Refactoring Multi-Layered Access Control Policies Through (De)Composition

Matteo Maria Casalino*†
*SAP Research Sophia-Antipolis
805 Avenue Dr M. Donat
06250 Mougins, France

Romuald Thion‡
†Université de Lyon
Université Claude Bernard Lyon 1
LIRIS CNRS UMR5205, France

Abstract—Policy-based access control is a well-established paradigm for securing layered IT systems. Access control policies, however, often do not focus on dedicated architecture layers, but increasingly employ concepts of multiple layers. Web application servers, for instance, typically support request filtering on the basis of network addresses. The resulting flexibility comes with increased management complexity and the risk of security-relevant misconfiguration when looking at the various policies in isolation. We therefore propose a flexible access control framework able to provide a comprehensive view of the global access control policy implemented in a given system. The focus of this paper is to lay down the theoretical foundations of this framework that allows (i) to describe authorization policies from different architecture layers, (ii) to capture the semantics of dependencies between layers in order to create a composed view of the global policy, and (iii) to decompose the global policy again into a collection of simpler ones by means of algebraic techniques inspired from database normalization theory.

I. INTRODUCTION

Access control is pervasive within the different layers of IT infrastructures. For instance, network filtering and application-layer authorization policies are different forms of access control that typically cooperate in real scenarios. The access control process is distributed across several IT components, each one potentially operating on different architecture layers and residing on different hosts. For instance, a classical network firewall is able to take allow/deny decisions for network requests, having parameters such as IP addresses and TCP/UDP ports. A Web server instead handles a different kind of requests that rather belong to the application ISO/OSI layer, e.g., having parameters such as the requested URL.

However, the separation between network and application layers is typically not as neat. For example, many common services (e.g., the Apache Web server, the MySQL database server or the anti-spam features of the sendmail mail server), perform access control based not only on application-specific parameters (e.g. respectively, URLs, tables, mail addresses), but can overlap with the lower layer (e.g., by filtering on the IP address of the requester). Conversely, modern firewalls are more and more capable of inspecting application layer fields.

Such inter-layer relationships can increase system security, in practice, however, more complexity often leads to misconfiguration, compromising systems security [1]. Accordingly, the problem of distributed inter-dependent policies is identified as one of the security automation research challenges in [2].

In this paper we focus on inter-layer policy refactoring, i.e., the task of finding the least permissive rewriting of a collection of policies that belong to different layers such that the global composed policy remains identical. Policy refactoring is a means (i) to enforce the least privilege principle in multi-layered policy-based access control systems, (ii) to reduce management overhead by simplifying local policies and (iii) to adapt to changing security capabilities of single components. Essential to solve this problem is to model the structure of requests at each layer, to keep track of inter-layer relationships and to support the notion of inter-layer policy (de)composition, by capturing how decisions taken within different layers influence each other.

A. Motivating Scenario

In order to illustrate refactoring, we consider the following hypothetical scenario. Suppose a Web server featuring the capability to filter on the clients’ IP addresses that is harbored in a network whose access from the Internet is mediated by a firewall. As the policies of the two devices are written independently, some unnecessary privileges may be granted by either of them. For instance, the firewall policy may be granting access to a larger portion of clients than actually allowed by the Web server policy or vice versa. Refactoring exploits the knowledge of the inter-layer interactions to reduce such privileges to the minimum, by preserving the composite policy. As such, unauthorized access attempts are blocked as soon as possible, according to the least privilege principle.

Suppose now that we are interested in replacing the Web server policy by a simpler one that does not discriminate clients’ IP addresses while keeping the global composite policy unaltered. We would need to transfer part of the Web server policy to the lower layer, such that all the network-layer filtering is then performed by the firewall only. Whether such a decomposition is possible at all, depends both on the internal structure of the original policies and on the access control capabilities of the devices. For instance, if the Web server policy prevents a client from accessing only some specific URLs, the firewall cannot enforce it on its own unless it is capable of HTTP header inspection. In this case refactoring consists in first determining if the new desired access control
layers layout can enforce the global policy and then finding how the original policies are to be rewritten.

B. Contribution

Our contributions are as follows. In Section II we define a model that captures the access control behavior of a collection of policy decision points that cooperate on different layers of the same IT infrastructure. In Section III we define composition of inter-dependent policies as the operation that, given a pair of access control decision functions, produces a composite decision function, decomposition being its inverse. In Section IV we provide a compact representation of our model based on which we devise algorithms that compute policy (de)composition and that we formally prove correct. In Section V we identify a criterion inspired from database normalization theory, more specifically multivalued dependencies [3], that characterizes when policies can be decomposed. Moreover, we show how the combination of policy composition and decomposition can solve the refactoring problem.

II. ACCESS CONTROL LAYERS

In this section we lay the foundations of a model that captures the access control policy implemented by the collection of policy decision points that operate at different layers within the same IT infrastructure (e.g., firewalls, application servers, Web servers, database servers, etc.). In particular, we aim at characterizing each such layer in terms of its access control capabilities and its interface with the other layers.

We begin with some preliminaries about access control models that are necessary to provide a more precise definition of our notion of access control layer.

We rely on a classic and general description of access control systems, where a logical subsystem (usually called policy decision point) associates, for a given policy, a unique decision to any possible access control request [4], [5].

Once we come to reason about the composition of layers, we need to consider relations between the different types of requests they handle. For instance, the IP and port destination fields of the requests handled by a firewall are related to the IP addresses and ports of available services (e.g., Web and application servers). Intuitively, a particular firewall, depending on its policy, either enables requests to be further processed by other policy decision points or block them right away. Keeping track of these relationships allows to determine how decisions taken by one layer’s policy decision points influence the ones taken in other layers. In order to formalize the above concepts we start from access control request fields and types.

Definition 1 (Request Field and Field’s Domain): Let \( \mathcal{F} \) be the finite set of all fields totally ordered by the relation \( \preceq \subseteq \mathcal{F} \times \mathcal{F} \). Each field \( f \in \mathcal{F} \) has a corresponding domain, written \( \text{dom}(f) \), that is the set of all possible values \( f \) can take in a request.

Table I presents some example request fields which we will refer to throughout the paper, together with their respective domains. The purpose of the special field \( \mathcal{I} \) is to represent a fictitious singleton domain. A given set of fields characterizes a particular type of access control requests, as stated in the following definition.

Definition 2 (Request Type and Request Space): A request type is a finite subset of request fields \( F \in 2^{\mathcal{F}} \). The request space \( \Omega(F) \) associated with a request type \( F = \{f_i\}_{i=1}^n \) characterizes all the requests existing over \( F \). It is the Cartesian product of the domains of the fields in \( F \), taken according to \( \preceq \): \( \Omega(F) = \text{dom}(f_1) \times \ldots \times \text{dom}(f_n) \), \( f_i \preceq f_j \) for \( i \leq j \).

Example request types are \( F_{fw} = \{i_1,i_4,i_5,p_s,p_d,T_p\} \), which characterizes requests handled by firewalls, or \( F_{uw} = \{i_1,H,U\} \) for application-layer requests in the context of a Web server capable of filtering on the clients’ IP address. Other combinations are possible too: for instance \( F_{fw} \cup \{H,U\} \) models the request type of a firewall that can inspect parts of the HTTP header.

The definition of access control request follows from the definitions of request type and request space. We define two operations on requests: concatenation and projection.

Definition 3 (Access Control Request): An access control request of type \( F = \{f_i\}_{i=1}^n \) is an element of the request space \( \Omega(F) \). The requests belonging to \( \Omega(F) \) are therefore all the possible sequences \( \langle v_1,\ldots,v_n \rangle \) with \( v_i \in \text{dom}(f_i) \) and \( f_i \preceq f_j \) for \( i \leq j \). The \( i \)th coordinate of \( q \in \Omega(F) \) is written \( q(i) = v_i \).

Given the requests \( q_1 \in \Omega(F_1) \) and \( q_2 \in \Omega(F_2) \), such that \( F_1 \cap F_2 = \emptyset \), their concatenation is the request \( q_1 + q_2 \in \Omega(F_1 \cup F_2) \) (also denoted \( q_1 \bowtie q_2 \)) such that, \( \forall f \in F_1 \cup F_2, (q_1 + q_2)(f) = q_1(f) \) if \( f \in F_1 \) (with \( i \in \{1,2\} \)). Given the request \( q \in \Omega(F) \), the projection of \( q \) on some request type \( P \subseteq F \) is denoted by \( q|_P \) and is simply the restriction of the sequence \( q \) to the fields in \( P = \{p_1,\ldots,p_n\} \), it is defined by \( q|_P = \langle q(p_1) \ldots q(p_n) \rangle \).

We are now ready to provide a formal description of access control layers. An access control layer represents a collection of policy decision points that are all capable of processing access control requests of the same type. It conveys essentially the following three pieces of information:

- the type of access control requests that are in the layer’s scope, i.e., those which the layer’s decision points are deputed to express a decision for;
- how the request type handled locally relates to that of requests handled within other layers;
- which decision is taken, for every request, by any decision point in the layer according to its policy.

<table>
<thead>
<tr>
<th>( f )</th>
<th>Meaning</th>
<th>( \text{dom}(f) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i_1 )</td>
<td>IP source address</td>
<td>Integers range ([0,2^{32}-1])</td>
</tr>
<tr>
<td>( i_4 )</td>
<td>IP destination address</td>
<td>Integers range ([0,2^{32}-1])</td>
</tr>
<tr>
<td>( p_s )</td>
<td>Port source</td>
<td>Integers range ([0,2^{16}-1])</td>
</tr>
<tr>
<td>( p_d )</td>
<td>Port destination</td>
<td>Integers range ([0,2^{16}-1])</td>
</tr>
<tr>
<td>( T_p )</td>
<td>Transport Protocol</td>
<td>Enumeration {TCP,UDP}</td>
</tr>
<tr>
<td>( H )</td>
<td>HTTP Host Header</td>
<td>Dot-separated strings</td>
</tr>
<tr>
<td>( U )</td>
<td>URL of HTTP Requests</td>
<td>Strings complying with rfc1738</td>
</tr>
</tbody>
</table>

\[ \mathcal{I} \] Singleton field \{\} |
**Definition 4 (Access Control Layer):** An Access Control Layer (ACL) is a triple \( \{F, C, \delta\} \) where:

(i) \( F \in 2^P \) is the layer request type;
(ii) \( C \in 2^P \setminus \{\emptyset\} \), s.t. \( C \cap F = \emptyset \), is the layer coupling type;
(iii) \( \delta : \Omega(C) \to (\Omega(F) \to D) \) is the access decision function with \( D \) the set of decisions.

The coupling type \( C \) characterizes the interface with the layers an ACL can be composed with. Every value in the coupling space \( c \in \Omega(C) \) identifies a policy decision point within the layer; the function \( \delta(c) : \Omega(F) \to D \), which maps every possible request \( q \in \Omega(F) \) to a unique decision, represents its policy.

Suppose a scenario where a firewall \( F \) protects the access to the IP network 1.1.1.0/24 in which resides a Web server \( W \) running on the address 1.1.1.1 and TCP port 80. Other services may possibly be running in the same IP network.

Depending on the access control capabilities of the policy decision points we consider, several configurations of access control layers are possible. Assume that \( W \) supports IP source address filtering. We then have the access control layers \( L_a = \{1.1.1.1, P_a\} \), \( \{\sharp\} \), \( \delta_a \) and \( L_b = \{1.1.1.1, H, U\}, \{I_d, P_d\} \), \( \delta_b \), where a possible instance of \( \delta_a \) (resp. \( \delta_b \)) is reported in Table II (resp. Table III).

Each row in the tables maps all the requests matching the wildcards to the decision reported in the last column. The set difference symbol “\( - \)” represents exceptions (e.g., \( *.*.*.* \setminus 3.3.3.3.3 \) means every IP address except the subnet 3.3.3.0/24). The ellipsis “\( \ldots \)” represents all the requests that do not match any other rule. Note that the graph of decision functions is 3.3.3.

The fictitious coupling type \( \{\sharp\} \) of \( L_a \) does not allow any choice in the coupling space, hence we cannot distinguish different policy decision points in this layer. The reason why we do not consider multiple firewalls is that the interest in analyzing distributed network filtering typically concerns inlaayer dependencies, whereas we only want to model (and reason about) inter-layer dependencies. We believe that intra-layer reasoning is an orthogonal problem that, as argued in Section VI, has already been addressed in related work.

Note that more layers could be added on top of \( L_b \). For instance, Web applications running on certain virtual hosts within the Web server and performing access control based on application-specific fields, such as users, roles, actions and resources. However, for the sake of conciseness, in this paper we limit the scope of our analysis to these two layers.

### III. Composition and Decomposition

In this section we define composition as a binary operation between ACLs. In order to compose access control layers, we first need a way to combine the decisions yielded by their respective policies. We thus briefly discuss how this is achieved by equipping the set of decisions with a suitable algebraic structure named decision space.

The standard decision space is \( B = \{0, 1\} \) where 0 stands for prohibition and 1 for authorization. Note that in this case the decision function \( \delta : \Omega(F) \to B \) simply tests whether some request \( q \in \Omega(F) \) is a member of a set \( Q^\text{Auth} \subseteq \Omega(F) \) of authorized queries. Many existing languages (e.g., [5], [6], [7]) assume that the decision space is larger than \( B \) to include for instance undefined decision (\( \bot \)) or conflicting decision (\( \top \)) to cope with modular specification of authorization policies. Here, the decision space is equipped with operators mimicking classical boolean disjunction and conjunction, which we denote \( \sqcup \) and \( \sqcap \).

**Definition 5 (Decision Space):** A decision space is a bounded distributive lattice \( \langle D, \sqcup, \sqcap, \bot, \top \rangle \), where \( D \) is a non empty finite set of decisions and \( \bot, \top \) are respectively the least upper bound and greatest lower bound operators on \( D \). The lattice top and bottom elements are denoted respectively \( 1_D \) and \( 0_D \).

Where no ambiguity arises we identify a decision space with its underlying set \( D \). Figure 1a shows the Hasse diagrams of the boolean decision space \( B \) and its extension to undefined decisions \( B \bot, \) which we will use in our examples throughout the rest of the paper.
The key to ACL composition is the overlap of request types, because it implies interdependency between decisions taken by different decision functions. Let \( L_1 = \langle F_1, C_1, \delta_1 \rangle \) and \( L_2 = \langle F_2, C_2, \delta_2 \rangle \) be two ACLs that act in composition, e.g., suppose \( L_2 \) is over \( L_1 \) in the network stack. Then, every lower layer request shall match to some upper layer policy decision point, hence the upper layer coupling type \( C_2 \) is included in the lower layer request type \( F_1 \). Furthermore, it may be the case that the two layers’ request types have some fields in common, that is \( F_1 \) and \( F_2 \) have a non empty intersection. Figure 1b depicts this situation, where the double hatched areas identify the overlap between layers.

The union of the request types of \( L_1 \) and \( L_2 \) can then be thought as the request type of a new ACL \( L \), that we are going to define as their composition. The decision function of \( L \) needs to depend both on \( \delta_1 \) and \( \delta_2 \). If we assumed a boolean decision space, then we would expect every lower layer request \( q_1 \) that agrees with an upper layer request \( q_u \) to yield a composite request that is allowed if and only if both \( q_1 \) and \( q_u \) are allowed. In the following definition we generalize this intuition by substituting the logic conjunction by the greatest lower bound operator \( \sqcap \).

**Definition 6 (ACL Composition):** Given the ACLs \( L_1 = \langle F_1, C_1, \delta_1 \rangle \) and \( L_2 = \langle F_2, C_2, \delta_2 \rangle \) such that \( C_2 \subseteq F_1 \) and \( C_1 \cap F_2 = \emptyset \), their composition is the ACL \( L_1 \otimes L_2 = \langle F_1 \cup F_2, C_1 \cup C_2, \delta \rangle \), where \( \delta \) is defined as follows:

\[
\begin{align*}
\delta &: \Omega(C_1 \cup F_1 \cup F_2) \rightarrow D \\
q &\mapsto \delta_1(q|_{C_1 \cup F_1}) \cap \delta_2(q|_{C_2 \cup F_2}).
\end{align*}
\]

As an example, we consider the composition \( L_c \) of ACLs \( L_a \) and \( L_b \) introduced in Section II. \( L_a \otimes L_b = L_c = \langle \{l, l_a, d_a, P_a, P_d, H, U\}, \{\sharp\}, \delta_c \rangle \), where the graph of \( \delta_c \) is represented in Table IV. For instance, the request \( 2.2.2.1, 1.1.1.1, 0, 80, \text{acme.com}, /\text{private}/* \rangle \) is authorized in \( L_c \) because \( L_a \) allows \( 2.2.2.1, 1.1.1.1, 0, 80 \) and \( L_b \) allows \( 2.2.2.1, \text{acme.com}, /\text{private}/* \rangle \). However, the decision related to the request \( 3.3.3.3, 1.1.1.20, 0, 80, \text{acme.com}, /\text{private}/* \rangle \) is \( \perp \) because it is unknown whether any Web server is listening on the IP address 1.1.1.20.

Given an ACL \( L = \langle F, C, \delta \rangle \), decomposition is the problem of finding ACLs \( L_1 = \langle F_1, C_1, \delta_1 \rangle \) and \( L_2 = \langle F_2, C_2, \delta_2 \rangle \) such that \( F_1 \otimes F_2 = F \). As Definition 6 defines the structure of the candidate decompositions, the problem amounts to compute \( \delta_1 \) and \( \delta_2 \) from \( \delta \). This means that, for every request \( q_1 \in \Omega(F_1 \cup C_1) \), we need the value of \( \delta_1(q_1) \) as a function of all the requests \( q \in \Omega(F) \) such that \( q|_{F_1 \cup C_1} = q_1 \).

In the case of a boolean decision space, the natural semantics we would like to assign to such an operation is that of projection. For instance, let \( \delta : \Omega(F) \rightarrow \mathbb{B} \). Its projection on \( P \subseteq F \) would be \( \pi_P(\delta) : \Omega(P) \rightarrow \mathbb{B} \) such that \( \pi_P(\delta)(q) = 1 \) if and only if \( \delta(q') = 1 \) for at least one \( q' \in \Omega(F) \). Hence, \( \pi_P(\delta) \) would map each request to the logic disjunction of all such \( q'(q') \). In the next definition we generalize to larger decision spaces by replacing disjunction with just upper bound \( \sqcup \).

**Definition 7 (Projection):** The projection of a decision function \( \delta : \Omega(F) \rightarrow D \) over the set of fields \( P \subseteq F \) is the decision function \( \pi_P(\delta) \) defined as follows:

\[
\pi_P(\delta) : \Omega(P) \rightarrow D \\
q \mapsto \bigcup_{x \in \Omega(F \setminus P)} \delta(q + x).
\]

As stated in the next proposition, decompositions obtained through projection are always the least permissive among those that preserve the composite decision function unchanged.

**Proposition 1 (Least Privilege Decomposition):** Let \( L = \langle F_1 \cup F_2, C_1, \delta \rangle \) such that \( L = \langle F_1, C_1, \delta_1 \rangle \otimes \langle F_2, C_2, \delta_2 \rangle \). Then, \( \forall q \in \Omega(F_1 \cup C_1), \pi_{F_1 \cup C_1}(\delta)(q) \leq \delta_i(q) \) for \( i \in \{1, 2\} \).

This result guarantees the least privilege principle for policy refactoring as stated in the introduction. We begin to see that the definitions of composition and decomposition can be suitable tools to tackle the refactoring problem.

IV. COMPACT REPRESENTATION

The request spaces we considered so far are, in general, infinite or very large in cardinality. This follows from the fact that each field can have either an infinite (e.g. URLs) or a very big (e.g. IP addresses) domain of values. In order to cope with this issue, in this section we introduce a finite and compact representation for generic request spaces and decision functions. We then show how (de)composition can be computed on instances of such a representation.

**Definition 8 (Field Descriptor):** Given a request field \( f \in F \), a field descriptor for \( f \) is a structure \( \langle \Phi_f, [f]_f \rangle \) where \( \Phi_f \) is a language that allows to describe sets of elements in the domain of \( f \) and \( [f]_f : \Phi_f \rightarrow 2^{\text{dom}(f)} \) is an interpretation function that maps every sentence of the language to its extension, i.e., the subset of the field’s domain it describes.
Furthermore, the descriptor language is closed under the intersection of sentences’ extensions, namely \( \forall \varphi_1, \varphi_2 \in \Phi_f, \exists \varphi_3 \in \Phi_f \) s.t. \( \{ \varphi_3 \} = \{ \varphi_1 \} \cap \{ \varphi_2 \} \). For every such combination we call \( \varphi_3 \) the conjunction of \( \varphi_1, \varphi_2 \), written \( \varphi_3 = \varphi_1 \land \varphi_2 \).

In Table V we provide an example field descriptor that allows to describe intervals of positive integers. A bounded interval is described by a pair of integers representing its minimum and maximum values; an unbounded one has its maximum value set to \( \infty \). The conjunction of two intervals is the (potentially empty) interval ranging from the maximum of their lower boundaries to the minimum of their upper ones.

This descriptor is already suitable for representing many of the fields introduced in Table I (i.e., \( l_1, l_2, p_s, p_d \)). For fields associated with large or infinite domains of strings (such as \( U \)) we could analogously define a field descriptor where the sentences in \( \Phi \) are arbitrary regular expressions, and their conjunction is the intersection of the corresponding automata.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varphi )</td>
<td>( 'n : ' m )</td>
</tr>
<tr>
<td>( m )</td>
<td>( n \in \mathbb{N} )</td>
</tr>
<tr>
<td>( n )</td>
<td>Integer</td>
</tr>
</tbody>
</table>

\[ \bigwedge \left( \left[ 'n : ' m \right], \left[ 'm : ' n 'm \right], \left[ 'm : ' n 'm \right] \right) \implies \varphi_1 \land \varphi_2 = [20 : 100] \]

\[ \left[ 'i \cdot ' n \right] \implies \{ \varphi_1 = [0,1,\ldots,100] \} \]

\[ \{ i | n \leq i < m \} \]

\[ \{ \varphi_1 \land \varphi_2 = [20,21,\ldots,100] \} \]

\[ \{ \varphi_1 \land \varphi_2 = \emptyset \} \]

Algorithm 1 ACL Composition with DFD

**Input:** \( L_1 = (F_1, C_1, \Delta_1), L_2 = (F_2, C_2, \Delta_2) \) are ACLs.

**Output:** \( L_1 \otimes L_2 \)

1. \( W \leftarrow (F_1 \land F_2) \land C_2 \)
2. \( U \leftarrow (F_1 \land C_1) \land W \)
3. \( V \leftarrow (F_2 \land C_2) \land W \)
4. \( \Delta \leftarrow \emptyset \)
5. for all \( (\psi_1, d_1) \in \Delta_1, (\psi_2, d_2) \in \Delta_2 \) do
   6. if \( [\psi_1 \land \psi_2]_U \neq \emptyset \) then
      7. \( \psi \leftarrow (\psi_1 \land \psi_2) \land \psi_1 \land \psi_2 \land V \)
     8. \( \Delta \leftarrow \Delta \cup \{ \psi, d_1 \cap d_2 \} \)
   9. end if
10. end for
11. return \( (F_1 \cup F_2, C_1, \Delta) \)

covers the entire request space \( \Omega(F) \), that is such that

\[
\bigcup \{ [\psi]_F \mid (\psi, d) \in \Delta \} = \Omega(F).
\]

The decision function \( \delta : \Omega(F) \rightarrow D \) is the extension of \( \Delta \), written \( \delta = \text{ext}(\Delta) \), if the next statement holds:

\[
\delta(q) = \bigcup \{ d \mid (\psi, d) \in \Delta \land q \in [\psi]_F \}, \forall q \in \Omega(F).
\]

Note that, as a consequence of (4), the condition (3) can be always fulfilled without altering the extension of the rest of the policy by adding a default RD to \( \Delta \) that (i) matches all the possible requests and (ii) is associated with the decision 0. Moreover, whenever a concrete policy language features a deny by default semantics (as it is typically the case, e.g., for firewalls), the translation of such policies to DFD reduces to computing decisions for all the possible overlaps among rules within the policy. As discussed in Section VI the latter problem has been already extensively studied in literature on policy conflicts, hence it is left out in this paper.

We now define two operations on DFDs that are consistent with the (de)composition of decision functions.

**Definition 11 (DFD Projection):** The projection of a DFD \( \Delta \) over the set of fields \( P \subseteq F \) is the DFD \( \pi_P(\Delta) = \Delta' \) such that:

\[
(\psi, d) \in \Delta \Rightarrow (\psi, d) \in \Delta', \quad (\psi, d) \in \Delta.
\]

To see that Definition 11 is equivalent to the projection (Definition 7) of the DFD extension, it is sufficient to combine (4) with (5), that yields precisely (2).

Algorithm 1 computes instead the composition (Definition 6) of ACLs whose decision function is described by a DFD. The correctness of the algorithm, as stated in Proposition 2, is ensured by showing that the extension of the output DFD is the same as the composition of the extension of the input DFDs.

**Proposition 2 (Correctness of Algorithm 1):** Let the output of the algorithm be \( \langle F_1 \cup F_2, C_1, \Delta \rangle \) for input ACLs \( \langle F_1, C_1, \Delta_1 \rangle \) and \( \langle F_2, C_2, \Delta_2 \rangle \).

Then,

\[
\langle F_1, C_1, \text{ext}(\Delta_1) \rangle \otimes \langle F_2, C_2, \text{ext}(\Delta_2) \rangle = (F_1 \cup F_2, C_1, \text{ext}(\Delta)).
\]

Note how both Definition 11 and Algorithm 1 benefit from the choice of allowing the RDs in a DFD to overlap. In particular, we were able to define both operations without ever computing the disjunction or the complement of RDs.
(resp. the union and the difference of their extensions). This is convenient as the Cartesian product distributes with respect to intersection, but not to union and difference. Hence, as argued in [8], a number of RDs that grows linearly with the dimension of the product would be necessary to represent each binary union or difference.

In order to perform refactoring, we need to determine whether a decomposition is possible. This is equivalent to check if a decision function, once projected and composed back, equals itself. As different DFDS can have the same extension, this translates into testing the equivalence of \( (F_1 \cup F_2, C_1, \Delta) \) with \( (F_1, C_1, \pi_{F_1 \cup C_1}(\Delta)) \otimes (F_2, C_2, \pi_{F_2\cup C_2}(\Delta)) \), which means to compare the (possibly infinite) extensions of their DFDS.

In the next section we deal with this issue by developing an alternative criterion to test decomposability.

V. REFACTORIZATION

Through decomposition, we aim at factorizing the complexity of some layer’s policy into simpler ones. This means that the request type of any of the decomposed layers shall be a strict subset of the one of the original (composite) layer.

The next result shows that it is not guaranteed that such a decomposition exists for a generic access control layer.

**Proposition 3:** Given an ACL \( L = (F_1 \cup F_2, C_1, \delta) \) it is not always possible to find two ACLs \( (F_1, C_1, \delta_1), (F_2, C_2, \delta_2) \) that decompose \( L \) with \( F_2 \not\subseteq F_1 \).

**Sketch of the proof:** Assume that such a decomposition exists. Let \( q_1, q_2, q_3 \in \Omega(F_1 \cup F_2 \cup C_1) \) such that \( q_1 \neq q_2 \neq q_3 \) and \( q_1 \mid_{C_2\cup F_2} = q_2 \mid_{C_2\cup F_2} \) and \( q_3 \mid_{C_1 \cup C_1} = q_3 \mid_{C_1 \cup F_1} \). The assumption that \( F_2 \not\subseteq F_1 \) is crucial, otherwise \( q_2 = q_3 \). One choses \( \delta \) such that \( \delta(q_1) \neq \delta(q_2) \neq \delta(q_3) \). One obtains, according to (1), a contradiction.

The counterexample shown in the last proof captures the essence of non-decomposability. Consider, for instance, the decision function \( \delta \) (Table IV) and let \( q_1 = (2.2.2.1, 1.1.1.1, 0, 80, \text{acme.com}, /\text{private/}) \), \( q_2 = (2.2.3.1, 1.1.1.1, 0, 80, \text{acme.com}, /\text{private/}) \) and \( q_3 = (2.2.3.1, 1.1.1.1, 0, 80, \beta.com, /) \). Note that, in case \( C_1 \cup F_1 = \{I_d, I_2, P_3, P_4\} \) and \( C_2 \cup F_2 = \{I_d, P_3, H, U\} \), we have indeed a contradiction. In fact, as \( \delta(q_2) = 0 \), we need either \( \delta_1(2.2.3.1, 1.1.1.1, 0, 80) = \delta_1(x) = 0 \) or \( \delta_2(\text{acme.com, /private/}) = \delta_2(y) = 0 \). On the other hand, as \( \delta(q_3) = \delta(q_3) = 1 \), both \( \delta_1(x) = 1 \) and \( \delta_2(y) = 1 \) must hold. Intuitively, we see that in order to have decomposability, the decisions associated to requests that satisfy a specific inter-field dependency cannot be independent one from another. The next definition formalizes this intuition.

**Definition 12 (Inter-Field Dependency):** For \( W \) and \( V \) non-empty and disjoint subsets of \( F \), we say that a decision function \( \delta \) satisfies the Inter-Field Dependency (IFD) condition \( W \rightarrow V \), written \( \delta \models W \rightarrow V \), if and only if \( \forall q, q' \in \Omega(F), q|_W = q'|_W \Rightarrow \delta(q) \cap \delta(q') = \delta(q|_{F\setminus V} + q'|_{F\setminus V}) \cap \delta(q|_V + q'|_V) \).

We are now ready to state the main result of this section: IFDs precisely characterize when an ACL can be decomposed by projections without loss.

---

**Algorithm 2 DF D Partition**

**Input:** \( \Delta \subseteq \Psi(F) \times D \).

**Input:** \( X \subseteq F \).

**Output:** \( \text{PARTITION}(\Delta, X) \).

```
1: \( Y \leftarrow F \setminus X \)
2: \( P \leftarrow \emptyset \)
3: for all \( (\psi, d) \in \Delta \) do
4: \( P \leftarrow P \cup \{\psi|_X, \{\psi|_Y, d\}\} \)
end for
5: end if
6: while \( \exists (\psi_1, \Delta_1), (\psi_2, \Delta_2) \in P \text{ s.t. } [\psi_1 \wedge \psi_2]X \neq \emptyset \) do
7: \( P \leftarrow P \setminus \{(\psi_1, \Delta_1), (\psi_2, \Delta_2)\} \)
8: \( P \leftarrow P \cup \{(\psi_1 \cup \psi_2, \Delta_1 \cup \Delta_2)\} \)
9: for all \( \psi \in \text{DIFF}(\psi_1, \psi_2), \psi_{\Delta_1} \in \text{DIFF}(\psi_1, \psi_2) \) do
10: if \( [\psi_1 \wedge \psi_2]X \neq \emptyset \) then
11: \( P \leftarrow P \cup \{(\psi_{\Delta_1}, \Delta_2)\} \)
end if
13: if \( [\psi_{\Delta_1}]X \neq \emptyset \) then
14: \( P \leftarrow P \cup \{(\psi_{\Delta_1}, \Delta_2)\} \)
end if
15: end if
16: end for
17: end while
18: return \( P \)
```

---

**Algorithm 3 Inter-Field Dependency check on DF D**

**Input:** \( \Delta \subseteq \Psi(F) \times D \).

**Input:** \( W \rightarrow V \), with \( W, V \subseteq F \setminus \{\emptyset\}, W \cap V = \emptyset \).

**Output:** true/false

```
1: \( U \leftarrow F \setminus (W \cup V) \)
2: for all \( (\psi_1, \Delta_1) \in \text{PARTITION}(\Delta, W) \) do
3: \( \text{Diff_1} \leftarrow \emptyset \)
4: for all \( (\psi_1, \Delta_1) \in \text{PARTITION}(\Delta, W) \) do
5: \( \text{Diff_1} \leftarrow \text{Diff_1} \cup \{\psi\} \)
6: \( \text{Diff_1} \leftarrow \text{Diff_1} \cup \{\psi\} \)
7: end for
8: for all \( (\psi, \Delta_V) \in \text{PARTITION}(\Delta, V) \) do
9: \( \text{Diff_1} \leftarrow \text{Diff_1} \cup \{\psi\} \)
10: end for
11: end for
12: \( \text{Diff_1} \leftarrow \text{Diff_1} \cup \{\psi\} \)
13: end for
14: end if
15: return false
16: end if
17: end if
18: end for
19: end for
20: end for
21: end for
22: return true
```

---

**Theorem 4 (Decomposability):** Given a generic ACL \( L = (F_1 \cup F_2, C_1, \delta) \), the following are equivalent:

1. \( L = (F_1, C_1, \pi_{F_1\cup C_1}(\delta)) \otimes (F_2, C_2, \pi_{F_2\cup C_2}(\delta)) \)
2. \( \delta \models (F_1 \cup F_2) \rightarrow (F_2 \setminus F_1) \).

Theorem 4 gives an alternative way to test the decomposability of an ACL: we need to check if its decision function satisfies the IFD \( (F_1 \cup F_2) \rightarrow (F_2 \setminus F_1) \), for the subsets \( F_1, F_2, C_2 \) of its request type (with \( C_2 \subseteq C_1 \)) that represent the new layers’ layout we want to find a policy for.

IFDs are particularly interesting because they are constraints on the internal structure of the decision function that we can use well check on any corresponding DFDS. Algorithm 3 computes this check and the next proposition ensures its correctness.

**Proposition 5 (Correctness of Algorithm 3):** Let \( \Delta \subseteq \Psi(F) \times D \) be a DF D, \( \delta = \text{ext}(\Delta) \) its extension
and $W, V$ two non-empty and disjoint subsets of $F$.

Then, $\Delta \models W \rightarrow V \iff \delta \models W \rightarrow V$.

The key ideas underlying Algorithm 3 are as follows: First, we refactor the RDs contained in $\Delta$ to make them partition the entire request space (lines 2–7) such that every request matches exactly one RD $\psi_W + \psi_U + \psi_V$. Second, for every $\psi_W$, we consider all the pairs $\psi_U, \psi'_U$ and $\psi_V, \psi'_V$ (lines 8, 9) and we compute the greatest lower bound of the decisions associated with all the pairs of requests matching respectively $\psi_W + \psi_U + \psi'_V$ and $\psi_W + \psi'_U + \psi'_V$ (line 10). We finally check if the latter equals the greatest lower bound of all the pairs of requests matching $\psi_W + \psi_U + \psi'_V$ and $\psi_W + \psi'_U + \psi'_V$ (lines 11–18). To do the request space partition, we use iteratively the procedure defined by Algorithm 2 that computes a closure w.r.t. intersection and difference for the portion of the RDs concerning the subset of fields $X$. Note that here, unlike for previous algorithms, we need to compute the set of RDs describing the difference between two given RDs, formally $\text{DIFF}(\psi_1, \psi_2) = \{\psi'_1\}$ s.t. $\bigcup\{\|\psi'_1\|_F\} = \|\psi_1\|_F \setminus \|\psi_2\|_F$. This is generally possible for the RDs composed of the field descriptors considered in this paper (e.g., intervals of integers).

Given any ACL we can now first test for decomposability by the means of Algorithm 3 and then, in case of success, project (Definition 11) to obtain the desired decomposition.

Consider, for example, the ACL $L_a = \langle \{I, l_a, P_s, P_d, H, U\}, \{z\}, \delta_a \rangle$ that was introduced in Section III. As it is the result of the composition of ACLs $L_a = \langle \{I, l_a, P_s, P_d\}, \{z\}, \delta_a \rangle$ and $L_b = \langle \{I, H, U\}, \{l_a, P_d\}, \delta_b \rangle$, we naturally expect it to be decomposable with respect to their same structure. The reader can check by inspecting Table IV, that indeed $\delta_c = \{I, l_a, P_d\} \rightarrow \{H, U\}$. Hence, if we name $L'_a = \langle \{I, l_a, P_s, P_d\}, \{z\}, \pi_{\{l_a, l_b, P_s, P_d\}}(\delta_c) \rangle$ and $L'_b = \langle \{I, H, U\}, \{l_a, P_d\}, \pi_{\{l_a, l_b, P_s, H, U\}}(\delta_c) \rangle$, we know by Theorem 4 that $L_a = L'_a \otimes L'_b$. This is an example of refactoring that keeps the request type unchanged.

Table VI represents the decision function $\pi_{\{l_a, l_b, P_s, P_d\}}(\delta_c)$. Notice that it is not exactly equal to the original decision function $\delta_a$ of the ACL $L_a$ (cf. Table II). In particular, it is never more permissive than the original; on the other hand it is, where possible and according to the least privilege principle, more restrictive. For instance, requests coming from the 3.3.3.0/24 IP network and directed to 1.1.1.1, which were allowed by the original policy, are denied by the refactored version. This is consistent with the fact that such requests were anyway already denied in $L_b$ (cf. Table III). On the other hand, the decisions for all requests having source within 3.3.0.0/24 and directed to the rest of the 1.1.1.0/24 network, are refactored to $\bot$. This is a consequence of assuming partial knowledge of the $L_b$ policy. Since additional information is required to decide on the usefulness of those permissions, the choice is left to the user who can either trust the original policy, i.e., change $\bot$ to 1, or follow a more restrictive approach and change $\bot$ to 0. Let us now try to refactor $L_a, L_b$ to a pair of ACLs that have smaller request types. Suppose, for instance, to substitute the WS Web server of our scenario with one that does not discriminate requests on the basis of the IP source address; this means that the field $l_a$ does not belong any more to the request type of $L'_a$. We know, however, that such a decomposition is not possible, as $\delta_c \not\models \{l_a, P_d\} \rightarrow \{H, U\}$. See, as proof, the counterexample given earlier in this section, that clearly violates the condition given in Definition 12.

Had all the requests with $H = \text{acme.com}$ in $\delta_c$ been mapped to 0, the IFD would have been instead satisfied. In such a case we would have had a refactoring with a change in request types that simplified the decision function $\delta_c$.

VI. RELATED WORK

The pioneering work of Moffett and Sloman [9] has opened a large avenue for research on policy conflict analysis in distributed systems and has been refined, classified and formalized in the network-level security field. For instance, Al-Shaer et al. [10] or Basile et al. [8] have proposed techniques and algorithms to automatically discover and manage inconsistencies between firewall rule sets. One of the key points of these techniques is to turn rule sets into some canonical structure (e.g., ordered binary decision diagram [10], intersection closed semi-lattice of subsets [8]): An intermediate policy representation on which analysis and discovery is performed.

Although this paper does not focus on inconsistencies, it shares the abstract definition of devices and the ideas behind composition of policies to capture and analyze interactions between different layers. Related work on network-level analysis may complete our approach by providing means to deal with interactions between firewalls.

There exists a substantial body of work on combination of policies not limited to firewalls. Notably, several logical and algebraic approaches for composing and unifying access control policies have been proposed quite recently [7], [5], [6]. Different algebraic varieties have been used to combine rich decision spaces, for instance, $D$-algebras [7], Belnap bilattices [6] or XACML tailored logic [5]. One of the goals is to provide mathematical foundations to the expressive XACML access control language and its many resolution strategies.

The algebraic structure chosen here for decision space is motivated by previous work which have shown the need for expressive ones. However, the goal is not to capture resolution strategies but to find a structure both expressive enough and sufficient to define a decomposition with good properties. The use of an expressive common pivot model like XACML, logical frameworks or subject-target-condition rules [11] is very interesting. However, this paper chooses...
a different perspective by sticking as closely as possible to the original policies. Algorithms 1 and 3 carry computation directly on original policies, with minimal prior normalization.

Our approach has a strong connection to the policy refinement problem [12], [13], [11]. The key difference is that we start from interdependent concrete policies and we provide a device oriented decomposition instead of starting from high level requirements which are ultimately refined into operational policies. The refactoring problem studied in this paper is bottom-up: The global policy is nothing but the composition of all devices interacting along the network stack. By contrast, the policy refinement problem is clearly top-down.

Finally, the definition of access control layer is quite close to that of abstract access control systems [14], [15], [4]. The first two references formally compare the expressiveness of access control models with respect to the set of decision functions they can produce. Interest is not brought on the state but on the model itself, whereas we focus on the first. For the last, a set of desirable properties of abstract access control systems is identified. The question whether our decomposition technique still applies in their case is left for future work.

VII. CONCLUSION

This paper proposes a relational oriented approach for access control policies decomposition. The key concept that captures decomposability is the inter-field dependency condition given in Section V from which an algorithm for decomposability is derived. This algorithm works directly on a relational representation of access control policies.

Relational database users may have recognized the syntactical resemblance between $W ightarrow V$ of Definition 12 and so-called MultiValued Dependencies (MVD) introduced in [3]. The two concepts are related in the fact that Theorem 1 of [3], linking lossless decomposition of a relation and MVDs, is a specific instance of Theorem 4 of this paper, when the decision space is reduced to the two-elements boolean algebra.

We argue that the relational databases machinery can be fruitfully mapped to solve policies management problems. As ongoing and future work, we describe two such problems for which techniques borrowed from the database community could be of great interest.

The first problem is about repairs. Assume an access control policy over several fields cannot be decomposed into independent sub-policies: Is there any canonical or best way to update the policy such that it will become decomposable? Applying the many results for the repair problems (e.g., see [16]) to policies is not without difficulties. Basically, one has to generalize the relational framework to be able to capture generic definition of access control systems and thus has to provide new results inspired from classical ones, as we have done for MVD in some sense. Moreover, one has to find repair semantics meaningful from the security point of view. Standard repair semantics are repair via insertions, deletions or updates [17]. It is interesting to study how these semantics apply to policy decomposition, the last one in particular.

The second problem is about the mining of dependencies. In the classical setting of database normalization, the set of dependencies is known in advance and one tries to provide the best structure fitting with the given constraints, as it is done in this paper. The data-mining perspective reverses the approach: The goal is to discover new dependencies by revealing the internal structure of relations. Savnik and Flach have provided an efficient algorithm to discover MVD [18], we envision to apply their techniques to reverse engineer large policies. The idea is to find some “best set of simplest access control systems” with the same authorized queries.

ACKNOWLEDGMENT

This work was partially supported by the FP7-ICT-2009.1.4 Project PoSecCo (no. 257129, www.posecco.eu).

REFERENCES

APPENDIX

Note to the reviewers: this appendix is provided for the sake of completeness and will be not part of the camera-ready version, if any.

**Lemma 6:** Let \( (F_1 \cup F_2, C_1, \delta) = (F_1, C_1, \delta_1) \otimes (F_2, C_2, \delta_2) \), \( W = (F_1 \cap F_2) \cup C_2 \cup U = ((F_1 \setminus F_2) \setminus C_2) \cup C_1 \) and \( V = F_2 \cup F_1 \).

Then, \( \{U, V, W\} \) is a partition of \( (F_1 \cup C_1 \cup F_2) \).

**Proof:** Recall that \( F_1 \cap C_1 = \emptyset, i \in \{1,2\} \) (Definition 4) and that \( C_2 \subseteq F_1, F_2 \cap C_1 = \emptyset \) (Definition 6).

We then have the following equalities, which can be easily verified by inspection of Figure 2:

\[
\begin{align*}
U \cup W &= \left((F_1 \setminus F_2) \setminus C_2\right) \cup C_1 \cup (F_1 \cap F_2) \cup C_2 = (F_1 \setminus F_2) \cup C_1 = F_1 \cap C_1 \\
U \cap W &= \left(((F_1 \setminus F_2) \setminus C_2) \cup C_1 \cap ((F_1 \cap F_2) \cup C_2) = \emptyset \right) \\
V \cup W &= (F_2 \setminus F_2) \cup (F_1 \cap F_2) \cup C_2 = F_2 \cup C_2 \\
V \cap W &= (F_2 \setminus F_1) \cap ((F_1 \cap F_2) \cup C_2) = \emptyset \\
U \cap V &= \left(((F_1 \setminus F_2) \setminus C_2) \cup C_1 \cap (F_2 \cap F_1) = \emptyset \right) \\
U \cup V &= (F_1 \cup C_1) \cup (F_2 \cup C_2) = F_1 \cup C_1, F_2 = F_2 \cap C_2.
\end{align*}
\]

**Lemma 7:** Let \( \mathcal{L} \) be a finite bounded distributive lattice and \( f : X \to \mathcal{L}, g : Y \to \mathcal{L} \) two generic functions having \( \mathcal{L} \) as codomain. Then the following holds true:

\[
\bigcup_{x \in X} f(x) \cap \bigcup_{y \in Y} g(y) = \bigcup_{x \in X, y \in Y} f(x) \cap g(y).
\]

**Proof:** As \( \mathcal{L} \) is finite and because of lattices’ idempotence, \( \bigcup_{x \in X} f(x) = \bigcup \{l \in \mathcal{L} \mid \exists x \in X, l = f(x)\} = \bigcup \mathcal{L}_f \subseteq \mathcal{L} \).

Likewise \( \bigcup_{y \in Y} g(y) = \bigcup \mathcal{L}_g \subseteq \mathcal{L} \). By applying the distributive law of \( \sqcap \) with respect to \( \sqcup n \) times, where \( n = |\mathcal{L}_f| \times |\mathcal{L}_g| \) we obtain:

\[
\bigcup_{x \in X} f(x) \cap \bigcup_{y \in Y} g(y) = \bigcup_{t \in \mathcal{L}_f} \bigcup_{t' \in \mathcal{L}_g} \bigcup_{t \cap t' = t'}.
\]

As there are up to \( n \) different pairs \((l, l')\), by idempotence we conclude the thesis.

**Proof of Proposition 1:** Let \( L = (F_1 \cup F_2, C_1, \delta) \) such that \( L = (F_1, C_1, \delta_1) \otimes (F_2, C_2, \delta_2) \).

By Lemma 6, sets \( W = (F_1 \cap F_2) \cup C_2 \cup U = ((F_1 \setminus F_2) \setminus C_2) \cup C_1 \) and \( V = F_2 \cap F_1 \) partition \( F \). Moreover \( F_1 \cap C_1 = W \cup U \) (resp. \( F_2 \cup C_2 = W \cup U \)).

By combining the definitions of projection (Definition 7) and composition (Definition 6) and using the distributive law we obtain, \( \forall q = wuv \in \Omega(F_1 \cup C_1) = \Omega(W \cup U) \),

\[
\pi_{W \cup U}(\delta)(wu) = \bigcup_{v \in \Omega(V)} \delta_1(wu) \cap \delta_2(wv) = \delta_1(wu) \cap \bigcup_{v \in \Omega(V)} \delta_2(wv) \leq \delta_1(wu).
\]

Likewise we have, \( \forall q' = wuv \in \Omega(F_2 \cup C_2) = \Omega(W \cup V) \),

\[
\pi_{W \cup V}(\delta)(wu) = \bigcup_{v \in \Omega(U)} \delta_1(wu) \cap \delta_2(wv) \leq \delta_2(wv).
\]

**Proof of Proposition 3:** Let \( L = (F_1 \cup F_2, C_1, \delta) \), where \( F_1, F_2, C_1 \in 2^X \setminus \{\emptyset\} \) and assume that there exist \( L_1 = (F_1, C_1, \delta_1) \) and \( L_2 = (F_2, C_2, \delta_2) \) such that \( L = L_1 \otimes L_2 \) (hence \( C_2 \subseteq F_1 \) and \( F_2 \subseteq F_1 \)).

Let \( F = F_1 \cup F_2 \). By Lemma 6, sets \( W = (F_1 \cap F_2) \cup C_2 \cup U = ((F_1 \setminus F_2) \setminus C_2) \cup C_1 \) and \( V = F_2 \cap F_1 \) partition \( F \). Since we assumed \( F_2 \not\subseteq F_1 \) and sets \( F_1, F_2 \) are not empty by hypothesis, it must hold that \( U \) and \( V \) are not empty either. Hence, \( \exists u_1, u_2 \in \Omega(U) \exists v_1, v_2 \in \Omega(V) \) such that \( u_1 \neq u_2 \wedge v_1 \neq v_2 \). Moreover, as \( U, V \) partition \( F \), there exist \( q_1, q_2, q_3 \in \Omega(W \cup U \cup V) = \Omega(\{F\}) \) such that:

\[
q_1 = wu_1 v_1, \\
q_2 = wu_2 v_1, \\
q_3 = wu_2 v_2.
\]

Assume now \( \delta(q_1) \neq \emptyset \) and \( \delta(q_2) = \emptyset \), \( \delta(q_3) \neq \emptyset \) that, according to Definition 6, entails the following contradiction:

\[
\begin{align*}
\delta_1(q_1\mid_{C_1 \cup F_1}) &= \delta_1(wu_2) \neq \emptyset_D \\
\delta_2(q_1\mid_{C_2 \cup F_2}) &= \emptyset_D \\
\delta_1(q_2\mid_{C_1 \cup F_1}) &= \delta_1(wu_2) = \emptyset_D \\
\delta_2(q_2\mid_{C_2 \cup F_2}) &= \delta_2(wu_2) = \emptyset_D.
\end{align*}
\]

**Proof of Proposition 4:** As stated in Lemma 6, it is possible to partition the decision space of \( \delta \) in the following three sets of fields: \( W = (F_1 \cap F_2) \cup C_2 \cup U = ((F_1 \setminus F_2) \setminus C_2) \cup C_1 \) and \( V = F_2 \cap F_1 \). A generic request \( q \) in such a request space can then be expressed as the concatenation of sequences \( w \in \Omega(W), u \in \Omega(U) \) and \( v \in \Omega(V) \), i.e., \( q = wuv \).

\[\Leftarrow\text{ For the } if\text{ direction we assume } \delta \mid W \rightarrow V\text{ and we show that } \forall q \in \Omega(F), \pi_{F_1 \cup C_1}(\delta)(q\mid_{F_1 \cup C_1}) \cap \pi_{F_2 \cup C_2}(\delta)(q\mid_{F_2 \cup C_2}) = \delta(q).\]

First, notice that, as \( \delta \mid W \rightarrow V \), according to Definition 12 the following is true \( \forall w \in \Omega(W), u, u' \in \Omega(U), v, v' \in \Omega(V): \)

\[
\delta(wuv) \cap \delta(wu'v) = \delta(wuv) \cap \delta(wu'v).
\]

We now have the following sequence of equalities (the passages marked by distr. and abs. hold respectively by distribu-
Thus, and according to absorption lattice laws,

$$\pi_{W \cup U}(\delta(wu) \cap \pi_{W \cup U}(\delta(wv))$$

$$= \left( \bigsqcup_{v' \in \Omega(V)} \delta(wuv') \right) \cap \left( \bigsqcup_{v' \in \Omega(U)} \delta(wuv') \right)$$

by (2)

$$= \delta(wuv) \cup \left( \bigsqcup_{v' \in \Omega(V) \setminus \{v\}} \delta(wuv') \right) \cap \delta(wuv) \cup \left( \bigsqcup_{v' \in \Omega(U) \setminus \{v\}} \delta(wuv') \right)$$

(distr.)

$$= \delta(wuv) \cap \left( \bigsqcup_{v' \in \Omega(V)} \delta(wuv') \right)$$

by (6)

$$= \delta(wuv).$$

(abs.)

⇒. For the only if direction we reason by contradiction. Assume $\forall q \in \Omega(F)$

$$\pi_{F_1 \cup C_1}(\delta(q) | F_1 \cup C_1) \cap \pi_{F_2 \cup C_2}(\delta(q) | F_2 \cup C_2) = \delta(q),$$

and $\delta \neq W \rightarrow V$. For instance, let

$$\delta(wu_1v_1) \cap \delta(wu_2v_2) \neq \delta(wu_1v_2) \cap \delta(wu_2v_1),$$

for some $w \in \Omega(W), u_1, u_2 \in \Omega(U), v_1, v_2 \in \Omega(V)$.

We know, from the first half of the proof, that we can rewrite (8) as

$$\delta(wu_1v_1) = \delta(wu_1v_1) \cup \left( \bigsqcup_{v' \in \Omega(V \setminus \{v_1\})} \delta(wu_1v') \right) \cap \delta(wu_2v_1).$$

Now we take the greatest lower bound of the expression $\delta(wu_1v_2) \cap \delta(wu_2v_1)$ with both sides of the equality. Then we recognize that the same expression is always a factor of the series of least upper bound operations. This allows us to cancel out the series by absorption:

$$\delta(wu_1v_1) \cap \delta(wu_2v_2) \cap \delta(wu_2v_1)$$

$$= \delta(wu_1v_2) \cap \delta(wu_2v_1)$$

$$= \left( \bigsqcup_{v' \in \Omega(V \setminus \{v_1\})} \delta(wu_1v') \right) \cap \delta(wu_2v_1)$$

$$= \delta(wu_1v_1) \cup \left( \bigsqcup_{v' \in \Omega(V \setminus \{v_1\})} \delta(wu_1v') \right)$$

$$= \delta(wu_1v_2) \cap \delta(wu_2v_1);$$

thus $\delta(wu_1v_2) \cap \delta(wu_2v_1) \leq \delta(wu_1v_1)$.

By repeating the same reasoning on $\delta(wu_1v_2) \cap \delta(wu_2v_1)$, and $\delta(wu_2v_1)$ we conclude

$$\begin{cases} 
\delta(wu_1v_2) \cap \delta(wu_2v_1) \leq \delta(wu_1v_1) \\
\delta(wu_1v_2) \cap \delta(wu_2v_1) \leq \delta(wu_2v_1) \\
\delta(wu_1v_1) \cap \delta(wu_2v_2) \leq \delta(wu_1v_1) \\
\delta(wu_1v_1) \cap \delta(wu_2v_2) \leq \delta(wu_2v_2)
\end{cases}$$

(10)

that, as $x \leq y \land z \leq w \Rightarrow x \lor z \leq y \lor w$ yields

$$\begin{cases} 
\delta(wu_1v_2) \cap \delta(wu_2v_1) \leq \delta(wu_1v_1) \cap \delta(wu_2v_1) \\
\delta(wu_1v_1) \cap \delta(wu_2v_2) \leq \delta(wu_1v_1) \cap \delta(wu_2v_2)
\end{cases}$$

(9)

By antisymmetry, the latter entails $\delta(wu_1v_1) \cap \delta(wu_2v_2) = \delta(wu_1v_2) \cap \delta(wu_2v_1)$, in contradiction with (9).

Proof of Proposition 2: Let $\langle F_1 \cup F_2, C_1, \Delta \rangle = \langle F_1, C_1, \Delta_1 \rangle \otimes \langle F_2, C_2, \Delta_2 \rangle$ and $\langle F_1 \cup F_2, C_1, \delta \rangle = \langle F_1, C_1, \delta_1 \rangle \otimes \langle F_2, C_2, \delta_2 \rangle$.

We need to show that

$$\delta_1 = \text{ext}(\Delta_1) \land \delta_2 = \text{ext}(\Delta_2) \Rightarrow \delta = \text{ext}(\Delta).$$

Let us denote $W \cup U \cup V = F$. Let $wu \in \Omega(W \cup U)$ and $wu \in \Omega(W \cup V)$. Then consider all the pairs $\langle \psi_1, d_1 \rangle \in \Delta_1$ and $\langle \psi_2, d_2 \rangle \in \Delta_2$ such that $wu \in \langle \psi_1 \rangle | W \cup U \land wu \in \langle \psi_2 \rangle | W \cup V$.

Since all such $\psi_1, \psi_2$ guarantee $wu \in \langle \psi_1 \rangle | W \land \psi_2 | w \subseteq \langle \psi_1 \rangle | W \cup U \cup V$, they surely fulfill the condition of line 6. Hence, by lines 7 and 8, there exists $\langle \psi, d_1 \land d_2 \rangle \in \Delta$ such that $wu \in \langle \psi \rangle | F$.

We thus have the following result:

$$\langle \psi, d_1 \rangle \in \Delta_1 \land wu \in \langle \psi_1 \rangle | W \cup U \land \langle \psi, d_2 \rangle \in \Delta_2 \land wu \in \langle \psi_2 \rangle | W \cup V \Rightarrow \langle \psi, d_1 \land d_2 \rangle \in \Delta \land wu \in \langle \psi \rangle | F.$$ (11)

We can now prove our thesis:

$$\forall wu \in \Omega(F), \ \delta(wu) = \delta_1(wu) \cap \delta_2(wu)$$

(Hyp)

$$= \text{ext}(\Delta_1) \cap \text{ext}(\Delta_2)$$

(12)

$$= \left\{ \begin{array}{l} 
\{d_1 \mid \langle \psi_1, d_1 \rangle \in \Delta_1 \land wu \in \langle \psi_1 \rangle | W \cup U \} \\
\{d_2 \mid \langle \psi_2, d_2 \rangle \in \Delta_2 \land wu \in \langle \psi_2 \rangle | W \cup V \} 
\end{array} \right\}$$

by (4)

$$= \left\{ \begin{array}{l} 
\{d_1 \cap d_2 \mid \langle \psi_1, d_1 \rangle \in \Delta_1 \land wu \in \langle \psi_1 \rangle | W \cup U \land \langle \psi_2, d_2 \rangle \in \Delta_2 \land wu \in \langle \psi_2 \rangle | W \cup V \} \\
\{d \mid \langle \psi, d \rangle \in \Delta \land wu \in \langle \psi \rangle | F \} 
\end{array} \right\}.$$ (13)

Hence, by (4), we conclude $\delta = \text{ext}(\Delta)$.

Lemma 8: Let $\Delta \subseteq \Psi(F) \times D$ and $X \subseteq F$. Then, $\forall q \in \Omega(F)$, $\exists \langle \psi_X, \Delta_X \rangle \in \text{PARTITION}(\Delta, X)$ s.t.

$$q | x \in \langle \psi_X \rangle | x$$

(12)

and

$$\text{ext}(\Delta)(q) = \text{ext}(\Delta_X)(q | F \setminus X)$$

(13)

Proof: The first result (12) follows by observing that the main loop of Algorithm 2 removes all the possible overlaps among the $X$-projection of the RDs in $\Delta$. Hence, at most one $\psi_X$ will exist that matches every possible $q | X$. More strongly, as the DFD covers the entire request space (cf. (3), exactly
one such $q|_X$ must exist. The second result (13) follows by substituting the definition of DFD extension (4).

**Proof of Proposition 5:** $\Leftarrow$. We obtain the first half by contraposition.

Assume that Algorithm 3 returns false (Line 15). Hence, there exist $\langle \psi_W, \Delta_W \rangle$, $\langle \psi_V, \Delta_V \rangle$, $\langle \psi'_U, \Delta'_U \rangle$, $\langle \psi''_V, \Delta''_V \rangle$, $\langle \psi'''_V, \Delta'''_V \rangle$ such that:

\[
\begin{align*}
\langle \psi_W, \Delta_W \rangle & \in \text{PARTITION}(\Delta, W) \\
\langle \psi_V, \Delta_V \rangle, \langle \psi'_U, \Delta'_U \rangle & \in \text{PARTITION}(\Delta_W, U) \\
\langle \psi''_V, \Delta''_V \rangle & \in \text{PARTITION}(\Delta_U, V) \\
\langle \psi'''_V, \Delta'''_V \rangle & \in \text{PARTITION}(\Delta'_U, V)
\end{align*}
\]

and (Line 12):

\[
[\psi' \land \psi''_V]_V \neq \emptyset \land [\psi' \land \psi'''_V]_V \neq \emptyset
\]  

(14)

and (Lines 10, 13 and 14):

\[
\begin{align*}
\bigcup_{d \in \Delta_V} d \cap \bigcup_{d' \in \Delta'_V} d' \neq \bigcup_{d'' \in \Delta''_V} d'' \cap \bigcup_{d''' \in \Delta'''_V} d'''.
\end{align*}
\]  

(15)

Note that for all $\langle \psi, d \rangle \in \Delta_V$ (and likewise for $\Delta'_V, \Delta''_V, \Delta'''_V$), $V$ must equal the empty sequence $\langle \rangle$. This is a consequence of partitioning $\Delta_U \subseteq \Psi(V) \times D$ (resp. $\Delta'_U$) on $V$. For the sake of clarity, we abuse notation and simply write $d$ for $\langle \rangle, d$). According to (4), we can then rewrite (15) as:

\[
\text{ext}(\Delta_V)(\langle \rangle) \cap \text{ext}(\Delta'_V)(\langle \rangle) \neq \text{ext}(\Delta''_V)(\langle \rangle) \cap \text{ext}(\Delta'''_V)(\langle \rangle).
\]  

(16)

The partitions of $\Delta$ are represented in Table VII.

Because of (12) and (14) we can choose $q, q', q'', q''' \in \Omega(W \cup U \cup V)$ such that:

\[
\begin{align*}
q|_W, q'|_W, q''|_W, q'''|_W & \in [\psi_W]|_W \\
q|_U, q'|_U & \in [\psi_U]|_U \\
q'|_U, q''|_U, q'''|_U & \in [\psi_U]|_U \\
q'|_V, q''|_V & \in [\psi_V \land \psi''_V]|_V \\
q'|_V, q'''|_V & \in [\psi_V \land \psi'''_V]|_V
\end{align*}
\]

and

\[
\begin{align*}
\text{ext}(\Delta_V)(\langle \rangle) \cap \text{ext}(\Delta'_V)(\langle \rangle) \neq \text{ext}(\Delta''_V)(\langle \rangle) \cap \text{ext}(\Delta'''_V)(\langle \rangle).
\]  

moreover, as a consequence of (13), we have:

\[
\begin{align*}
\text{ext}(\Delta)(q) & = \text{ext}(\Delta_U)(q|_U \cup V) = \text{ext}(\Delta_U)(q|_U \cup V) \\
& = \text{ext}(\Delta_V)(q|_U \cup V|_U) = \text{ext}(\Delta_V)(q) \\
\text{ext}(\Delta)(q') & = \text{ext}(\Delta_U)(q'|_U \cup V) = \text{ext}(\Delta_U)(q'|_U \cup V) \\
& = \text{ext}(\Delta'_V)(q'|_U \cup V|_U) = \text{ext}(\Delta'_V)(q') \\
\text{ext}(\Delta)(q'') & = \text{ext}(\Delta_U)(q''|_U \cup V) = \text{ext}(\Delta_U)(q''|_U \cup V) \\
& = \text{ext}(\Delta''_V)(q''|_U \cup V|_U) = \text{ext}(\Delta''_V)(q'') \\
\text{ext}(\Delta)(q''') & = \text{ext}(\Delta_U)(q'''|_U \cup V) = \text{ext}(\Delta_U)(q'''|_U \cup V) \\
& = \text{ext}(\Delta'''_V)(q'''|_U \cup V|_U) = \text{ext}(\Delta'''_V)(q''')
\end{align*}
\]

Table VII: \text{PARTITIONED $\Delta$}

<table>
<thead>
<tr>
<th>Partition($\Delta, W$)</th>
<th>Partition($\Delta_W, U$)</th>
<th>Partition($\Delta_U, V$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle \psi_W, \Delta_W \rangle$</td>
<td>$\langle \psi_V, \Delta_V \rangle$</td>
<td>$\langle \psi'_U, \Delta'_U \rangle$</td>
</tr>
<tr>
<td>$\langle \psi'_V, \Delta''_V \rangle$</td>
<td>$\langle \psi''_V, \Delta'''_V \rangle$</td>
<td>$\langle \psi'''_V, \Delta'''_V \rangle$</td>
</tr>
</tbody>
</table>

| \ldots | \ldots | \ldots |

\[
\begin{align*}
\text{ext}(\Delta)(q) & \cap \text{ext}(\Delta)(q') \neq \text{ext}(\Delta)(q'') \cap \text{ext}(\Delta)(q''').
\]  

(17)

By (12) then, there is a unique $\langle \psi_W, \Delta_W \rangle$ such that $q|_W, q'|_W \cup W, q''|_W \cup W, q'''|_W \cup W \in [\psi_W]|_W$.

Let $\langle \psi_U, \Delta_U \rangle \in \text{PARTITION}(\Delta_W, U)$. If $q|_U, q'|_U \in [\psi_U]|_U$, then $q''|_U, q'''|_U \in [\psi_U]|_U$ too, as $q|_U = q''|_U$ and $q''|_U = q'''|_U$. Similarly, as $q|_V = q''|_V$ and $q''|_V = q'''|_V$, there exist two unique $\langle \psi_V, \Delta_V \rangle, \langle \psi''_V, \Delta''_V \rangle \in \text{PARTITION}(\Delta_U, V)$ such that $q|_V, q''|_V \in [\psi_V]|_V$ and $q''|_V, q'''|_V \in [\psi''_V]|_V$. By (13), we then rewrite (17) as

\[
\text{ext}(\Delta)(\langle \rangle) \cap \text{ext}(\Delta''_V)(\langle \rangle) \neq \text{ext}(\Delta''_V)(\langle \rangle) \cap \text{ext}(\Delta'''_V)(\langle \rangle),
\]

a contradiction. Hence it must be $q|_U \in [\psi_U]|_U$ and $q''|_U \in [\psi''_U]|_U$ for some $\langle \psi_U, \Delta_U \rangle, \langle \psi''_U, \Delta''_U \rangle \in \text{PARTITION}(\Delta_W, U)$.

Let now $\langle \psi_V, \Delta_V \rangle, \langle \psi''_V, \Delta''_V \rangle \in \text{PARTITION}(\Delta_U, V)$ and $\langle \psi'''_V, \Delta'''_V \rangle \in \text{PARTITION}(\Delta'_U, V)$ such that:

\[
\begin{align*}
q|_V, q''|_V \in [\psi_V]|_V \land q''|_V, q'''|_V \in [\psi'''_V]|_V \\
q''|_V, q'''|_V \in [\psi'''_V]|_V \land q''|_V, q''|_V \in [\psi''_V]|_V
\end{align*}
\]

It is clear that such $\psi_V, \psi''_V, \psi'''_V$ satisfy the condition of Line 12 of Algorithm 3. Moreover, by rewriting (17), we now obtain

\[
\text{ext}(\Delta_V)(\langle \rangle) \cap \text{ext}(\Delta''_V)(\langle \rangle) \neq \text{ext}(\Delta''_V)(\langle \rangle) \cap \text{ext}(\Delta'''_V)(\langle \rangle),
\]

which means that such a combination will satisfy the condition of Line 14 too. As a consequence, the algorithm will return false (Line 15).■