On the Isofunctionality of Network Access Control Lists

Malek Belhaouane, Joaquin Garcia-Alfaro and Hervé Debar
Institut Mines-Telecom, Télécom SudParis
CNRS Samovar UMR 5157, Evry, France
{malek.belhaouane, joaquin.garcia_alfaro, herve.debar}@telecom-sudparis.com

***Abstract*—In a networking context, Access Control Lists (ACLs) refer to security rules associated to network equipment, such as routers, switches and firewalls. Methods and tools to automate the management of ACLs distributed among several equipment shall verify if the corresponding ACLs are functionally equivalent. In this paper, we address such a verification process. We present a formal method to verify when two ACLs are isofunctional and illustrate our proposal over a practical example.**

Keywords: Network Security, Computer Security, Authorization, Access Control, Policy Analysis, Policy Management.

I. Introduction

The use of information and communication technologies is growing rapidly. Corporate networks, always in constant metamorphose, shall face the expansion of network security threats, and constantly update their security strategies. These strategies must be consolidated in network equipment such as routers, switches and firewalls. Such equipment represent the first line of defense against network attacks, and are crucial to protect the corporate assets. They are in charge of supervising data flows and deciding, e.g., which ones to accept or to reject according to the Access Control Lists (ACLs) defined by network administrators. The task of authoring and managing ACLs becomes complex and error-prone [1], [2].

The need for tools to help network administrators in the network management process is increasing. Several tools have been proposed in the literature to assist administrators. For instance, numerous studies [3]–[9] focused on the detection and the resolution of anomalies in ACLs such as conflicts and misconfigurations. Other researchers worked on the bottom up and top down problem [10], [11]. This way, they propose methods and tools to automate the migration of an ACL from a network equipment to another. Finally, some work was carried out on analyzing the performance of network equipment and optimizing them by modifying their ACLs [12]–[15].

Generally, all the proposed tools and methods are either transforming an ACL to a more optimized one, or comparing the characteristics of two ACLs. As a consequence, the proposed tools and methods are manipulating two ACLs that implement the same security policy. Before comparing the characteristics of two ACLs, we need to prove that they are functionally equivalent or in other terms, isofunctional. In this paper, we address the isofunctionality concept and develop a formal method to verify when two ACLs are isofunctional.

Paper Organization – Section II presents our motivation. Section III provides some necessary definitions. Section IV presents our isofunctionality verification proposal. Section V illustrates our proposal over a practical example. Section VI provides related work. Section VII closes the paper.

II. Motivation

There are several situations that justify the necessity of verifying that two ACLs are isofunctional. For instance, to comply with their corporate security directives, network administrators often need to upgrade or replace some equipment. Such new equipment can come from different vendors and possibly use different languages. Existing tools proposed in the literature allow to transform the ACLs of different vendor equipment [16], [17]. After the transformation process, the administrator needs to verify that the new ACLs are functionally equivalent to the initial ones. In other words, it should be verified that the equipment implementing the new ACLs provide the same treatment for all data flows they will need to handle.

Network administrators may also need to address inconsistencies in the deployment of ACLs, such as conflicting or redundant rules. Solutions have been proposed in the literature to detect and resolve such anomalies [1], [4], [7]. After using them, the administrator will have two ACLs: the initial ACL and the resulting ACL where the anomalies were fixed. The administrator needs to verify again that the two ACLs are functionally equivalent. Similarly, network administrator often need to compare the characteristics of ACLs written under the assumption of different equipment capabilities [18]. This can be necessary to evaluate their latency during the execution of an equivalent process, in order to decide the most convenient deployment. In order to conduct the evaluation analysis, the isofunctionality of the different ACLs needs to be verified. Otherwise, the analysis will be biased.

In all the aforementioned situations, we observe the necessity of verifying the isofunctionality of two or more ACLs. In the following sections, we present our proposal to conduct such a verification process.

III. Definitions

Consider network security equipment as network entities in charge of controlling the traversal of packets across network segments. Then, assume such packets triggering queries into the network security equipment. For each received query, an access control decision shall be conducted by the network equipment, according to an Access Control List (ACL). In turn, an ACL consists of a list of rules that define the precise actions that must be performed upon queries satisfying certain conditions. Concretely, each rule is specified by a predicate and a decision. A predicate in a rule is a boolean function over of a set of attributes to identify a certain type of queries matched by the rule. The decision represents the action that the network security equipment shall perform. Next, we provide some more formal definitions about the aforementioned concepts.

Definition 1 (ACL): An Access Control List (or ACL for short) is a set of m rules, such as:

$$\begin{array}{c} r_1 : P_1, decision_1 \\ r_2 : P_2, decision_2 \\ \dots \\ r_m : P_m, decision_m \end{array}$$

where P_i is the predicate for the i^{th} rule and $decision_i \in \mathcal{D}$ is an access control decision. $\mathcal{D}$ denotes a finite set of access control decisions. In other words, every rule is composed of a predicate and a decision, being a predicate a boolean function over of a set of attributes used to describe queries.

Definition 2 (Attribute): An attribute is any characteristic of a network packet to which a value may be assigned. Let $\mathcal{A}$ denote a set of attributes. Let $a \in \mathcal{A}$ be an attribute. Then, the attribute domain of a, denoted as $\mathcal{V}_a$, is the set of values that can be taken by a.

Example 1: Assume that the attributes that characterize a network packet are: *Source address*, *Destination address*, *Source port* and *Destination port*. The domain of the attribute *Source port* is $\mathcal{V}_{source\_port} = [0, 65535]$

Definition 3 (Predicate): A predicate is a Boolean-valued function over attributes. Let P be a predicate over a set of n attributes of $\mathcal{A}$, then P is represented as follows:

$$P : \mathcal{A}^n \rightarrow \{\texttt{True}, \texttt{False}\}$$

where n is the number of attributes composing P.

Example 2: For a better illustration of the aforementioned concepts, assume the following example. Let attributes a_1, a_2, a_3, a_4 correspond to *Source address*, *Destination address*, *Source port* and *Destination*. Let $\delta_1 = \{192.168.1.[1, 100]\}$, $\delta_2 = \{[1024, 49151]\}$, $\delta_3 = \{172.16.10.50\}$ and $\delta_4 = \{[80, 443]\}$ be attribute value sets such that $\delta_1 \subset \mathcal{V}_{a_1}$, $\delta_2 \subset \mathcal{V}_{a_2}$, $\delta_3 \subset \mathcal{V}_{a_3}$ and $\delta_4 \subset \mathcal{V}_{a_4}$. Then, we can define predicate P as $P = (a_1 \in \delta_1) \wedge (a_2 \in \delta_2) \wedge (a_3 \in \delta_3) \wedge (a_4 \in \delta_4)$.

Definition 4 (Default rule predicate): Each ACL has a default rule. The predicate of the default rule shall match all the possible packets received by the network security equipment. Formally, let $a_1, a_2, \dots, a_n \in \mathcal{A}$ be attributes. Let $\mathcal{V}_{a_1}, \mathcal{V}_{a_2}, \dots, \mathcal{V}_{a_n}$ be their respective domains. Then, the default rule predicate is expressed as follows:

$$P_{default} = ((a_1 \in \mathcal{V}_{a_1}) \wedge (a_2 \in \mathcal{V}_{a_2}) \wedge \dots \wedge (a_n \in \mathcal{V}_{a_n}))$$

Definition 5 (Query): The function of a network equipment is to associate to each query it receives a decision according to its ACL. Therefore, a query represents the network packet that transits over the equipment. Formally, let $a_1, a_2, \dots, a_n$, be n attributes of a set of attributes $\mathcal{A}$, such that every attribute represents a given field of a network packet. Let $v_1, v_2, \dots, v_n$, be n attribute values such that $v_1 \in \mathcal{V}_{a_1}, v_2 \in \mathcal{V}_{a_2}, \dots, v_n \in \mathcal{V}_{a_n}$. Then, a query q is represented as follows:

$$q = (a_1 = v_1) \wedge (a_2 = v_2) \wedge \dots \wedge (a_n = v_n)$$

Definition 6 (Query interval): Set of queries whose attributes satisfy the conditions of a given ACL rule. Like predicates, query intervals are represented as Boolean expressions.

Definition 7 (Query space): A set of query intervals $QS = \{Q_1, Q_2, \dots, Q_p\}$ that satisfies the following properties:

- Query intervals in QS are mutually disjoint, i.e., $\forall QS' \subset QS$, $\forall Q \in QS \backslash QS'$, then
$$\left(\bigvee_{\forall Q' \in QS'} Q' \right) \wedge Q = False$$
- Let $\mathcal{Q}$ be the set of all possible queries, then any query in $\mathcal{Q}$ matches a query interval in QS, i.e., $\forall q \in \mathcal{Q}$, $\exists Q_i \in QS, 1 \leq i \leq p$, such that $q \wedge Q_i = True$

Definition 8 (Decision Space): A decision space is computed by applying the rules of an ACL over the query intervals of a query space. The goal is to map to each query interval those rules matching the interval and their corresponding decision. Formally, let QS be a query space and $\texttt{A}$ an ACL. Then, the decision space of QS over the rules in $\texttt{A}$ is a set of triples $\{Interval, Rules, Decision\}$, such that $Interval$ is a query interval in QS, $Rules$ contains those rules in $\texttt{A}$ whose predicates match the queries in $Interval$, and $Decision$ is defined as follows:

$$Decision = \begin{cases} \emptyset : Rules = \emptyset \\ \emptyset : \exists r_i, r_j \in Rules, decision_{r_i} \neq decision_{r_j} \\ D : \forall r \in Rules, D = decision_r \end{cases}$$

Definition 9 (Isofunctionality): Two ACLs $\texttt{A}$ and $\texttt{B}$ are isofunctional if, no matter the queries they receive, they are always equivalent in function to their decision results. In other words, any query issued to ACL $\texttt{A}$ gets exactly the same decision result than if issued to $\texttt{B}$, and vice versa.

IV. Our Proposal

This section presents our proposal for the verification of isofunctionality among two ACLs. Algorithm 1 performs the main actions associated to the verification process. As input, it receives the two ACLs that we want to compare. Some further operations are described by the auxiliary functions `GetQuerySpace` (cf. Algorithm 2), `GetDecisionSpace` (cf. Algorithm 3) and `CompareDecisionSpaces` (cf. Algorithm 4). Notice that the process has two different phases.

During the first phase (cf. Algorithm 1, lines [1-4]), the process computes the query space, $\mathcal{Q}_{\mathcal{A}}$, from ACL $\texttt{A}$. It checks then the query intervals of $\mathcal{Q}_{\mathcal{A}}$ against the ACL rules of both $\texttt{A}$ and $\texttt{B}$. Next, it computes the decision spaces $\mathcal{D}_{\mathcal{AA}}$ and $\mathcal{D}_{\mathcal{BA}}$ associated to the query space $\mathcal{Q}_{\mathcal{A}}$. This is done by applying respectively ACLs $\texttt{A}$ and $\texttt{B}$. Likewise, it compares the two decision spaces $\mathcal{D}_{\mathcal{AA}}$ and $\mathcal{D}_{\mathcal{BA}}$. If both are equal, it is concluded that any query issued to $\texttt{A}$ gets exactly the

Algorithm 1 `CheckIsofunctionality`(A,B)

```
1: $\mathcal{Q}_{\mathcal{A}} \leftarrow$ GetQuerySpace($A$)
2: $\mathcal{D}_{\mathcal{AA}} \leftarrow$ GetDecisionSpace($\mathcal{Q}_{\mathcal{A}}$,$A$)
3: $\mathcal{D}_{\mathcal{BA}} \leftarrow$ GetDecisionSpace($\mathcal{Q}_{\mathcal{A}}$,$B$)
4: $Output \leftarrow$ CompareDecisionSpaces($\mathcal{D}_{\mathcal{AA}}$,$\mathcal{D}_{\mathcal{BA}}$)
5: if $Output =$ False then
6:     return False //$A$ and $B$ are non-isofunctional
7: else
8:     $\mathcal{Q}_{\mathcal{B}} \leftarrow$ GetQuerySpace($B$)
9:     $\mathcal{D}_{\mathcal{BB}} \leftarrow$ GetDecisionSpace($\mathcal{Q}_{\mathcal{B}}$,$B$)
10:    $\mathcal{D}_{\mathcal{AB}} \leftarrow$ GetDecisionSpace($\mathcal{Q}_{\mathcal{B}}$,$A$)
11:    $Output \leftarrow$ CompareDecisionSpaces($\mathcal{D}_{\mathcal{BB}}$,$\mathcal{D}_{\mathcal{AB}}$)
12:    if $Output =$ False then
13:        return False //$A$ and $B$ are non-isofunctional
14:    else
15:        return True //$A$ and $B$ are isofunctional
16:    end if
17: end if
```

Algorithm 2 `GetQuerySpace(ACL)`

```
1: $Output \leftarrow \emptyset$
2: $Predicates \leftarrow$ False
3: for all $r \in ACL$ do
4:     $Interval \leftarrow Predicate_r \wedge \neg Predicates$
5:     $Predicates \leftarrow Predicates \vee Predicate_r$
6:     $Output \leftarrow Output \cup \{Interval\}$
7: return $Output$
```

same decision results than if issued to ACL `B`. Otherwise, it is concluded that the two ACLs are non-isofunctional.

During the second phase (cf. Algorithm 1, lines [8-11]), it is computed the query space $\mathcal{Q}_{\mathcal{B}}$. Using $\mathcal{Q}_{\mathcal{B}}$, decision spaces $\mathcal{D}_{\mathcal{AB}}$ and $\mathcal{D}_{\mathcal{BB}}$ (associated to $\mathcal{Q}_{\mathcal{B}}$) are obtained by applying, respectively, ACLs `A` and `B`. Once compared the two decision spaces, if $\mathcal{D}_{\mathcal{AB}}$ and $\mathcal{D}_{\mathcal{BB}}$ are equal, it is concluded that ACLs `A` and `B` are isofunctional. Otherwise, it is concluded that they are non-isofunctional.

Algorithm 2 describes auxiliary Function `GetQuerySpace`. Its input is an ACL. Each element in the ACL is a rule with a predicate and decision (cf. Definition 1). To simplify, we use notation $Predicate_r$ as an abbreviation of predicate of rule r. The output of Algorithm 2 is the query space (cf. Definition 7) of the ACL. Algorithm 2 iterates over each rule r in the ACL and extracts all query intervals (cf. Definition 6) by processing each rule predicate with regard to all other rule predicates in the ACL. In the end, all query intervals stored in variable *Output* are returned to the main function.

Algorithm 3 describes auxiliary Function `GetDecisionSpace`. It processes the query intervals of a query space, and returns the corresponding space of decisions (cf. Definition 8) with regard to the rules of an ACL. Each element of the decision space is returned to the main function as a set of three elements: a query interval ($Interval$), a set of the rules that match the query interval ($Rules$) and a decision associated to the query interval ($Decision$). Within the outer loop of Algorithm 3 (lines [2-24]), each query interval in set $QuerySpace$ is processed. A first nested loop (lines [5-15]) parses the ACL rules, and stores in $Inclusion$ (line 6) the overlapping portion of the query interval with $Predicate_r$. If $Inclusion$ is not

Algorithm 3 `GetDecisionSpace`($QuerySpace$, ACL)

```
1: $Output \leftarrow \emptyset$
2: for all $QueryInterval \in QuerySpace$ do
3:     $Interval \leftarrow QueryInterval$
4:     $Rules \leftarrow \emptyset$
5:     for all $r \in ACL$ do
6:         $Inclusion \leftarrow QueryInterval \wedge Predicate_r$
7:         $Exclusion \leftarrow QueryInterval \wedge \neg Predicate_r$
8:         if $Inclusion \neq$ False then
9:             $Rules \leftarrow Rules \cup \{r\}$
10:        end if
11:        if $Exclusion \neq$ False then
12:            $QueryInterval \leftarrow Exclusion$
13:        else
14:            //Leave inner loop
15:        end if
16:    $Decision \leftarrow \emptyset$
17:    for all $r \in Rules$ do
18:        if $Decision = \emptyset$ then
19:            $Decision \leftarrow Decision_r$
20:        else if $Decision \neq Decision_r$ then
21:            $Decision \leftarrow \emptyset$
22:            //Leave inner loop
23:        end if
24:    $Output \leftarrow Output \cup \{Interval, Rules, Decision\}$
25: return $Output$
```

`False` (lines [8-10]), then the corresponding rule is stored added to $Rules$, meaning that the predicate of such a rule matches the query interval. Similarly, the non-overlapping portion of the query interval with $Predicate_r$ is stored in $Exclusion$ (line 7), representing portions in $QueryInterval$ unmatched by rule r. If $Exclusion$ is not `False`, then some more portions in $QueryInterval$ still require being processed. Otherwise, all portions in $QueryInterval$ have already been processed, and the function moves to the following nested loop.

Algorithm 4 CompareDecisionSpaces(A,B)

```
1: for all $\{Interval_A, Rules_A, Decision_A\} \in A$ do
2:     for all $\{Interval_B, Rules_B, Decision_B\} \in B$ do
3:         if $Interval_A \wedge Interval_B =$ True then
4:             if $Decision_A \neq Decision_B$ then
5:                 return False
6:             end if
7:         end if
8: return True
```

The goal of the second nested loop (lines [17-23]) is to extract the decisions associated to the query intervals. Prior triggering the loop (line 16), the decision is initialized to an empty set, meaning that the decision associated to the query interval that is being processed has not yet been defined. Then, the loop is triggered to iterate over the rules previously derived during the first inner loop. A verification process to identify intervals with different treatments is conducted. Indeed, if the decision set by the first rule differs from the decision of other rules, the value of the decision is left as an empty set and the execution of the loop ends, meaning that the query interval has different treatment by the rules of the ACL[1]. When the loop ends, each triple $\{Interval, Rules, Decision\}$ is stored and returned to the main function when the outer loop ends.

Finally, Algorithm 4 describes Function `CompareDecisionSpaces`. It iterates over the query intervals of two decision spaces, and compares the value of their corresponding decisions. Whenever the decisions of two equivalent intervals differ, the function ends the execution and returns `False` to the main function. Otherwise, if there is full consistency, it returns `True`.

V. Applying the Verification Process

This section illustrates a practical application of our isofunctionality verification process. Assume the corporate network depicted in Figure 1. Two network segments are separated by a Firewall. The Firewall shall comply with the following policy requirements:

- All traffic from hosts in `192.168.10.64/26` to hosts in `172.16.50.0/24` are allowed, except those connections using a Telnet service (port 23).
- Only Web connections (port 80) from other hosts in `192.168.10.0/24` to hosts in `172.16.50.0/24` are allowed.

Assume we want to verify the isofunctionality of two different ACLs that implement the policy requirements described above. The two ACLs are written with respect to different constraints. For instance, `ACL A` represented in Table I, has a default rule set to *DENY ALL*, while `ACL B` represented in Table II has a default policy *ACCEPT ALL*.

The goal is to verify if `ACL A` and `ACL B` are isofunctional following the process described in Section IV. First, let us identify the attributes used to express the ACL rules. Three attributes can be identified: Source address (a_1), Destination address (a_2) and Destination port (a_3). Such attributes have the following domains:

- $\mathcal{V}_{a_1} = \{192.168.10.[0, 255]\}$
- $\mathcal{V}_{a_2} = \{172.16.50.[0, 255]\}$
- $\mathcal{V}_{a_3} = \{[0, 65535]\}$

For simplicity reasons, we assume that the elements of the decision set $\mathcal{D}$ hold only two access control decisions: ACCEPT and DENY. Therefore, $\mathcal{D}$ ={ACCEPT, DENY}.

TABLE I: `ACL A`

A_1:	{192.168.10.[64,127]	,172.16.50.[0.255]	,23	}, DENY
A_2:	{192.168.10.[64,127]	,172.16.50.[0.255]	,any	}, ACCEPT
A_3:	{192.168.10.[0.255]	,172.16.50.[0.255]	,80	}, ACCEPT
A_4:	{any	,any	,any	}, DENY

Phase 1: The first step of the process consists on extracting the query space of `ACL A`, $\mathcal{Q}_{\mathcal{A}}$, by applying Algorithm 2 (Function `GetQuerySpace`). Consider the first two rule predicates of `ACL A`:

[1]Notice that a straightforward modification of Algorithm 3 is possible, in case ACLs with different treatments is deemed necessary.

TABLE II: `ACL B`

B_1:	{192.168.10.[64,127]	,172.16.50.[0,255]	,23	}, DENY
B_2:	{192.168.10.[0,63]	,172.16.50.[0,255]	,[0,79]	}, DENY
B_3:	{192.168.10.[0,63]	,172.16.50.[0,255]	,[81,65535]	}, DENY
B_4:	{192.168.10.[128,255]	,172.16.50.[0,255]	,[0,79]	}, DENY
B_5:	{192.168.10.[128,255]	,172.16.50.[0,255]	,[81,65535]	}, DENY
B_6:	{any	,any	,any	}, ACCEPT

- $Predicate_{A_1} = a_1 \in \{192.168.10.[64, 127]\}$ $\wedge\ a_2 \in \{172.16.50.[0, 255]\} \wedge a_3 \in \{23\}$
- $Predicate_{A_2} = a_1 \in \{192.168.10.[64, 127]\}$ $\wedge\ a_2 \in \{172.16.50.[0, 255]\} \wedge a_3 \in [0, 65535]$

After initializing the variable $Predicates$ to $False$, the first iteration of Algorithm 2 starts. This leads to the following results:

- $Interval \leftarrow Predicate_{A_1} \wedge \neg Predicates$ $= Predicates_{A_1} \wedge True = Predicate_{A_1}$
- $Predicates \leftarrow Predicates \vee Predicate_{A_1}$ $= False \vee Predicate_{A_1} = Predicate_{A_1}$

At the end of the first iteration, the first query interval $Interval = Predicate_{A_1} = a_1 \in \{192.168.10.[64, 127]\} \wedge a_2 \in \{172.16.50.[0, 255]\} \wedge a_3 \in \{23\}$ is added to the query space, $\mathcal{Q}_{\mathcal{A}}$.

The second iteration leads to the following results:

- $Interval \leftarrow Predicate_{A_2} \wedge \neg Predicates$ $= Predicate_{A_2} \wedge \neg Predicate_{A_1}$
- $Predicates \leftarrow Predicates \vee Predicate_{A_2}$ $= Predicate_{A_1} \vee Predicate_{A_2}$

At the end of the second iteration, the query interval $Interval = a_1 \in \{192.168.10.[64, 127]\} \wedge a_2 \in \{172.16.50.[0, 255]\} \wedge (a_3 \in [0, 22] \vee a_3 \in [24, 65535])$ is added to $\mathcal{Q}_{\mathcal{A}}$.

By applying the same operations over all the rule predicates in `ACL A`, the query space $\mathcal{Q}_{\mathcal{A}}$ represented in Table III is obtained.

TABLE III: Query Space: $\mathcal{Q}_{\mathcal{A}}$

$Q_1 =$	$a_1 \in \{192.168.10.[64, 127]\} \wedge a_2 \in \{172.16.50.[0, 255]\} \wedge a_3 \in \{23\}$
$Q_2 =$	$a_1 \in \{192.168.10.[64, 127]\} \wedge a_2 \in \{172.16.50.[0, 255]\} \wedge (a_3 \in [0, 22] \vee a_3 \in [24, 65535])$
$Q_3 =$	$(a_1 \in \{192.168.10.[0, 63]\} \vee a_1 \in \{192.168.10.[128, 255]\}) \wedge a_2 \in \{172.16.50.[0, 255]\} \wedge a_3 \in \{80\}$
$Q_4 =$	$(a_1 \in \{192.168.10.[0, 63]\} \vee a_1 \in \{192.168.10.[128, 255]\}) \wedge a_2 \in \{172.16.50.[0, 255]\} \wedge (a_3 \in [0, 79] \vee a_3 \in [81, 65535])$

The second step of the process is to compute decision spaces $\mathcal{D}_{\mathcal{AA}}$ and $\mathcal{D}_{\mathcal{AB}}$ associated to $\mathcal{Q}_{\mathcal{A}}$ after applying, respectively, `ACL A` and `ACL B`. Consider the second query intervals of $\mathcal{Q}_{\mathcal{A}}$:

$$\begin{aligned} Q_2 = \ & a_1 \in \{192.168.10.[64, 127]\} \\ & \wedge\ a_2 \in \{172.16.50.[0, 255]\} \\ & \wedge\ (a_3 \in [0, 22] \vee a_3 \in [24, 65535]) \end{aligned}$$

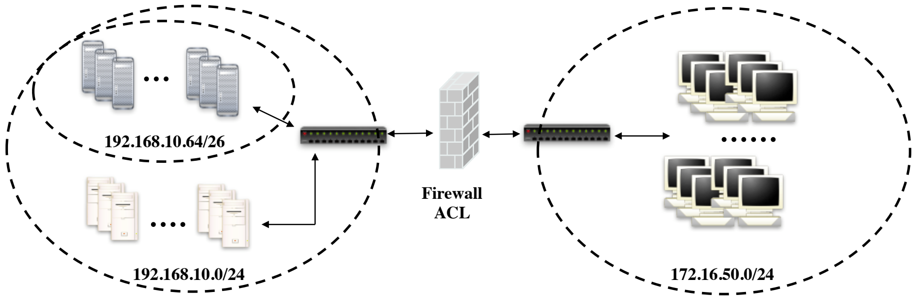

Fig. 1: Sample Network

To explain the execution of Algorithm 3, let us show its steps by using query interval Q_2 and `ACL A`. The outer loop variables are initialized as follows:

- $Interval \leftarrow Q_2$
- $Decision \leftarrow \emptyset$
- $Rules \leftarrow \emptyset$

The first iteration of the first inner loop starts with $r = A_1$. The following results are computed:

- $Inclusion \leftarrow Q_2 \wedge Predicate_{A_1} = False$
- $Exclusion \leftarrow Q_2 \wedge \neg Predicate_{A_1} = Q_2$

Notice that the overlapping portion is empty. Therefore, $Inclusion$ equals `False`, meaning that rule does not match any query of the query interval Q_2. Since the non-overlapping portion contains Q_2, a second iteration is conducted, following the statement $Q_2 \leftarrow Exclusion = Q_2$. The variables after this second iteration are as follows:

- $Inclusion \leftarrow Q_2 \wedge Predicate_{A_2} = Q_2$
- $Exclusion \leftarrow Q_2 \wedge \neg Predicate_{A_2} = False$

Now, since the overlapping portion is equal to Q_2, meaning that the predicate of the rule matches all the queries in Q_2. Therefore, rule A_2 is added to variable $Rules$.

The same rationale is applied until the last iteration of the first inner loop, in order to store all the rules matching the query interval Q_2. In the end, the only rule stored in variable $Rules$ is A_2.

The decision of the query interval Q_2 is set in the second inner loop. As $Rules$ contains only one rule, A_2, and $Decision = \emptyset$, then the decision of A_2 is affected to $Decision$. This leads to $Decision = \{\text{ACCEPT}\}$.

Following the same operations, decision spaces $\mathcal{D}_{\mathcal{AA}}$ and $\mathcal{D}_{\mathcal{AB}}$ (see Table IV) are computed.

TABLE IV: Decision Spaces $\mathcal{D}_{\mathcal{AA}}$ and $\mathcal{D}_{\mathcal{AB}}$

$\mathcal{Q}_{\mathcal{A}}$	*Rules* (A)	*Rules*(B)	*Decision* (A)	*Decision* (B)
Q_1	A_1	B_1	DENY	DENY
Q_2	A_2	B_6	ACCEPT	ACCEPT
Q_3	A_3	B_6	ACCEPT	ACCEPT
Q_4	A_4	B_2,B_3,B_4,B_5	DENY	DENY

In the first column of Table IV, we have the query interval extracted from `ACL A`. The second column contains the rules of `ACL A` that match the query interval in $\mathcal{Q}_{\mathcal{A}}$. The third column contains the rules of `ACL B` that match the query interval in $\mathcal{Q}_{\mathcal{A}}$. The fourth and fifth columns illustrate the decisions associated to each query interval, with respect to either `ACL A` or `ACL B`.

Notice that $\mathcal{D}_{\mathcal{AA}}$ and $\mathcal{D}_{\mathcal{AB}}$ are equivalent. Therefore, it can be concluded that any query issued to `ACL A` gets exactly the same decision result than if issued to `ACL B`.

Phase 2: Once ensured that all query intervals extracted from `ACL A` get exactly the same decision from `ACL A` or `ACL B`, Algorithm 1 starts the second phase of the isofunctionality verification process.

TABLE V: Query Space: $\mathcal{Q}_{\mathcal{B}}$

$$Q'_1 = \quad a_1 \in \{192.168.10.[64,127]\} \wedge a_2 \in \{172.16.50.[0,255]\} \wedge a_3 \in \{23\}$$

$$Q'_2 = \quad a_1 \in \{192.168.10.[0,63]\} \wedge a_2 \in \{172.16.50.[0,255]\} \wedge a_3 \in [0,79]$$

$$Q'_3 = \quad a_1 \in \{192.168.10.[0,63]\} \wedge a_2 \in \{172.16.50.[0,255]\} \wedge a_3 \in [81,65535]$$

$$Q'_4 = \quad a_1 \in \{192.168.10.[128,255]\} \wedge a_2 \in \{172.16.50.[0,255]\} \wedge a_3 \in [0,79]$$

$$Q'_5 = \quad a_1 \in \{192.168.10.[128,255]\} \wedge a_2 \in \{172.16.50.[0,255]\} \wedge a_3 \in [81,65535]$$

$$Q'_6 = \quad (a_1 \in \{192.168.10.[64,127]\} \wedge a_2 \in \{172.16.50.[0,255]\} \wedge (a_3 \in [0,22] \vee a_3 \in [24,65535])) \vee ((a_1 \in \{192.168.10.[0,63]\} \vee a_1 \in \{192.168.10.[128,255]\}) \wedge a_2 \in \{172.16.50.[0,255]\} \wedge a_3 \in \{80\})$$

The second phase starts by extracting the query space of `ACL B`, $\mathcal{Q}_{\mathcal{B}}$, by applying Function `GetQuerySpace`. Following the same rationale than the first phase, the query space $\mathcal{Q}_{\mathcal{B}}$ represented in Table V is extracted.

Then, the decision spaces $\mathcal{D}_{\mathcal{BB}}$ and $\mathcal{D}_{\mathcal{BA}}$ are computed, in conformity to $\mathcal{Q}_{\mathcal{B}}$ after applying receptively `ACL B` and `ACL A`. Table VI presents the decision spaces $\mathcal{D}_{\mathcal{BB}}$ and $\mathcal{D}_{\mathcal{BA}}$.

Notice that $\mathcal{D}_{\mathcal{BB}}$ and $\mathcal{D}_{\mathcal{BA}}$ are again equivalent. This means that any query issued to ACL `B` gets exactly the same decision result than if issued to ACL `A`. Therefore, we can finally conclude that `ACL A` and `ACL B` are isofunctional.

TABLE VI: Decision Spaces $\mathcal{D}_{\mathcal{BB}}$ and $\mathcal{D}_{\mathcal{BA}}$

$\mathcal{Q}_{\mathcal{B}}$	*Rules* (B)	*Rules*(A)	*Decision* (B)	*Decision* (A)
Q'_1	B_1	A_1	ACCEPT	ACCEPT
Q'_2	B_2	A_4	ACCEPT	ACCEPT
Q'_3	B_3	A_4	ACCEPT	ACCEPT
Q'_4	B_4	A_4	ACCEPT	ACCEPT
Q'_5	B_5	A_4	ACCEPT	ACCEPT
Q'_6	B_6	A_2, A_3	DENY	DENY

VI. Related Work

The verification process presented in this paper is a complementary method for the management of ACL analysis that require from isofunctionality verification. Analysis of network ACLs is a well-known topic and several studies have been carried out. Existing solutions include the discovery and resolution of anomalies of already deployed ACLs, and verification of ACLs prior enforcement.

Studies focusing on the analysis of already deployed network ACLs typically aim at discovering and fixing conflicts and redundancies [1]. Chomsiri and Pornavalai [6] propose a method of analyzing ACL rules using Relational Algebra to discover all the anomalies and relations between rules. Garcia-Alfaro et al. [7], [8] propose the discovery and removal of inconsistencies among distributed security policies enforced over firewalls and intrusion detection systems. Similarly, Hu et al. [4], [5] propose an anomaly management framework that permits the systematic detection and resolution of ACL anomalies by adopting a rule-based segmentation technique to identify ACL anomalies and derive effective resolutions.

Solutions for the verification of ACL rules at higher abstraction layers have also been proposed. Liu and Gouda [3] propose some firewall verification methods to crosscheck firewall ACLs and properties. Nelson et al. [19] propose a configuration analysis tool that allows users to check policies against security goals. Laborde et al. [20] address the problem of deploying security policies with regard to generic system functionalities. Finally, Yang and Lam [21] propose a tool (Atomic Predicate Verifier) to verify network properties using a set of atomic predicates extracted from ACL rules.

VII. Conclusion

The management of network Access Control Lists (ACLs) refers to the task of authoring and maintaining security rules associated to network equipment, such as routers, switches and firewalls. Methods and tools conceived to assist during those processes must verify if the ACLs distributed among different equipment are functionally equivalent. In this paper, we have addressed such a verification process. We have presented a formal method to verify when two ACLs are isofunctional. We have also illustrated our proposal over a practical example. Future work aims to extend the concept of isofunctionality to general-purpose access control models.

Acknowledgements — Authors acknowledge support from the EC Framework Program, under the PANOPTESEC project (GA 610416), as well as Spanish Ministry of Science and Innovation (project TIN2011-27076-C03-02 CO-PRIVACY).

References

[1] H. Hamed and E. Al-Shaer, "Taxonomy of conflicts in network security policies," *Communications Magazine, IEEE*, vol. 44, no. 3, pp. 134–141, March 2006.

[2] ——, "Dynamic rule-ordering optimization for high-speed firewall filtering," in *2006 ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '06, 2006, pp. 332–342.

[3] M. G. Gouda and A. X. Liu, "Structured firewall design," *Computer Networks*, vol. 51, no. 4, pp. 1106–1120, 2007.

[4] H. Hu, G. Ahn, and K. Kulkarni, "Detecting and resolving firewall policy anomalies," *IEEE Trans. Dependable Sec. Comput.*, vol. 9, no. 3, pp. 318–331, 2012.

[5] ——, "FAME: a firewall anomaly management environment," in *3rd ACM Workshop on Assurable and Usable Security Configuration, SafeConfig 2010, Chicago, IL, USA, October 4, 2010*, 2010, pp. 17–26.

[6] T. Chomsiri and C. Pornavalai, "Firewall rules analysis," in *2006 International Conference on Security & Management, SAM 2006, Las Vegas, Nevada, USA, June 26-29, 2006*, 2006, pp. 213–219.

[7] J. Garcia-Alfaro, F. Cuppens, and N. Cuppens-Boulahia, "Analysis of policy anomalies on distributed network security setups," in *11th European Symposium on Research in Computer Security (ESORICS 2006), Hamburg, Germany, September 18-20, 2006*, 2006, pp. 496–511.

[8] J. Garcia-Alfaro, N. Boulahia-Cuppens, and F. Cuppens, "Complete analysis of configuration rules to guarantee reliable network security policies," *International Journal of Information Security*, vol. 7, no. 2, pp. 103–122, 2008.

[9] J. García-Alfaro, F. Cuppens, N. Cuppens-Boulahia, S. M. Perez, and J. Cabot, "Management of stateful firewall misconfiguration," *Computers & Security*, vol. 39, pp. 64–85, 2013.

[10] J. Vaidya, V. Atluri, and Q. Guo, "The role mining problem: finding a minimal descriptive set of roles," in *12th ACM symposium on Access control models and technologies*. ACM, 2007, pp. 175–184.

[11] E. J. Coyne, "Role engineering," in *ACM Workshop on Role-based access control*. ACM, 1996, p. 4.

[12] G. Misherghi, L. Yuan, Z. Su, C. Chuah, and H. Chen, "A general framework for benchmarking firewall optimization techniques," *IEEE Transactions on Network and Service Management*, vol. 5, no. 4, pp. 227–238, 2008.

[13] M. R. Lyu and L. K. Y. Lau, "Firewall security: Policies, testing and performance evaluation," in *24th International Computer Software and Applications Conference (COMPSAC 2000), 25-28 October 2000, Taipei, Taiwan*, 2000, pp. 116–121.

[14] A. Tapdiya and E. W. Fulp, "Towards optimal firewall rule ordering utilizing directed acyclical graphs," in *18th International Conference on Computer Communications and Networks (ICCCN 2009), San Francisco, California, August 3-6, 2009*, 2009, pp. 1–6.

[15] E. W. Fulp, "Optimization of network firewall policies using directed acyclic graphs," in *IEEE Internet Management Conference*, 2005, pp. 10–15.

[16] Firewall Builder, Last Access: 2014, available at http://fwbuilder.org.

[17] Athena Firepac, Last Access: 2014, available at http://www.solarwinds.com/athena-security-products.aspx.

[18] S. Preda, N. Cuppens-Boulahia, F. Cuppens, and L. Toutain, "Architecture-aware adaptive deployment of contextual security policies," in *5th International Conference on Availability, Reliability and Security, February 2010, Poland*, 2010, pp. 87–95.

[19] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The margrave tool for firewall analysis." in *24th USENIX Large Installation System Administration Conference (LISA 2010)*, 2010.

[20] R. Laborde, M. Kamel, F. Barrère, and A. Benzekri, "Implementation of a formal security policy refinement process in wbem architecture," *Journal of Network and Systems Management*, vol. 15, no. 2, pp. 241–266, 2007.

[21] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *2013 21st IEEE International Conference on Network Protocols, ICNP 2013, Göttingen, Germany, October 7-10, 2013*, 2013, pp. 1–11.