engineering in 2010, both from the Politecnico di Torino, Italy. He his a Researcher with the Institute of Electronics, Computer, and Telecommunication Engineering of the National Resarch Council of Italy, in Torino. His research interests include formal methods for the verification of security properties in industrial systems and the exploitation of SDN/NFV paradigms to improve the security level of the same critical systems.

Luca Durante is Senior Researcher at the Institute of Electronics, Information Engineering and Telecommunications of the National Research Council of Italy (CNR-IEIIT) in Torino, Italy. He graduated in Electronic Engineering in 1992, and received the PhD degree in Computer Engineering in 1996, both from the Politecnico di Torino. He has co-authored about 60 scientific journal, conference papers and technical reports in the area of industrial communication protocols and formal techniques for distributed systems. He also serves as a technical referee for several international conferences and journals, and he has been the scientific coordinator of the CNR team in European Projects. He served as Program Co-Chairman for the 13th (WFCS 2017) edition of the IEEE Workshop on Factory Communication Systems. Currently his research interests include formal verification of cryptographic protocols, source-level model checking of software and network vulnerability and dependability analysis.

Lucia Seno (M'15) received the B.S. and M.S. degrees in Automation Engineering and the Ph.D. degree in Information Engineering from the University of Padova, Padova, Italy, in 2004, 2007, and 2011, respectively.

Since then, she has been with the Institute of Electronics, Computer, and Telecommunication Engineering of the National Research Council of Italy in Padova, and, subsequently, in Torino, Italy, where she is currently a Researcher. Her research interests include industrial communication systems and technologies, wireless networks and wireless sensor networks for real-time and safety-critical control applications, formal verification of system security and communication protocols, and model-checking.

Fulvio Valenza received the M.Sc. (summa cum laude) in 2013 and the Ph.D. in Computer Engineering in 2017 from the Politecnico di Torino, Torino, Italy. His research activity focus on network security policies. Currently he is a Researcher at the CNR-IEIIT Torino, Italy, where he works on orchestration and management of network security functions in the context of SDN/NFV-based networks and industrial systems.

Adriano Valenzano (SM'09) received the Laurea degree in electronic engineering from Politecnico di Torino, Torino, Italy, in 1980. He is Director of Research with the National Research Council of Italy (CNR). He is currently with the Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Torino, Italy, where he is responsible for research concerning distributed computer systems, local area networks, and communication protocols. He has coauthored approximately 200 refereed journal and conference papers in the area of computer engineering. Dr. Valenzano is the recipient of the 2013 IEEE IES and ABB Lifetime Contribution to Factory Automation Award. He also received, as a coauthor, the Best Paper Award presented at the 5th, 8th and 13th IEEE Workshops on Factory Communication Systems (WFCS 2004, WFCS 2010 and WFCS 2017). He has served as a technical referee for several international journals and conferences, also taking part in the program committees of international events of primary importance. Since 2007, he has been serving as an Associate Editor for the IEEE Transactions on Industrial Informatics.