Complete analysis of configuration rules to guarantee reliable network security policies

J. G. Alfaro · N. Boulahia-Cuppens · F. Cuppens

Abstract The use of different network security components, such as *firewalls* and *network intrusion detection systems* (NIDSs), is the dominant method to monitor and guarantee the security policy in current corporate networks. To properly configure these components, it is necessary to use several sets of security rules. Nevertheless, the existence of anomalies between those rules, particularly in distributed multi-component scenarios, is very likely to degrade the network security policy. The discovery and removal of these anomalies is a serious and complex problem to solve. In this paper, we present a complete set of mechanisms for such a management.

Keywords Network security · Firewalls · Intrusion Detection systems · Policy anomalies

1 Introduction

Generally, once a security administrator has specified a security policy, he or she aims to enforce it in the information system to be protected. This enforcement consists in distributing the security rules expressed in this policy over different security components of the information system—such as firewalls, intrusion detection systems (IDSs), intrusion prevention systems (IPSs), proxies, etc.—both at application, system, and network level. This implies cohesion of the security functions supplied by these components. In other words, security rules deployed over the different components must be consistent, not redundant and, as far as possible, optimal.

An approach based on a formal security policy refinement mechanism (using for instance abstract machines grounded on set theory and first order logic) ensures cohesion, completeness and optimization as built-in properties. Unfortunately, in most cases, such an approach has not a wide following and the policy is more often than not empirically deployed based on security administrator expertise and flair. It is then advisable to analyze the security rules deployed to detect and correct some policy anomalies—often referred to in the literature as *intra- and inter-configuration anomalies* [7]. These anomalies might be the origin of security holes and/or difficulty of the intrusion prevention and detection processes. Firewalls [11] and network intrusion detection systems (NIDSs) [21] are the most commonly used security components and, in this paper, we focus particularly on their security rules.

Firewalls are prevention devices ensuring access control. They manage the traffic between the public network and the private network zones on one hand and between private zones in the local network on the other hand. Undesirable traffic is blocked or re-routed by such a component. NIDSs are detection devices ensuring a monitoring role. They are components that monitor the traffic and generate alerts in the case of suspicious traffic. The attributes used to block or to generate alerts are almost the same. The challenge, when these two kinds of components coexist in the security architecture of an information system is then to avoid inter-configuration anomalies.

In [13,14], we presented an audit process to manage intra-firewall policy anomalies, in order to detect and remove anomalies within the set of rules of a given firewall. This audit

J. G. Alfaro · N. Boulahia-Cuppens · F. Cuppens
GET-ENST Bretagne, 02, rue de la Châtaigneraie, CS 17607, 35576 Cesson Sévigné Cedex, France
e-mail: nora.cuppens@enst-bretagne.fr

J. G. Alfaro (✉)
UOC, Rambla Poble Nou 156, 08018 Barcelona, Spain
e-mail: joaquin.garcia-alfaro@acm.org

F. Cuppens
e-mail: frederic.cuppens@enst-bretagne.fr

process is based on the existence of relationships between the condition attributes of the filtering rules—such as coincidence, disjunction, and inclusion—and proposes a transformation process which derives from an initial set of rules (potentially misconfigured) to an equivalent one which is completely free of errors. Furthermore, the resulting rules are totally disjoint, i.e., the ordering of rules is no longer relevant.

In this paper we extend our proposal for detecting and removing intra-firewall policy anomalies to a distributed setup where both firewalls and NIDSs might be in charge of the network security policy. In this way, and assuming that the role of both prevention and detection of network attacks is assigned to several components, our objective is to avoid intra and inter-component anomalies between filtering and alerting rules. The proposed approach is based on the similarity between the parameters of a filtering rule and those of an alerting rule. We can therefore check whether there are errors in those configurations regarding the policy deployment over each component which matches the same traffic.

The advantages of our approach are the following. First, as opposite to the closest related work shown in Sect. 2, our approach not only considers the analysis of relationships between rules two by two but also a complete analysis of the whole set of rules. This way, those conflicts due to the union of rules that are not detected by other proposals (such as [5,6,16]) are properly discovered by our intra- and inter-component algorithms. Second, after applying our intra-component algorithms the resulting rules of each component are totally disjoint, i.e., the ordering of rules is no longer relevant. Hence, one can perform a second rewriting of rules in a *close* or *open* manner, generating a configuration that only contains *deny* (or *alert*) rules if the component default policy is open, and *accept* (or *pass*) rules if the default policy is close (cf. Sect. 4.5). Third, the use of a network model to determine topological properties better defines all the set of anomalies studied in the related work. Furthermore the lack of this model in other approaches, such as [5–7], may lead to inappropriate decisions.

The rest of this paper is organized as follows. Section 2 starts with an analysis of some related work. Section 3 introduces a network model that is further used in Sects. 4 and 5 when presenting, respectively, our intra and inter-component anomaly classifications and algorithms. Section 6 overviews a first implementation of our proposals in order to validate its performance over real multi-component scenarios. Section 7 closes the paper giving some conclusions and further work.

2 Related work

A first approach to addressing our problem domain is the use of refinement mechanisms. In this way, we can perform a top-down deployment of rules by unfolding a global set of security policies into the configurations of several components and guaranteeing that those deployed configurations are free of anomalies. In [8], for example, the authors present a refinement mechanism that uses a formal model for the generation of filtering rules by transforming general rules into specific configuration rules. Indeed, the authors propose the use of roles to better define network capabilities, and the use of an inheritance mechanism through a hierarchy of entities to automatically generate permissions and prohibitions. However, their work does not fix, from our point of view, clear semantics; and their concept of role becomes, moreover, ambiguous. A second refinement approach based on the concept of roles is presented in [17]. However, and although the authors claim that their work is based on the Role Base Access Control (RBAC) model [25], their specification of network entities, roles, and permission assignments are not rigorous and does not fit any reality. Most of these limitations are solved in the approach presented in [15], where a global set of rules based on the Organization Based Access Control (OrBAC) model [1] are further deployed into specific firewall configuration files through a transformation process. Unfortunately, and although we think this approach heads in the right direction, we consider that the single use of refinement mechanisms is not always enough. Generally, administrators are reluctant to set up from scratch a whole network security policy, and prefer recycling existing configurations.

A second manner to address our problem domain is through the use of automatic network support tools intended for the creation of configurations for security devices. Firewall Builder [18], for example, provides a common interface to specify a network access control policy and then this policy is automatically translated into various firewall configuration languages, such as *netfilter* [27], *ipfilter* [23], or Cisco PIX [10]. Similarly, the Cisco Security Manager [12] is a commercial support tool designed to manage security policy deployments on heterogeneous networks based on Cisco devices. However, we consider that these two solutions do not offer a semantic model rich enough to express complete security policies; and, although they offer some routines for the discovery of conflicts between rules, such functionality requires the administrator's assistance and only simple redundancy that corresponds to trivial equality or inclusion between zones is detected. A more complete taxonomy of anomalies (as the one we present in this paper) should be addressed by these tools.

The closest works to ours are those of [2,5–7,16,19,28] which provide means to directly manage the discovery of anomalies from the components' configurations. The authors in [2] consider that, in a configuration set, two rules are in conflict when the first rule in order matches some packets that match the second rule, and the second rule also matches some of the packets that match the first rule. This approach is very limited since it just detects a particular case of ambiguity within a single component configuration. Furthermore, it does not provide detection in multiple-component configurations. In [16], two cases of anomalies are considered. First, a rule R_j is defined as *backward redundant* iff there exists another rule R_i with higher priority in order such that all the packets that match rule R_j also match rule R_i. Second, a rule R_i is defined as *forward redundant* iff there exists another rule R_j with the same decision and less priority in order such that the following conditions hold: (1) all the packets that match R_i also match R_j; (2) for each rule R_k between R_i and R_j, and that matches all the packets that also match rule R_i, R_k has the same decision as R_i. Although this approach seems to head in the right direction, we consider it as incomplete, since it does not detect all the possible cases of intra-component anomalies (as we do in this paper). For instance, given the following set of filtering rules (where each rule is in the form $R_i : condition_i \rightarrow decision_i$, being i the relative position of the rule within the set of rules, $decision_i$ a boolean expression in $\{accept, deny\}$, and $condition_i$ the condition attribute *source zone*—*szone* for short):

$R_1 : szone \in [10, 50] \rightarrow deny$
$R_2 : szone \in [40, 70] \rightarrow accept$
$R_3 : szone \in [50, 80] \rightarrow accept$

and since R_2 comes after R_1, rule R_2 only applies over the interval [51, 70]—i.e., R_2 is not necessary, since, if we remove this rule from the configuration, the filtering policy does not change. The detection proposal defined in [16] cannot detect the redundancy of rule R_2 within the configuration of such a given firewall. A similar but more complete approach to detect those anomalies is presented in [19]. However, neither [16] nor [19] provide detection on multiple-component configurations.

The authors of [5–7] propose in their work an efficient set of algorithms to detect policy anomalies in both single- and multi-firewall configuration setups. Nonetheless, we also consider their approach as incomplete. First, their intra- and inter-component discovery approach is not complete since, given a single- or multiple-component security policy, their detection algorithms are based on the analysis of relationships between rules two by two. This way, errors due to the union of rules are not explicitly considered (as our approach does). For example, the following set of rules:

$R_1 : szone \in [10, 50] \rightarrow accept$
$R_2 : szone \in [40, 90] \rightarrow accept$
$R_3 : szone \in [30, 80] \rightarrow deny$

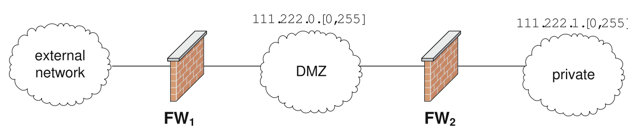
may lead their discovery algorithms to inappropriate decisions. The approach defined in [5] cannot detect that rule R_3 will never be applied due to the union of rules R_1 and R_2. Just a correlation signal—that is obviously a weaker signal than a shadowing one—would be labeled.

Although in [6] the authors pointed out this problem, claiming that they break down the initial set of rules into an equivalent set of rules free of overlaps between rules, no specific algorithms for solving it have been provided in [5–7]. From our point of view, the proposal presented in [28] best addresses this limitation, although it also presents some limitations. For instance, giving again the following set of rules:

$R_1 : szone \in [10, 50] \rightarrow deny$
$R_2 : szone \in [40, 70] \rightarrow accept$
$R_3 : szone \in [50, 80] \rightarrow accept$

the proposal presented in [28] reports two partial redundancies (respectively, between rules R_1,R_2; and rules R_2,R_3), instead of the full redundancy of rule R_2.

The inter-component discovery presented in [5–7], moreover, considers as anomalies some situations that, from our point of view, must be tolerated to avoid inconsistent decisions between components used in the same policy to control or monitor the access to different zones. For instance, given the following scenario (where the condition attributes of both rule $FW_1\{R_1\}$ and $FW_2\{R_1\}$ are, respectively, (p)rotocol, (s)ource zone, (d)estination zone, and destination port—*dport* for short):



$FW_1\{R_1\} : p = tcp \wedge s \in any \wedge d \in 111.222.1.0/24 \wedge dport = 80 \rightarrow deny$

$FW_2\{R_1\} : p = tcp \wedge s \in 111.222.0.0/24 \wedge d \in 111.222.1.0/24 \wedge dport = 80 \rightarrow deny$

their algorithms will inappropriately report a redundancy anomaly between filtering rules $FW_1\{R_1\}$ and $FW_2\{R_1\}$. This is because rule $FW_1\{R_1\}$ matches every packet that also $FW_2\{R_1\}$ does. As a consequence, [5] considers rule $FW_2\{R_1\}$ as redundant since packets denied by this rule are already denied by rule $FW_1\{R_1\}$. However, this conclusion is not appropriate because rule $FW_1\{R_1\}$ applies to packets from the external zone to the private zone whereas rule $FW_2\{R_1\}$ applies to packets from the DMZ zone to the private zone. So, rule $FW_2\{R_1\}$ is useful and cannot be removed. Though in [5,6] the authors claim that their analysis technique marks every rule that is used on a network path, no specific algorithms have been provided for doing so. The main advantage of our proposal over their approach is that it includes a model of the traffic which flows through each component. We consider this is necessary to draw the right conclusion in this case.

Finally, although in both [7,28] the authors consider their work as sufficiently general to be used for verifying many other filtering based security policies such as intrusion detection and prevention systems, no specific mechanisms have been provided for doing so.

3 Network model and topology properties

The purpose of our network model is to determine which components within the network are traversed by a given packet, knowing its source and destination. It is defined as follows. First, and concerning the traffic flowing from two different zones of the distributed policy scenario, we may determine the set of components that are traversed by this flow. Regarding the scenario shown in Fig. 1, for example, the set of components traversed by the network traffic flowing from zone $external\ network$ to zone $private_3$ equals $[C_1, C_2, C_4]$, and the set of components traversed by the network traffic flowing from zone $private_3$ to zone $private_2$ equals $[C_4, C_2, C_3]$.

Let C be a set of components and let Z be a set of zones. We assume that each pair of zones in Z are mutually disjoint, i.e., if $z_i \in Z$ and $z_j \in Z$ then $z_i \cap z_j = \emptyset$. We then define the predicate $connected(c_1, c_2)$ as a symmetric and anti-reflexive function which becomes $true$ when there exists, at least, one interface connecting component c_1 to component c_2. On the other hand, we define the predicate $adjacent(c, z)$ as a relation between components and zones which becomes $true$ when the zone z is interfaced to component c. Referring to Fig. 1, we can verify that predicates $connected(C_1, C_2)$ and $connected(C_1, C_3)$, as well as $adjacent(C_1, DMZ)$, $adjacent(C_2, private_1)$, $adjacent(C_3, DMZ)$, and so on, become $true$. We then define the set of paths, P, as follows. If $c \in C$ then $[c] \in P$ is an atomic path. Similarly, if $[p\,.\,c_1] \in P$ (be "." a concatenation functor) and $c_2 \in C$, such that $c_2 \notin p$ and $connected(c_1, c_2)$, then $[p\,.\,c_1\,.\,c_2] \in P$. This way, we can notice that, concerning Fig. 1, $[C_1, C_2, C_4] \in P$ and $[C_1, C_3] \in P$.

Let us now define a set of functions related to the order between paths. We first define functions $first$, $last$, and the order functor between paths. We define function $first$ from P in C such that if p is a path, then $first(p)$ corresponds to the first component in the path. Conversely, we define function $last$ from P in C such that if p is a path, then $last(p)$ corresponds to the last component in the path. We then define the order functor between paths as $p_1 \leq p_2$, such that path p_1 is shorter than p_2, and where all the components within p_1 are also within p_2. We also define the predicates $isFirewall(c)$ and $isNIDS(c)$ which become $true$ when the component c is, respectively, a firewall or a NIDS.

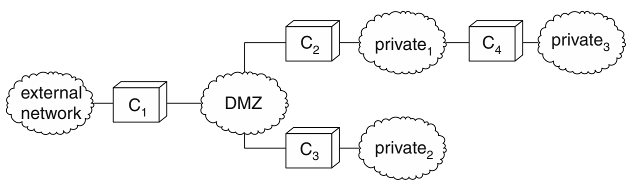


Fig. 1 Simple distributed policy setup

Two additional functions are $route$ and $minimal\_route$. We first define function $route$ from Z to Z in 2^P, such that $p \in route(z_1, z_2)$ iff the path p connects zone z_1 to zone z_2. Formally, we define that $p \in route(z_1, z_2)$ iff the predicates $adjacent(first(p), z_1)$ and $adjacent(last(p), z_2)$ become $true$. Similarly, we define $minimal\_route$ (or MR for short) from Z to Z in 2^P, such that $p \in MR(z_1, z_2)$ iff the following conditions hold: (1) $p \in route(z_1, z_2)$; (2) there does not exist $p' \in route(z_1, z_2)$ such that $p' < p$. Regarding Fig. 1, we can verify that the minimal route from zone $private_3$ to zone $private_2$ equals $[C_4, C_2, C_3]$, i.e., $MR(private_3, private_2) = \{[C_4, C_2, C_3]\}$. We finally conclude by defining the predicate $affects(Z, A_c)$ as a boolean expression which becomes $true$ when there is, at least, an element $z \in Z$ such that the configuration of z is vulnerable to the attack category $A_c \in V$, where V is a vulnerability set built from a vulnerability database, such as CVE/CAN [20] or OSVDB [22].

4 Intra-component classification and algorithms

In this section we present our set of intra-component audit algorithms, whose main objective is the complete discovery and removal of policy anomalies that could exist in a single component policy, i.e., to discover and warn the security officer about potential anomalies within the configuration rules of a given component.

For our work, we define the security rules of both firewalls and NIDSs as filtering and alerting rules, respectively. In turn, both filtering and alerting rules are specific cases of a more general configuration rule, which *typically* defines a *decision* (such as *deny*, *alert*, *accept*, or *pass*) that applies over a set of *condition* attributes, such as *protocol*, *source zone* (or *szone*), *destination zone* (or *dzone*), *classification*, and so on. We define a general configuration rule as follows:

$$R_i : \{condition_i\} \rightarrow decision_i \qquad (1)$$

where i is the relative position of the rule within the set of rules, $\{condition_i\}$ is the conjunctive set of condition attributes such that $\{condition_i\}$ equals $A_1 \wedge A_2 \wedge \cdots \wedge A_p$—being p the number of condition attributes of the given rule—and *decision* is a boolean value in $\{true, false\}$.

We shall notice that, for our work, the decision of a filtering rule will be positive (*true*) when it applies to a specific value related to *deny* (or *filter*) the traffic it matches, and will be negative (*false*) when it applies to a specific value related to *accept* (or *ignore*) the traffic it matches. Similarly, the decision of an alerting rule will be positive (*true*) when it applies to a specific value related to *alert* (or *warn*) about the traffic it matches, and will be negative (*false*) when it applies to a specific value related to pass (or *ignore*) the traffic it matches.

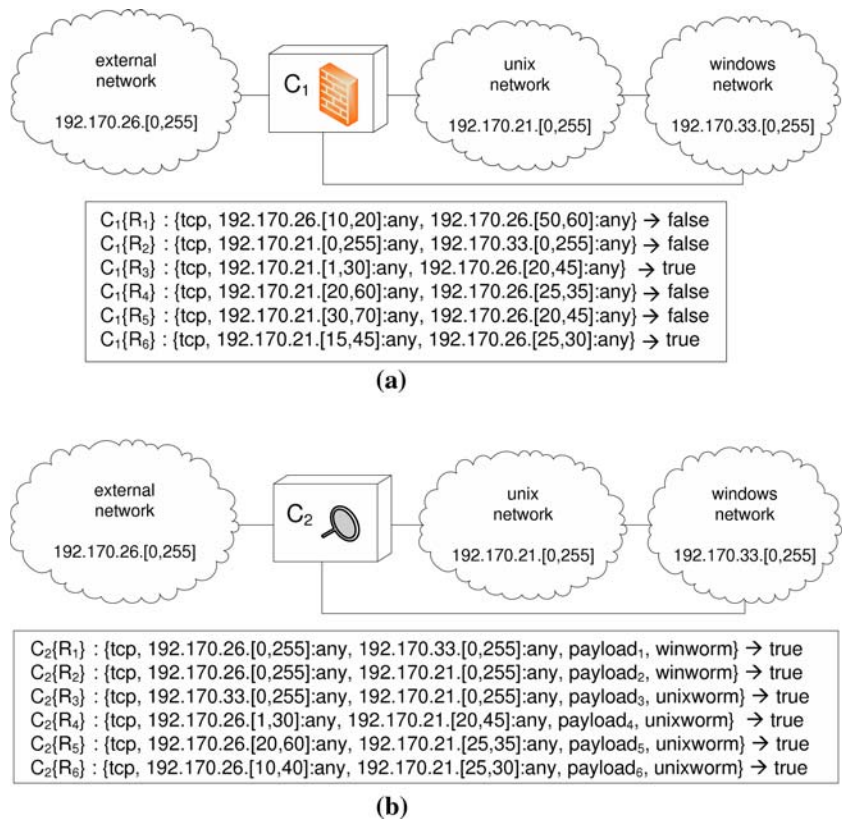


Fig. 2 Example of filtering and alerting policies. **a** Example scenario of a filtering policy. **b** Example scenario of an alerting policy

Let us continue this section by classifying the complete set of anomalies that can occur within a single component configuration. An example for each anomaly will be illustrated through the sample scenario shown in Fig. 2.

Intra-component shadowing A configuration rule R_i is shadowed in a set of configuration rules R when such a rule never applies because all the packets that R_i may match, are previously matched by another rule, or combination of rules, with higher priority. Regarding Fig. 2, rule $C_1\{R_6\}$ is shadowed by the overlapping of rules $C_1\{R_3\}$ and $C_1\{R_5\}$.

Intra-component redundancy A configuration rule R_i is redundant in a set of configuration rules R when the following conditions hold: (1) R_i is not shadowed by any other rule or set of rules; (2) when removing R_i from R, the security policy does not change. For instance, referring to Fig. 2, rule $C_1\{R_4\}$ is redundant, since the overlapping between rules $C_1\{R_3\}$ and $C_1\{R_5\}$ is equivalent to the policy of rule $C_1\{R_4\}$.

Intra-component irrelevance A configuration rule R_i is irrelevant in a set of configuration rules R if one of the following conditions holds:

(1) Both source and destination addresses are within the same zone. For instance, rule $C_1\{R_1\}$ is irrelevant since the source of this address, $external\ network$, as well as its destination, is the same.

(2) The component is not within the minimal route that connects the source zone, concerning the irrelevant rule which causes the anomaly, to the destination zone. Hence, the rule is irrelevant since it matches traffic which does not flow through this component. Rule $C_1\{R_2\}$, for example, is irrelevant since component C_1 is not in the path which corresponds to the minimal route between the source zone $unix\ network$ to the destination zone $windows\ network$.

(3) The component is a NIDSs, i.e., the predicate $isNIDS(c)$ (cf. Sect. 3) becomes $true$, and, at least, one of the condition attributes in R_i is related with a classification of attack A_c which does not affect the destination zone of such a rule—i.e., the predicate affects (z_d, A_c) becomes $false$. Regarding Fig. 2, we can see that rule $C_2\{R_2\}$ is irrelevant since the nodes in the destination zone $unix\ network$ are not affected by vulnerabilities classified as $winworm$.

4.1 Intra-component Algorithms

Our proposed audit process is a way of alerting the security officer in charge of the network about these configuration errors, as well as to remove all the useless rules in the initial firewall configuration. The data to be used for the detection process is the following. A set of rules R as a list of initial size n, where n equals $count(R)$, and where each element is an associative array with the strings $condition$, *decision*, *shadowing*, *redundancy*, and *irrelevance* as keys to access each necessary value.

For reasons of clarity, we assume one can access a linked-list through the operator R_i, where i is the relative position regarding the initial list size—$count(R)$. We also assume one can add new values to the list as any other normal variable does ($element \leftarrow value$), as well as remove elements through the addition of an empty set ($element \leftarrow \emptyset$). The internal order of elements from the linked-list R keeps with the relative ordering of rules.

Each element $R_i[condition]$ is a boolean expression over p possible attributes. To simplify, we only consider the following attributes: *szone* (source zone), *dzone* (destination zone), *sport* (source port), *dport* (destination port), *protocol*, and *attack_class*—or A_c for short—which will be empty when the component is a firewall. In turn, each element $R_i[decision]$ is a boolean variable whose values are in $\{true, false\}$. Each element $R_i[type]$ is a boolean variable whose values are in $\{filtering, alerting\}$. Finally, elements $R_i[shadowing]$, $R_i[redundancy]$, and $R_i[irrelevance]$ are boolean variables in $\{true, false\}$—which will be initialized to $false$ by default.

We split the whole process into four different algorithms. The first algorithm (cf. Algorithm 1) is an auxiliary function whose input are two rules, A and B. Once executed, this auxiliary function returns a further rule, C, whose set of condition attributes is the exclusion of the set of conditions from A over B. In order to simplify the representation of this algorithm, we use the notation A_i as an abbreviation of the variable $A[condition][i]$, and the notation B_i as an abbreviation of the variable $B[condition][i]$—where i in $[1, p]$.

The second algorithm (cf. Algorithm 2) is a boolean function in $\{true, false\}$ which applies the necessary verifications to decide whether a rule r is irrelevant for the configuration of a component c. To properly execute such an algorithm, let us define $source(r)$ as a function in Z such that $source(r) = szone$, and $dest(r)$ as a function in Z such that $dest(r) = dzone$.

The third algorithm (cf. Algorithm 3) is a boolean function in $\{true, false\}$ which, in turn, applies the transformation *exclusion* (cf. Algorithm 1) over a set of configuration rules to check whether the rule obtained as a parameter is potentially redundant.

The last algorithm (cf. Algorithm 4) performs the whole process of detecting and removing the complete set of intra-component anomalies. This process is split into three different phases. During the first phase, a set of shadowing rules are detected and removed from a top-bottom scope, by iteratively applying Algorithm 1—when the decision field of the two rules is different. Let us notice that this stage of detecting and removing shadowed rules is applied before the detection and removal of proper redundant and irrelevant rules.

The resulting set of rules is then used when applying the second phase, also from a top-bottom scope. This stage is performed to detect and remove proper redundant rules, through an iterative call to Algorithm 3 (i.e., *testRedundancy*), as well as to detect and remove all the further shadowed rules remaining during the latter process. Finally, during a third phase the whole set of non-empty rules is analyzed in order to detect and remove irrelevance, through an iterative call to Algorithm 2 (i.e., *testIrrelevance*).

We give in the following sections an outlook on applying these four algorithms over some representative examples, as well as a proof of their correctness, and an analysis of their complexity.

Algorithm 1: exclusion(B,A)

```
C[condition] ← ∅;
C[shadowing] ← false;
C[redundancy] ← false;
C[irrelevance] ← false;
C[decision] ← B[decision];
C[type] ← B[type];
forall the elements of A[condition] and B[condition] do
    if ((A1 ∩ B1) ≠ ∅ and (A2 ∩ B2) ≠ ∅
    and ... and (Ap ∩ Bp) ≠ ∅) then
        C[condition] ← C[condition] ∪
        {(B1 − A1) ∧ B2 ∧ ... ∧ Bp,
        (A1 ∩ B1) ∧ (B2 − A2) ∧ ... ∧ Bp,
        (A1 ∩ B1) ∧ (A2 ∩ B2) ∧ (B3 − A3) ∧ ... ∧ Bp,
        ...
        (A1 ∩ B1) ∧ ... ∧ (Ap−1 ∩ Bp−1) ∧ (Bp − Ap)};
    else
        C[condition] ← (C[condition] ∪ B[condition]);
return C;
```

Algorithm 2: testIrrelevance(c,r)

```
zs ← source (r);
zd ← dest (r);
if (zs = zd) then
    warning ("First case of irrelevance");
else if zs ≠ zd then
    p ← MR (zs,zd);
    if c ∉ p then
        warning ("Second case of irrelevance");
    else if ¬empty(r[Ac]) and ¬affects(zd, r[Ac])
    then
        warning ("Third case of irrelevance");
    else return false;
return true;
```

Algorithm 3: testRedundancy(R,r)

```
i ← 1;
temp ← r;
while i ≤ count(R) do
    temp ← exclusion(temp, Ri);
    if temp[condition] = ∅ then
        return true;
    i ← (i + 1);
return false;
```

Algorithm 4: intra-component-audit(c,R)

```
n ← count(R);
/*Phase 1*/
for i ← 1 to (n − 1) do
    for j ← (i + 1) to n do
        if Ri[decision] ≠ Rj[decision] then
            Rj ← exclusion (Rj,Ri);
            if Rj[condition] = ∅ then
                warning ("Shadowing");
                Rj[shadowing] ← true;

/*Phase 2*/
for i ← 1 to (n − 1) do
    Ra ← {rk ∈ R | n ≥ k > i and
        rk[decision] = ri[decision]};
    if testRedundancy (Ra,Ri) then
        warning ("Redundancy");
        Ri[condition] ← ∅;
        Ri[redundancy] ← true;
    else
        for j ← (i + 1) to n do
            if Ri[decision]=Rj[decision] then
                Rj ←exclusion (Rj,Ri);
                if Rj[condition] = ∅) then
                    warning ("Shadowing");
                    Rj[shadowing] ← true;

/*Phase 3*/
for i ← 1 to n do
    if Ri[condition] ≠ ∅ then
        if testIrrelevance (c,Ri) then
            Ri[condition] ← ∅;
            Ri[irrelevance] ← true;
```

4.2 Applying the Intra-component Algorithms

Let us start this section by showing how can we apply function *exclusion* (Algorithm 1) over a set of two rules R_i and R_j, each one of them with two condition attributes ($szone$ and $dzone$), and where R_j has less priority than R_i.

In this first example,

$$R_i[condition] = (szone \in [80, 100]) \wedge (dzone \in [1, 50])$$
$$R_j[condition] = (szone \in [1, 50]) \wedge (dzone \in [1, 50])$$

since $(szone \in [1, 50]) \cap (szone \in [80, 100]) = \emptyset$, the condition attributes of rules R_i and R_j are completely independent. Thus, the applying of $exclusion(R_j, R_i)$ is equal to $R_j[condition]$.

The following three examples show the same execution over a set of condition attributes with different cases of conflict. A first case is the following:

$$R_i[condition] = (szone \in [1, 60]) \wedge (dzone \in [1, 30])$$
$$R_j[condition] = (szone \in [1, 50]) \wedge (dzone \in [1, 50])$$

where there is a main overlap of attribute $szone$ from $R_i[condition]$ which completely excludes the same attribute on $R_j[condition]$. Then, there is a second overlap of attribute $dzone$ from $R_i[condition]$ which partially excludes the range [1, 30] into attribute $dzone$ of $R_j[condition]$, which becomes $dzone$ in [31, 50]. This way, $exclusion(R_j, R_i) \leftarrow \{(s \in [1, 50]) \wedge (dzone \in [31, 50])\}$. For reasons of clarity, we do not show the first empty set corresponding to the first overlap. If shown, the result should become as follows: $exclusion(R_j, R_i) \leftarrow \{\emptyset, (szone \in [1, 50]) \wedge (dzone \in [31, 50])\}$. In the next example,

$$R_i[condition] = (szone \in [1, 60]) \wedge (dzone \in [20, 30])$$
$$R_j[condition] = (szone \in [1, 50]) \wedge (dzone \in [1, 50])$$

there are two simple overlaps of both attributes $szone$ and $dzone$ from $R_i[condition]$ to $R_j[condition]$, such that $exclusion(R_j, R_i)$ becomes $\{(szone \in [1, 50]) \wedge (dzone \in [1, 19]), (szone \in [1, 50]) \wedge (dzone \in [31, 50])\}$.

A more complete example is the following,

$$R_i[condition] = (szone \in [10, 40]) \wedge (dzone \in [20, 30])$$
$$R_j[condition] = (szone \in [1, 50]) \wedge (dzone \in [1, 50])$$

where $exclusion(R_j, R_i)$ becomes $\{(szone \in [1, 9]) \wedge (dzone \in [1, 50]), (szone \in [41, 50]) \wedge (dzone \in [1, 50]), (szone \in [10, 40]) \wedge (dzone \in [1, 19]), (szone \in [10, 40]) \wedge (dzone \in [31, 50])\}$.

Regarding a full exclusion, let us show the following example,

$$R_i[condition] = (szone \in [1, 60]) \wedge (dzone \in [1, 60])$$
$$R_j[condition] = (szone \in [1, 50]) \wedge (dzone \in [1, 50])$$

where the set of condition attributes of rule R_i completely excludes the ones of rule R_j. Then, applying $exclusion$ (R_j, R_i) returns an empty set (i.e., $\{\emptyset, \emptyset\} = \emptyset$). Hence, on a further execution of Algorithm 4 (and assuming that the decision field of both rules were different) the shadowing field of rule R_j (initialized as $false$ by default) would become $true$ (i.e., $R_j[shadowing] \leftarrow true$).

In order to show the execution of Algorithm 4 over a more complete set of rules, we sketch such an execution over the following set of rules:

$$R_1 : szone \in [10, 50] \rightarrow true$$
$$R_2 : szone \in [40, 90] \rightarrow false$$
$$R_3 : szone \in [60, 100] \rightarrow false$$
$$R_4 : szone \in [30, 80] \rightarrow true$$
$$R_5 : szone \in [1, 70] \rightarrow false$$

We start by showing the initial step within the first phase of Algorithm 4, where $i = 1$, and applied over the previous set of filtering rules. Let us notice that on this first step, the execution of function *exclusion*, with rules R_2 and R_1, since their decision is different, becomes the range [51, 90]. Similarly, the execution of function *exclusion*, with rules R_5 and R_1 becomes the range {[1, 9], [51, 70]}. The result of this first step is the following:

$R_1 : szone \in [10, 50] \rightarrow true$
$R_2 : szone \in [51, 90] \rightarrow false$
$R_3 : szone \in [60, 100] \rightarrow false$
$R_4 : szone \in [30, 80] \rightarrow true$
$R_5 : szone \in \{[1, 9], [51, 70]\} \rightarrow false$

Let us now move to the second step, with $i = 2$. In this step, the range of rule R_4 decreases since the execution of function *exclusion*, with rules R_2 and R_4, whose decision is different, becomes the range [30, 50]:

$R_1 : szone \in [10, 50] \rightarrow true$
$R_2 : szone \in [51, 90] \rightarrow false$
$R_3 : szone \in [60, 100] \rightarrow false$
$R_4 : szone \in [30, 50] \rightarrow true$
$R_5 : szone \in \{[1, 9], [51, 70]\} \rightarrow false$

At the end of the first phase, once executed both third and fourth steps, the resulting rules remain as above:

$R_1 : szone \in [10, 50] \rightarrow true$
$R_2 : szone \in [51, 90] \rightarrow false$
$R_3 : szone \in [60, 100] \rightarrow false$
$R_4 : szone \in [30, 50] \rightarrow true$
$R_5 : szone \in \{[1, 9], [51, 70]\} \rightarrow false$

Once the first phase is finished and running over the first step of the second phase, i.e., i equals 1, we notice that: (1) the result of applying function *testRedundancy* with rule R_1 as the second parameter becomes $false$; (2) the execution of function *exclusion*, with rules R_4 and R_1, completely excludes the condition attribute of rule R_4. Hence, rule R_4, is reported as shadowed by the combination of rules R_1 and R_2, and its condition attribute becomes an empty set. Therefore, the status field $shadowing$ of rule R_4, i.e., $R_4[shadowing]$, switches its value to $true$:

$R_1 : szone \in [10, 50] \rightarrow true$
$R_2 : szone \in [51, 90] \rightarrow false$
$R_3 : szone \in [60, 100] \rightarrow false$
$R_4 : \emptyset \rightarrow true$
$R_5 : szone \in \{[1, 9], [51, 70]\} \rightarrow false$

Then, we proceed to the second step of the second phase, i.e., i equals 2, and notice that rule R_2 disappears since the result of applying function *testRedundancy* with rule R_2 as the second parameter becomes $true$. Thus, the condition attribute of rule R_2 becomes an empty set, and its status field $redundancy$, i.e., $R_2[redundancy]$, switches its value to $true$:

$R_1 : szone \in [10, 50] \rightarrow true$
$R_2 : \emptyset \rightarrow false$
$R_3 : szone \in [60, 100] \rightarrow false$
$R_4 : \emptyset \rightarrow true$
$R_5 : szone \in \{[1, 9], [51, 70]\} \rightarrow false$

At the end of the following step, where i equals 3, the execution of function *testRedundancy* with rule R_3 as the second parameter becomes $false$. Thus, we apply function *exclusion*, with rules R_5 and R_3 as parameters. As a result of this execution, the second subrange of rule R_5 scarcely decreases from [51, 70] to [51, 59]:

$R_1 : szone \in [10, 50] \rightarrow true$
$R_2 : \emptyset \rightarrow false$
$R_3 : szone \in [60, 100] \rightarrow false$
$R_4 : \emptyset \rightarrow true$
$R_5 : szone \in \{[1, 9], [51, 59]\} \rightarrow false$

We do not show the rest of the execution, since the resulting set of filtering rules does not modify the previous one, which is the following:

$R_1 : szone \in [10, 50] \rightarrow true$
$R_3 : szone \in [60, 100] \rightarrow false$
$R_5 : szone \in \{[1, 9], [51, 59]\} \rightarrow false$

Let us recall that the following two warnings will notify the security officer of the discovery of both shadowing and redundancy anomalies, in order to verify the correctness of the whole detection and transformation process:

Shadowing on R_4 with R_2,R_1
Redundancy on R_2 with R_3,R_5

To conclude this section, let us finally show the warnings reported when executing Algorithm 4 over the configuration of the two components we showed in Fig. 2.

First case of irrelevance on $C_1\{R_1\}$
Second case of irrelevance on $C_1\{R_2\}$
Redundancy on $C_1\{R_4\}$ with $C_1\{R_3\}$,$C_1\{R_5\}$
Shadowing on $C_1\{R_6\}$ with $C_1\{R_3\}$,$C_1\{R_5\}$
Third case of irrelevance on $C_2\{R_2\}$

4.3 Correctness of the intra-component algorithms

Lemma 1 *Let* $R_i : \{condition_i\} \rightarrow decision_i$ *and* $R_j : \{condition_j\} \rightarrow decision_j$ *be two configuration rules. Then* $\{R_i, R_j\}$ *is equivalent to* $\{R_i, R'_j\}$ *where* $R'_j \leftarrow exclusion (R_j, R_i)$.

Proof Let us assume that

$R_i[condition] = A_1 \wedge A_2 \wedge \cdots \wedge A_p, and$
$R_j[condition] = B_1 \wedge B_2 \wedge \cdots \wedge B_p.$

If $(A_1 \cap B_1) = \emptyset$ or $(A_2 \cap B_2) = \emptyset$ or $\cdots$ or $(A_p \cap B_p) = \emptyset$ then $exclusion(R_j, R_i) \leftarrow R_j$. Hence, to prove the equivalence between $\{R_i, R_j\}$ and $\{R_i, R'_j\}$ is trivial in this case.

Let us now assume that

$$(A_1 \cap B_1) \neq \emptyset \quad and \quad (A_2 \cap B_2) \neq \emptyset \quad and \ldots \quad and \quad (A_p \cap B_p) \neq \emptyset .$$

If we apply rules $\{R_i, R_j\}$ where R_i comes before R_j, then rule R_j applies to a given packet if this packet satisfies $R_j[condition]$ but not $R_i[condition]$ (since R_i applies first). Therefore, notice that $R_j[condition] - R_i[condition]$ is equivalent to

$(B_1 - A_1) \wedge B_2 \wedge \cdots \wedge B_p$ *or*
$(A_1 \cap B_1) \wedge (B_2 - A_2) \wedge \cdots \wedge B_p$ *or*
$(A_1 \cap B_1) \wedge (A_2 \cap B_2) \wedge (B_3 - A_3) \wedge \cdots \wedge B_p$ *or*
$\ldots$
$(A_1 \cap B_1) \wedge \cdots \wedge (A_{p-1} \cap B_{p-1}) \wedge (B_p - A_p)$

which corresponds to $R'_j = exclusion(R_j, R_i)$. This way, if R_j applies to a given packet in $\{R_i, R_j\}$, then rule R'_j also applies to this packet in $\{R_i, R'_j\}$. Conversely, if R'_j applies to a given packet in $\{R_i, R'_j\}$, then this means this packet satisfies $R_j[condition]$ but not $R_i[condition]$. So, it is clear that rule R_j also applies to this packet in $\{R_i, R_j\}$. Since in Algorithm 1 $R'_j[decision]$ becomes $R_j[decision]$, this enables us to conclude that $\{R_i, R_j\}$ is equivalent to $\{R_i, R'_j\}$. □

Theorem 1 *Let R be a set of configuration rules and let* Tr(R) *be the resulting rules obtained by applying Algorithm* 4 *to R. Then R and* Tr(R) *are equivalent.*

Proof Let $\text{Tr}'_1(R)$ be the set of rules obtained after applying the first phase of Algorithm 4.

Since $\text{Tr}'_1(R)$ is derived from rule R by applying $exclusion(R_j, R_i)$ to some rules R_j in R, it is straightforward, from Lemma 1, to conclude that $\text{Tr}'_1(R)$ is equivalent to R.

Let us now move to the second phase, and let us consider a rule R_i such that $testRedundancy(R_i)$ (cf. Algorithm 3) is $true$. This means that $R_i[condition]$ can be derived by conditions of a set of rules S with the same decision and that come after in order than rule R_i.

Since every rule R_j with a decision different from the one of rules in S has already been excluded from rules of S in the first phase of the algorithm, we can conclude that rule R_i is definitely redundant and can be removed without changing the component configuration.

This way, we conclude that Algorithm 4 preserves equivalence in this case.

On the other hand, if $testRedundancy(R_i)$ is $false$, then the transformation consists in applying function $exclusion$ (R_j, R_i) to some rules R_j which also preserves equivalence. Similarly, and once in the third phase, let us consider a rule R_i such that $testIrrelevance(c, R_i)$ is $true$. This means that this rule matches traffic that will never traverse component c, or that it is irrelevant for the component's configuration. So, we can remove R_i from R without changing such a configuration.

Thus, in this third case, as in the other two cases, $Tr'(R)$ is equivalent to $Tr'_1(R)$ which, in turn, is equivalent to R. □

Lemma 2 *Let R_i : $\{condition_i\} \rightarrow decision_i$ and R_j : $\{condition_j\} \rightarrow decision_j$ be two configuration rules. Then rules R_i and R'_j, where $R'_j \leftarrow exclusion(R_j, R_i)$ will never simultaneously apply to any given packet.*

Proof Notice that rule R'_j only applies when rule R_i does not apply. Thus, if rule R'_j comes before rule R_i, this will not change the final decision since rule R'_j only applies to packets that do not match rule R_i. □

Theorem 2 *Let R be a set of configuration rules and let* Tr(R) *be the resulting rules obtained by applying Algorithm* 4 *to R. Then the following statements hold:* (1) *Ordering the rules in* Tr(R) *is no longer relevant;* (2) Tr(R) *is completely free of anomalies.*

Proof For any pair of rules R_i and R_j such that R_i comes before R_j, R_j is replaced by a rule R'_j obtained by recursively replacing R_j by $exclusion(R_j, R_k)$ for any $k < j$.

Then, by recursively applying Lemma 2, it is possible to commute rules R'_i and R'_j in $Tr(R)$ without changing the policy.

Regarding the second statement—Tr(R) is completely free of anomalies—notice that, in $Tr(R)$, each rule is independent of all other rules.

Thus, if we consider a rule R_i in $Tr(R)$ such that $R_i[condition] \neq \emptyset$, then this rule will apply to any packet that satisfies $R_i[condition]$, i.e., it is not shadowed.

On the other hand, rule R_i is not redundant because if we remove this rule, since this rule is the only one that applies to packets that satisfy $R_i[condition]$, then configuration of the component will change if we remove rule R_i from Tr(R).

Finally, and after the execution of Algorithm 4 over the initial set of configuration rules, one may verify that for each rule R_i in Tr(R) the following conditions hold:

(1) $szone = z_1 \cap source(r) \neq \emptyset$ and $dzone = z_2 \cap dest(r) \neq \emptyset$ such that $z_1 \neq z_2$ and component c is in $MR(z_1, z_2)$;
(2) if $A_c = attack\_category(R_i) \neq \emptyset$, the predicate $affects(A_c, z_2)$ becomes $true$.

Thus, each rule R_i in $Tr(R)$ is not irrelevant. □

4.4 Complexity of the intra-component algorithms

Let us discuss in this section the degree of computational complexity of our approach's main algorithm, i.e., Algorithm 1, with respect to the increase of the initial number of rules due to the whole rewriting process of Algorithm 4. Indeed, in a worst case scenario (e.g., Fig. 3b), Algorithm 1 may generate a huge number of rules due to the exclusion routine defined by Algorithm 1. For instance, if we have two rules

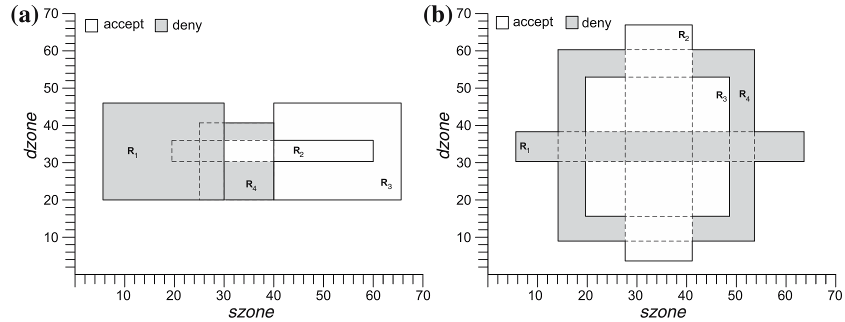


Fig. 3 Normal and worst ruleset examples. **a** Normal case example **b** Worst case example

with p attributes, the second rule can be replaced by p new rules in the worst case, leading to $p + 1$ rules.

If we now assume that we have n rules ($n > 2$) with p attributes, then each rule except the first one can be replaced by p new rules in the first rewriting step of the algorithm. In the second step, the p rules that replace the second rule are combined with the p rules that replace rules 3 to n. Thus, each rule from 3 to n can be replaced by p^2 new rules. In the third step, the p^2 rules corresponding to rule 3 are combined with the p^2 rules corresponding to rules 4 to n. We can show that this may lead to p^3 new rules. And so on. Hence, in the worst case, if we have n rules ($n > 2$) with p attributes, then we can obtain $1 + p + p^2 + \cdots + p^{n-1}$ rules when applying Algorithm 1 from Algorithm 4, that is $\frac{p^n-1}{p-1}$ rules.

Although this complexity seems very high, in all the experiments we have done (cf. Sect. 6), we were always very far from this case. First, because only attributes *szone* and *dzone* may significantly overlap and exert a bad influence on our algorithm's complexity. Other attributes, such as *protocol*, *sport*, and/or *dport*, are generally equal or completely different when combin- ing configuration rules. Second, administrators generally use overlapping rules in their configurations to represent rules that may have *exceptions* [4]. This situation is closer to the normal case presented in Fig. 3a than to the worst case scenario shown in Fig. 3b. Third, when anomalies are detected by our algorithms, some rules are removed—which significantly reduces the theoretical complexity.

4.5 Default policies

We assume in our work that each component implements a positive (i.e., close) or negative (i.e., open) policy. If it is positive, the default decision is to *alert* or to *deny* a packet when any configuration rule applies. By contrast, the negative policy will *accept* or *pass* a packet when no rule applies.

After rewriting the rules with our intra-component-audit algorithms, we can actually remove every rule whose decision is *pass* or *accept* if the policy of this component is negative (since this rule is redundant with the default policy); and similarly we can remove every rule whose decision is *deny* or *alert* if its policy is positive. Thus, we can consider that our proposed *intra-component-audit* algorithm generates a configuration that only contains positive rules if the component default policy is negative, and negative rules if the default policy is positive.

5 Inter-component classification and algorithms

The objective of the inter-component audit algorithms is the complete detection of policy anomalies that could exist in a multi-component policy, i.e., to discover and warn the security officer about potential anomalies between policies of different components.

The main hypotheses for applying our inter-component algorithms assume the following:

1. An upstream traffic flows away from the closest component to the origin of this traffic (i.e., the most-upstream component [6]) towards the closest component to the remote destination (i.e., the most-downstream component [6]);
2. Every component's policy in the network has been rewritten using the intra-component algorithms defined in Sect. 4, i.e., it does not contain intra-component anomalies and the rules within such a policy are completely independent between them.

5.1 Inter-component anomalies classification

In this section, we classify the complete set of anomalies that can occur within a multi-component policy. Our classification is based on the network model presented in Sect. 3. An example for each anomaly is illustrated through the distributed multi-component policy setup shown in Fig. 4.

Inter-component shadowing A shadowing anomaly occurs between two components when the following conditions hold: (1) The most-upstream component is a firewall; (2) The downstream component, where the anomaly is detected, does not block or report (completely or partially) traffic that is blocked (explicitly, by means of positive rules; or implic-

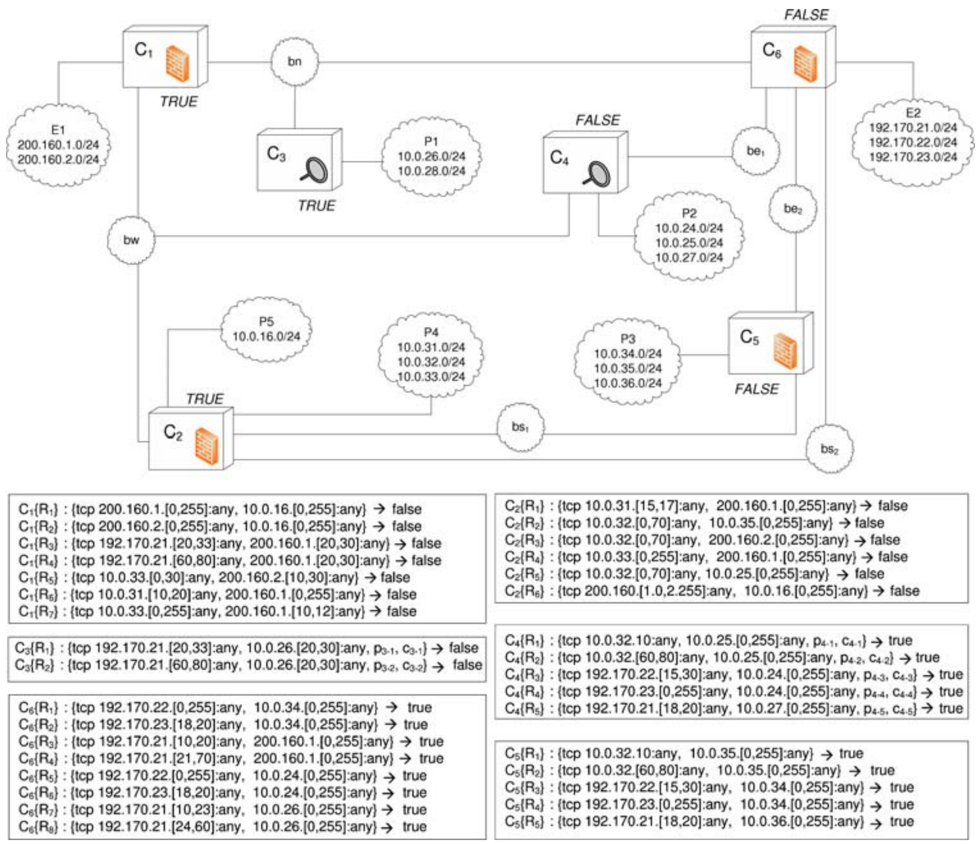


Fig. 4 Example of a distributed network security policy setup

itly, by means of its default policy), by the most-upstream component.

The explicit shadowing as result of the union of rules $C_6\{R_7\}$ and $C_6\{R_8\}$ to the traffic that the component C_3 matches by means of rule $C_3\{R_1\}$ is a proper example of *full shadowing* between a firewall and a NIDS. Similarly, the anomaly between $C_3\{R_2\}$ and $C_6\{R_8\}$ shows an example of an *explicit partial shadowing* anomaly between a firewall and a NIDS.

On the other hand, the implicit shadowing between the rule $C_1\{R_5\}$ and the default policy of component C_2 is a proper example of *implicit full shadowing* between two firewalls. Finally, the anomaly between the rule $C_1\{R_6\}$, $C_2\{R_1\}$, and the default policy of component C_2 shows an example of an *implicit partial shadowing* anomaly between two firewalls.

Inter-component redundancy A redundancy anomaly occurs between two components when the following conditions hold: (1) The most-upstream component is a firewall; (2) The downstream component, where the anomaly is detected, blocks or reports (completely or partially) traffic that is blocked by the most-upstream component.

A proper example of *full redundancy* between two firewalls is shown by rules $C_5\{R_3\}$ and $C_6\{R_1\}$; rules $C_4\{R_3\}$ and $C_6\{R_5\}$, on the other hand, show an example of *full redundancy* between a firewall and a NIDS. Similarly, rules $C_5\{R_4\}$ and $C_6\{R_2\}$ show a proper example of *partial redundancy* between two firewalls, whereas rules $C_4\{R_4\}$ and $C_6\{R_6\}$ show an example of *partial redundancy* between a firewall and a NIDS.

Although this kind of redundancy is expressly introduced by network administrators sometimes (e.g., to guarantee the forbidden traffic will not reach the destination), it is important to discover it since, if such a rule is applied, we may conclude that at least one of the redundant components is working wrongly. For that reason, our proposal does not advise the administrator to remove the redundant rule from the set of rules; but it advises the administrator to give a different meaning to that rule—by adding, for instance, an extra attribute to the rule (e.g., a log attribute pointing out to such a situation).

Inter-component misconnection A misconnection anomaly occurs between two components when the most-upstream component is a firewall that permits (explicitly, by means of negative rules; or implicitly, through its default policy) all the traffic—or just a part of it—that is then denied by a downstream firewall. For example, we have a full explicit misconnection between firewalls C_5 and C_2 due to rules $C_5\{R_1\}$ and $C_2\{R_2\}$ (*full misconnection*); and a partial explicit misconnection due to rules $C_5\{R_2\}$ and $C_2\{R_2\}$. Similarly, we can observe a full implicit misconnection anomaly between firewalls C_1 and C_2 due to rule $C_2\{R_3\}$ and the default policy of firewall C_1; and a partial implicit misconnection anomaly

Algorithm 5: inter-component-audit(C)

```
foreach c ∈ C do
    foreach r ∈ c[rules] do
        Z_s ← {z ∈ Z | z ∩ source (r) ≠ ∅};
        Z_d ← {z ∈ Z | z ∩ dest (r) ≠ ∅};
        foreach z_1 ∈ Z_s do
            foreach z_2 ∈ Z_d do
                audit (c,r,z_1,z_2);
```

Algorithm 6: audit(c,r,z_1,z_2)

```
foreach p ∈ MR (z_1,z_2) do
    path_d ← tail (c,p);
    path_u ← head (c,p);
    if path_d ≠ ∅ and r[decision] = "false" and
     isFirewall(c) then
        c_d ← firstFirewall(path_d);
        downstream(r,c,c_d);
    if path_u ≠ ∅ then
        c_u ← last(path_u);
        if isFirewall(c_u) then upstream(r,c,c_u);
```

Algorithm 7: downstream(r,c,c_d)

```
if c_d[policy] = "true" then
    R_df ← {r_d ∈ c_d | r_d ∽ r ∧ r_d[decision] = "false"};
    if R_df = ∅ then warning ("Full Misconnection ");
    else if ¬testRedundancy(R_df, r) then
        warning ("Partial Misconnection");
```

Algorithm 8: upstream(r,c,c_u)

```
R_uf ← {r_u ∈ c_u | r_u ∽ r ∧ r_u[decision] = "false"};
R_ut ← {r_u ∈ c_u | r_u ∽ r ∧ r_u[decision] = "true"};
if r[decision] = "true" then
    if testRedundancy (R_ut,r) then
        warning ("Full Redundancy");
    else if R_ut ≠ ∅ then
        warning ("Partial Redundancy");
    else if isFirewall(c) then
        if testRedundancy (R_uf,r) then
            warning ("Full Misconnection");
        else if R_uf ≠ ∅ then
            warning ("Partial Misconnection");
        else if R_uf = ∅ and R_ut = ∅ and
         c_u[policy] = "false" then
            warning ("Full Misconnection");
else if r[decision] = "false" then
    if testRedundancy (R_ut,r) then
        warning ("Full Shadowing");
    else if R_ut ≠ ∅ then
        warning ("Partial Shadowing");
    else if R_uf = ∅ and c_u[policy] = "true" then
        warning ("Full Shadowing");
    else if ¬ testRedundancy (R_uf,r)
    and c_u[policy] = "true" then
        warning ("Partial Shadowing");
```

due to rules $C_1\{R_6\}$ and $C_2\{R_1\}$, together with the default policy of C_2.

5.2 Inter-component analysis algorithms

For reasons of clarity, we split the whole analysis process into four different algorithms. The input for the first algorithm (cf. Algorithm 5) is the set of components C, such that for all $c \in C$, we note $c[rules]$ as the set of configuration rules of component c, and $c[policy] \in \{true, false\}$ as the default policy of such a component c.

In turn, each rule $r \in c[rules]$ consists of a conjunctive set of condition attributes (i.e., $szone$, $dzone$, $sport$, $dport$, $protocol$, etc.) pointing out to a $decision$ over the values $true$ or $false$.

Let us recall here the functions $source(r) = szone$ and $dest(r) = dzone$. Thus, we compute for each component $c \in C$ and for each rule $r \in c[rules]$, each one of the source zones $z_1 \in Z_s$ and destination zones $z_2 \in Z_d$—whose intersection with respectively $szone$ and $dzone$ is not empty—which become, together with a reference to each component c and each rule r, the input for the second algorithm (i.e., Algorithm 6).

Once in Algorithm 6, we compute the minimal route of components that connects zone z_1 to z_2, i.e., $[C_1, C_2, \ldots, C_n] \in MR(z_1, z_2)$. Then, we decompose the set of components inside each path in downstream path ($path_d$) and upstream path ($path_u$). To do so, we use functions $head$ and $tail$ (defined below). The first component $c_d \in path_d$, and the last component $c_u \in path_u$ are passed, respectively, as argument to the last two algorithms (i.e., Algorithms 7 and 8) in order to conclude the set of necessary checks that guarantee the audit process.

Some other operators and routines called from these algorithms are the following: (1) operator "$\backsim$", which denotes that two rules r_i and r_j are correlated if every attribute in R_i has a non-empty intersection with the corresponding attribute in R_j; (2) routine $tail(c_i, path)$, which returns the downstream path cointaining those components $c_j \in path$ placed just after component c_i; (3) routine $head(c_i, path)$, which returns the upstream path of components $c_j \in path$ which are placed just before component c_i; and (4) routine $firstFirewall(path)$, which returns the first component $c_i \in path$ such that predicate $isFirewall(c_i)$ becomes $true$.

Let us conclude this section by giving in Fig. 5 an outlook to the set of warnings sent to the security officer after the execution of Algorithm 5 in the scenario of Fig. 4.

5.3 Correctness of the inter-component algorithms

To prove the correctness of our inter-component algorithms, we first define what is a deployment of configuration rules

$C_1\{R_3\} - C_6\{R_3, R_4\}$: Full Shadowing
$C_1\{R_4\} - C_6\{R_4\}$: Partial Shadowing
$C_1\{R_5\} - C_2\{pol.\}$: Full Shadowing
$C_1\{R_6\} - C_2\{R_1, pol.\}$: Partial Shadowing
$C_2\{R_3\} - C_1\{pol.\}$: Full Misconnection
$C_2\{R_4\} - C_1\{R_7, pol.\}$: Partial Misconnection
$C_3\{R_1\} - C_6\{R_7, R_8\}$: Full Shadowing
$C_3\{R_2\} - C_6\{R_8\}$: Partial Shadowing
$C_4\{R_3\} - C_6\{R_5\}$: Full Redundancy
$C_4\{R_4\} - C_6\{R_6\}$: Partial Redundancy
$C_5\{R_1\} - C_2\{R_2\}$: Full Misconnection
$C_5\{R_2\} - C_2\{R_2\}$: Partial Misconnection
$C_5\{R_3\} - C_6\{R_1\}$: Full Redundancy
$C_5\{R_4\} - C_6\{R_2\}$: Partial Redundancy
$C_5\{R_5\} - C_6\{pol.\}$: Full Misconnection

Fig. 5 Execution of Algorithm 5 over the scenario of Fig. 4

Algorithm 9: policy-rewriting(R)

```
for i ← 1 to (count(R) − 1) do
    for j ← (i + 1) to count(R) do
        if R_j[type] = R_i[type] then
            R_j ← exclusion (R_j, R_i);
```

without anomalies. For this purpose, let us consider a set R of configuration rules to be deployed over a set C of components that partitions a network into a set Z of zones. We assume that the set of rules R has been rewritten by Algorithm 9 into $\mathrm{Tr}(R)$, which, in turn, is equivalent to R, but completely free of any possible relation between rules of the same type (e.g., filtering or alerting rules).

Algorithm 9 is a simplified version of Algorithm 4. It automatically fixes any dependency between rules of the same type (e.g., filtering or alerting rules). Like Algorithm 4, the rewriting process defined in Algorithm 9 relies on a iterative execution of the auxiliary function *exclusion* defined in Algorithm 1 (cf. Sect. 4). Therefore, similar reasonings as used to prove the correctness of Algorithm 4 allow us to prove the correctness of Algorithm 9.

Let us now consider a rule $r \in \mathrm{Tr}(R)$ and let us assume that r applies to a source zone z_1 and a destination zone z_2, i.e., $szone = z_1 \cap source(r) \neq \emptyset$ and $dzone = z_2 \cap dest(r) \neq \emptyset$. Let r' be a rule identical to r except that $source(r') = szone$ and $dest(r') = dzone$. Let us also assume that $[C_1, C_2, \ldots, C_k] \in MR(z_1, z_2)$. We then define our deployment principle as follows.

Definition 1 Any rule $r \in Tr(R)$ will be deployed over the set C of components. There are two different cases: $r[decision] =$ "$false$" or $r[decision] =$ "$true$".

If $r[decision] =$ "$false$" then, on every component on the minimal route from source $szone$ to destination $dzone$, deploy a negative rule r' (i.e., an *accept* filtering rule r' if the component is a firewall, or a *pass* alerting rule r' if the component is a NIDS).

Conversely, if $r[decision] =$ "$true$", then the two following possibilities hold:

(1) if r is a filtering rule, then deploy a $deny$ filtering rule r' on the most-upstream firewall of the minimal route (if such a firewall does not exist, then generate a deployment error message);
(2) if r is an alerting rule, then deploy an $alert$ rule r' on the first NIDS located before the most-upstream firewall of the minimal route (if such a NIDS does not exist, then generate a deployment error message).

Having defined our deployment principle, let us now consider the aggregation process shown in Algorithm 10, which is intended for the aggregation of configurations rules from a set of components C into a global set of rules R. (An earlier version of this algorithm is presented in [3].) The input data of our aggregation process are the set C of components whose configurations we want to fold up. As we can notice in line 3 of Algorithm 10, the configuration of each component $c_i \in C$ if first fixed by applying the intra-component-audit algorithm presented in Sect. 4.

The gathering of configuration rules is according to the deployment principle stated in Definition 1. In this way, for each negative rule configured in a component, we expect to find an open flow of permissions within every component in the minimal route from the source zone to the destination zone of such a rule. Otherwise, an aggregation error message is generated. On the other hand, for each positive rule, if it is a filtering rule, we expect to find such a prohibition on the first firewall of the minimal route from the source zone to the destination zone; otherwise, an aggregation error message is generated; if such a rule is an alerting rule, we expect to not find an upstream firewall on the minimal route from the source zone to the destination zone blocking its traffic; otherwise, an aggregation error message is generated.

Based on the deployment and aggregation processes defined above, we can now prove the following theorem:

Theorem 3 *Let $C[rules]$ be the set of component configurations obtained by applying Definition 1 over the set R of configuration rules obtained, in turn, by applying Algorithm 10 over C. Then, the audit process of Algorithm 5 does not detect any inter-component anomaly in the configurations of $C[rules]$.*

Proof Let C be a set of components that partitions the network into a set Z of zones, and whose component configurations are aggregated into R by applying Algorithm 10.

Let us first prove that if there exists, at least, one rule $r_i \in C[rules]$ such that it presents an inter-component anomaly (as defined in Sect. 5.1), then the aggregation of rules $R \leftarrow aggregation(C)$ through the use of Algorithm 10 does

Algorithm 10: aggregation(C)

```
1  R ← ∅; i ← ∅;
2  foreach c0 ∈ C do
3      intra-component-audit (c0,c0[rules]);
4  foreach c1 ∈ C do
5      foreach r1 ∈ c1[rules] do
6          Zs ← {z ∈ Z | z ∩ source (r1) ≠ ∅};
7          Zd ← {z ∈ Z | z ∩ dest (r1) ≠ ∅};
8          foreach z1 ∈ Zs do
9              foreach z2 ∈ Zd do
10                 if (r1[decision] = "false") then
11                     C2 ← {c2 ∈ head(c1, MR(z1, z2)) | isFirewall(c2) = true};
12                     foreach c2 ∈ C2 do
13                         c2rf ← {r2 ∈ c2[rules] | r1 ∽ r2 ∧ r2[decision] = "false"};
14                         if (empty(c2rf)) then
15                             c2rt ← {r2 ∈ c2[rules] | r1 ∽ r2 ∧ r2[decision] = "true"};
16                             if (¬empty(c2rt)) or (c2[policy] = "true") then
17                                 aggregationError ();
18                                 return ∅;
19                 else if (r1[decision] = "true") then
20                     if (isFirewall(c1)) and (first(MR(z1, z2)) ≠ c1) then
21                         aggregationError ();
22                         return ∅;
23                     else if (isNIDS(c1)) then
24                         C2 ← {c2 ∈ head(c1, MR(z1, z2)) | isFirewall(c2) = true};
25                         foreach c2 ∈ C2 do
26                             c2rf ← {r2 ∈ c2[rules] | r1 ∽ r2 ∧ r2[decision] = "false"};
27                             if (empty(c2rf)) then
28                                 c2rt ← {r2 ∈ c2[rules] | r1 ∽ r2 ∧ r2[decision] = "true"};
29                                 if (¬empty(c2rt)) or (c2[policy] = "true") then
30                                     aggregationError ();
31                                     return ∅;
32                 Ri ← Ri ∪ r1;
33                 Ri[szone] ← z1;
34                 Ri[dzone] ← z2;
35                 i ← (i + 1);
36 policy-rewriting (R);
37 return R;
```

not generate a consistent set of rules R that can be further deployed over the network by using the deployment principle stated in Definition 1.

For instance, let us assume that $r_i \in C[rules]$ presents an inter-component shadowing. If so, r_i is a negative rule (i.e., either *accept* or *pass*) that applies to a source zone z_1 and a destination zone z_2 such that $szone = z_1 \cap source(r_i) \neq \emptyset$, $dzone = z_2 \cap dest(r_i) \neq \emptyset$; r_i belongs to a component $C_i \in C$ which is in the path $[C_1, C_2, \ldots, C_k] \in MR(z_1, z_2)$; and it exists at least one component C_j such that the following conditions hold: (1) component C_j is an upstream firewall, i.e., $C_j \in head(C_i, MR(z_1, z_2)) \wedge isFirewall(C_j)= true$; (2) component C_j explicitly or implicitly blocks the traffic that r_i matches, i.e., either there exists a rule $r_j \in C_j[rules]$ such that $r_j \backsim r_i \wedge r_j[decision] =$ "$true$"; or $C_j[policy] =$ "$true$" and there is not $r_j \in C_j[rules]$ such that $r_j \backsim r_i \wedge r_j[decision] =$ "$false$".

If this situation applies, we can observe that during the aggregation process specified by Algorithm 10, rule r_i matches statement 10 , i.e., $r_i[decision]$= "$false''$ becomes $true$. Then, the process analyzes through statements 12–18 whether there exists at least an upstream firewall C_j such that it blocks the traffic that r_i also matches, i.e., it does not contain negative filtering rules accepting that traffic (statement 14 becomes $true$) and either it explicitly blocks that traffic through a positive filtering rule (first condition of statement 16 becomes $true$), or it implicitly blocks that traffic through its default policy (second condition of statement 16 becomes $true$). If so, the process finishes with an error and returns an empty set of rules (cf. statements 17 and 18).

Let us now assume that $r_i \in C[rules]$ presents an inter-component redundancy. If so, r_i is a positive rule (i.e., either $deny$ or $alert$) that applies to a source zone z_1 and a destination zone z_2 (such that $szone = z_1 \cap source(r_i) \neq \emptyset$, $dzone = z_2 \cap dest(r_i) \neq \emptyset$); r_i belongs to a component $C_i \in C$ which is in the path $[C_1, C_2, \ldots, C_k] \in MR(z_1, z_2)$; and one of the following conditions hold: (1) component C_i is a firewall and there exists, at least, an upstream firewall C_j that either explicitly or implicitly blocks the traffic that r_i already blocks (without justifying r_i such a redundancy by means of an additional attribute like, for example, an attribute for $logs$); (2) component C_i is a NIDS located after an upstream firewall on the minimal route which blocks the traffic of r_i (without justifying r_i such a redundancy by means of an additional attribute like, for example, an attribute for $logs$). If condition (1) of this situation applies, we can observe that during the folding process specified by Algorithm 10, rule r_i matches statement 19, i.e., $r_i[decision] =$ "$true$" becomes $true$, and the two conditions of statement 20, i.e., r_i is placed within a firewall and such a firewall is not the most-upstream component of the minimal route from z_1 to z_2. Thus, the process finishes with an error and returns an empty set of rules (cf. statements 21 and 22).

Similarly, if condition (2) of this situation applies, we can observe that during the folding process specified by Algorithm 10 rule r_i matches both statement 19, i.e., $r_i[decision]$ ="$true$", and statement 23, i.e., predicate $isNIDS(C_i) = true$. Then, the process analyzes through statements 25 –31 whether there exists at least an upstream firewall C_j that blocks the traffic that r_i also matches, i.e., it does not contain negative filtering rules accepting that traffic (statement 27 becomes $true$) and either it explicitly blocks that traffic through a positive filtering rule (first condition of statement 29 becomes $true$), or it implicitly blocks that traffic through its default policy (second condition of statement 29 becomes $true$). If so, the process finishes with an error and returns an empty set of rules (cf. statements 30 and 31).

Let us finally assume that $r_i \in C[rules]$ presents an inter-component misconnection. If so, r_i is a positive filtering rule (i.e., $deny$) that applies to a source zone z_1 and a destination zone z_2 such that $szone = z_1 \cap source(r_i) \neq \emptyset$, $dzone = z_2 \cap dest(r_i) \neq \emptyset$; r_i belongs to a firewall $C_i \in C$ which is in the path $[C_1, C_2, \ldots, C_k] \in MR(z_1, z_2)$; and there exists, at least, an upstream firewall C_j that either explicitly or implicitly accepts the traffic that r_i blocks. In order to avoid this situation it suffices to detect whether firewall C_i is not the most-upstream firewall. As we have shown in the previous case, this situation is handled by the aggregation process specified by Algorithm 10 through statement 19 and the two conditions of statement 20. So, if r_i is placed within a firewall and such a firewall is not the most-upstream component of the minimal route from z_1 to z_2, the aggregation process finishes with an error and returns an empty set of rules (cf. statements 30 and 31).

It is straightforward, then, to conclude that when no inter-component anomalies apply to the set of component configurations $C[rules]$, the aggregating process specified by Algorithm 10 returns a global set of filtering rules R with the union of all the configuration rules (cf. statements 32–35 of Algorithm 10) previously deployed over the set of components C.

Let us notice that we apply in line 36 of Algorithm 10 the rewriting process defined in Algorithm 9. In this way, we can guarantee that there are no dependencies between rules of the same type (i.e., alerting and filtering rules) in the set of rules aggregated during the folding process of Algorithm 10. As stated above, and similarly to Algorithm 4 (cf. Sect. 4), Algorithm 9 relies on a iterative execution of the auxiliary function *exclusion* defined in Algorithm 1 (cf. Sect. 4). Therefore, similar reasonings as used to prove the correctness of Algorithm 4 (cf. Sect. 4.3) enables us to prove that the set of rules returned by Algorithm 10 is free of intra-component anomalies.

If we now deploy the set of rules R obtained from Algorithm 10 by using the deployment principle stated in Definition 1, and since we agree that R belongs to a set of configuration rules $C[rules]$ that is free of inter-component anomalies, we can then guarantee that the deployed set of configurations is also free of inter-component anomalies, i.e., the audit process of Algorithm 5 does not detect any inter-component anomaly in the configurations already deployed. □

6 Implementation and performance evaluation

We implemented the complete set of algorithms and processes presented in this paper in a software prototype called MIRAGE (which stands for MIsconfiguRAtion manaGEr). MIRAGE has been developed using PHP, a general purpose scripting language that is especially suited for web services development and can be embedded into HTML for the construction of client-side GUI based applications [9]. MIRAGE can be locally or remotely executed by using a HTTP server (e.g., Apache server over UNIX or Windows setups) and a web browser. The user interface of MIRAGE not only allows the whole management of those processes described in this paper, but also the management of the network properties described in Sect. 3. In order to do so, MIRAGE extracts such information from SKYBOX [26], an automatic network tool that allows us to properly manage the set of components, the set of configurations rules of each component, the set of zones of the system, and so on. In fact, both the network properties and the whole policies are derived from—and represented into—SKYBOX-based XML files.

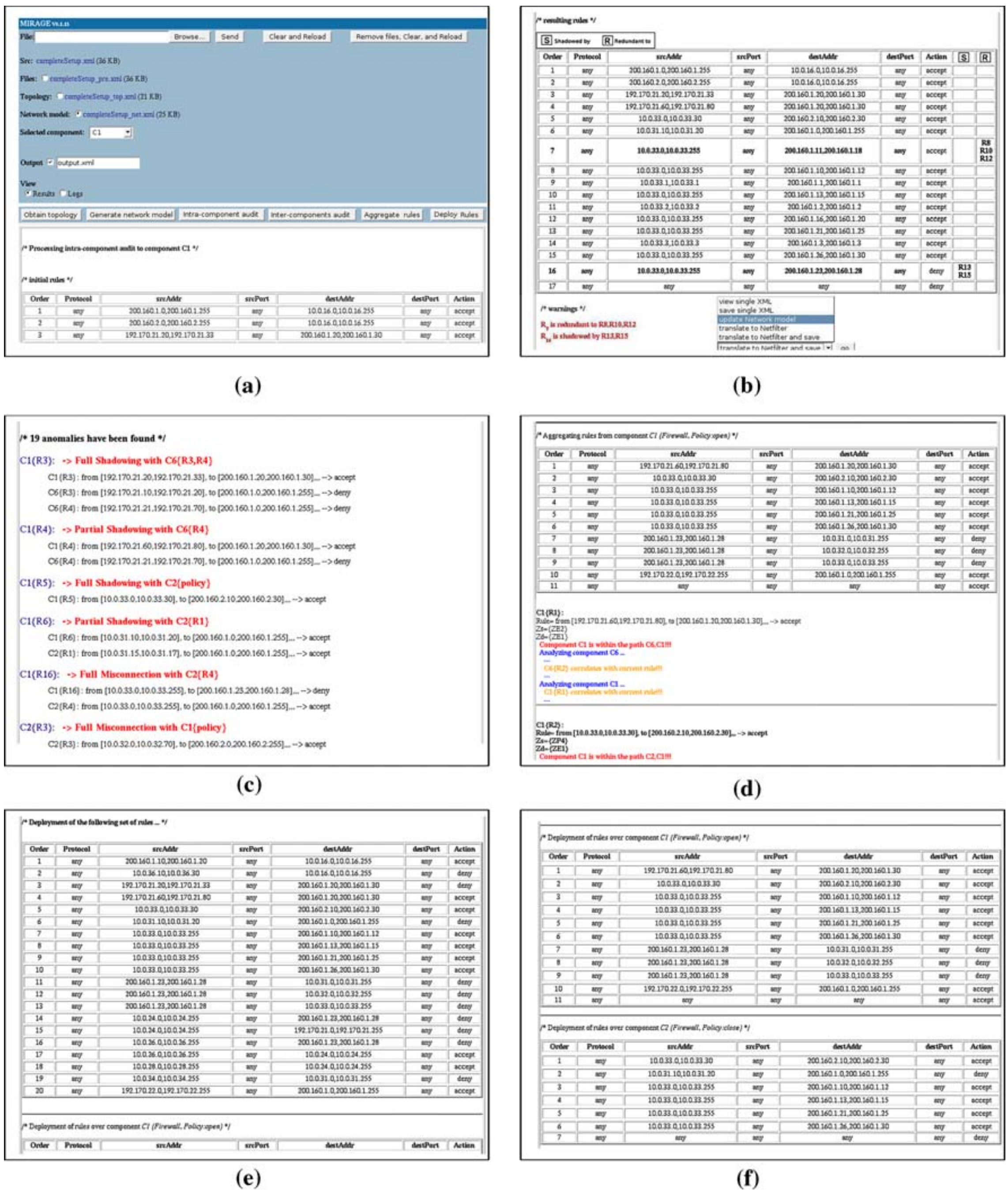


Fig. 6 Samples of the graphical environment of MIRAGE. **a** Main interface of MIRAGE. **b** Intra-component analysis results. **c** Inter-component analysis results. **d** Rule aggregation results. **e** Rule deployment results (1/2). **f** Rule deployment results (2/2)

We show in Fig. 6 some screenshots of the graphical environment of MIRAGE. We first see in Fig. 6a the main interface of our tool. The top-left panel allows the load of SKYBOX-based XML files, from which one can supply the topology of the system and the set of security rules already deployed over the network from a single XML file based on SKYBOX. Through a set of transformations, MIRAGE derives the specific instances of the network model described in Sect. 3, and remains ready to perform the complete set of processes defined in this paper.

Figure 6 also shows some other options placed in its middle panel, such as the selection of components, and buttons to call its main functions/routines, which are the following: (1) intra-component analysis of rules; (2) inter-component

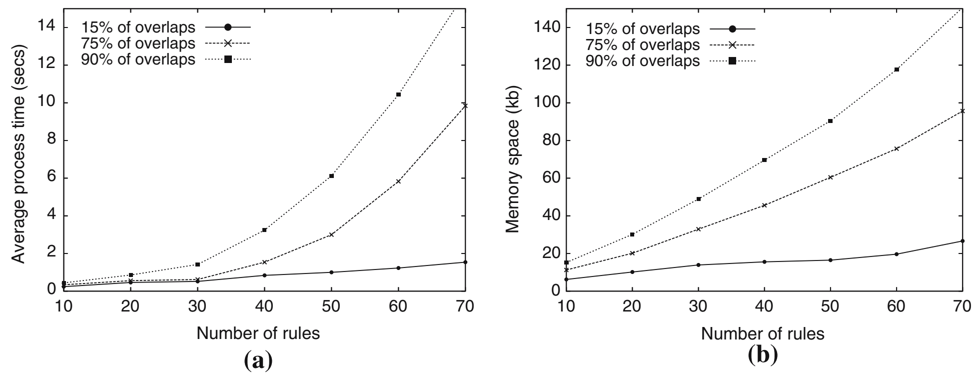


Fig. 7 Intra-component analysis evaluations. **a** CPU evaluation; **b** Memory evaluation

analysis of rules; (3) aggregation of existing rules into a single global policy; (4) deployment of the resulting global set of rules into a different set of components. These four procedures can work in two different modes: (a) *results* mode, which is a quick mode that only reports warnings and proper results of a given process; (b) *logs* mode, which is a more detailed mode that not only reports warnings and results, but also those information generated and exchanged between functions during the whole execution of a given process.

We can see in Fig. 6b the output view of MIRAGE when performing the intra-component audit process to the set of rules of a given component. In this case, the set of filtering rules of a firewall are analyzed, and two intra-component anomalies are detected. Furthermore, the prototype leaves the option to the administrator to fix those anomalies by updating the network model. Similarly, we can see in Fig. 6c the result of applying our inter-component audit process to the complete set of components' rules. Finally, we show in Fig. 6d–f the output view after applying, respectively, the aggregation and deploying processes defined in Sect. 5.3.

In the following section, we show some experimental results carried out by using the graphical user interface of MIRAGE.

6.1 Performance evaluation

We evaluated the implementation of MIRAGE through a set of experiments over different IPv4-based security components and networks, and through the use of the *results* mode of its four main routines. The experiments were carried out on an Intel-Pentium M 1.4 GHz processor with 512 MB RAM, running Debian GNU/Linux 2.6, and using Apache/1.3 with PHP/4.3 configured. We did not measure in our evaluations the performance for parsing and constructing the topological descriptions derived from the XML files loaded into MIRAGE. This process was performed just once at the beginning of each evaluation, and we do not consider it as relevant.

We first evaluated the performance of our intra-component audit algorithms by analyzing the average time and memory space utilized when processing different set of security rules for three different components. We created the configuration of each component based on the security policy characteristics of our real institutional network. More specifically, the set of components utilized for this first evaluation consisted of two firewalls based on netfilter [27] and ipfilter [23], and a NIDS based on snort [24].

Each component was configured towards three different zones with more than 50 hosts in each zone. The configuration rules of those components consisted of the following main attribute fields: source IP address, destination IP address, source port number, destination port number, and protocol type. The configuration rules of the NIDS included, moreover, two additional values to take into account, the payload and the attack classification associated to each rule.

Figure 7a shows the average execution times (in seconds) for performing the intra-component analysis of those three components versus the total number of rules of their configurations. Three different curves are shown, one for each of the following cases: (1) netfilter firewall rules, of which 15% presented overlaps between their attributes; (2) ipfilter firewall rules, of which 75% presented overlaps between their attributes; and (3) snort-based alerting rules, of which 90% presented overlaps between their attributes. The horizontal axis indicates the total number of rules and the vertical axis indicates the average process time. Similarly, Fig. 7b indicates the associated space memory consumption during the same executions, where its horizontal axis indicates the total number of rules and its vertical axis the memory space consumption (in kilobytes).

As expected, according to the complexity analyzed in Sect. 4.4, the first case scenario showed the least processing time and memory space consumption (it took less than 2 s and almost 27 kilobytes of memory the analysis of 70 rules with 15% of overlaps); and the third case scenario presented the highest processing time and memory space consumption

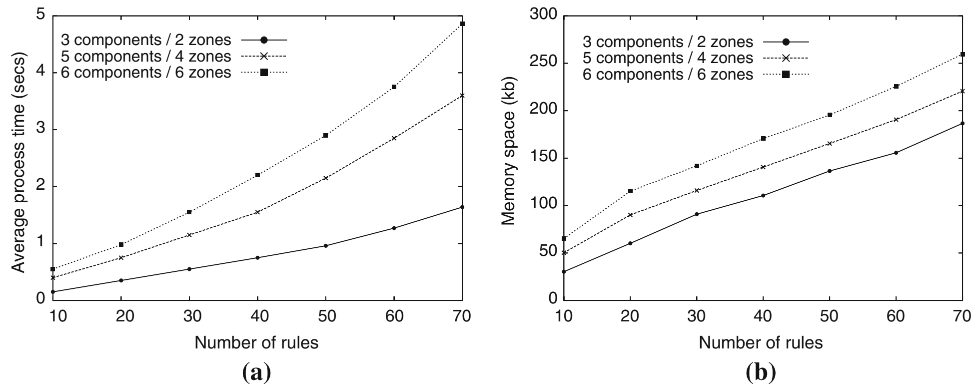


Fig. 8 Inter-component analysis evaluations. **a** CPU evaluation; **b** Memory evaluation

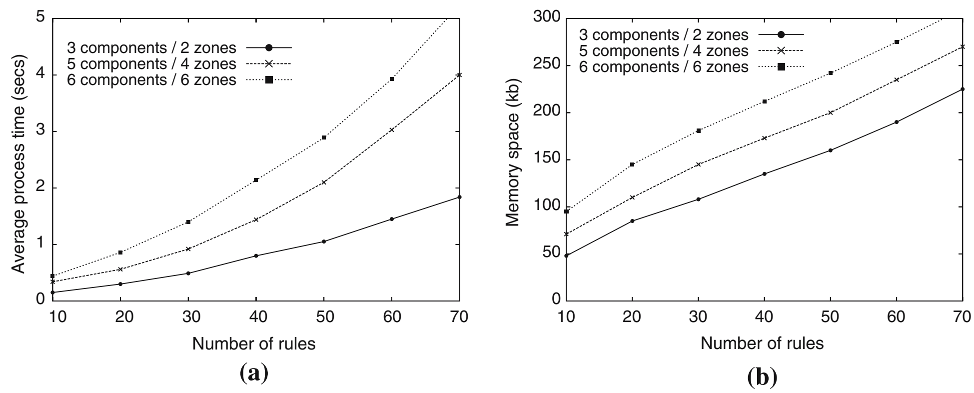


Fig. 9 Aggregation process evaluations. **a** CPU evaluation. **b** Memory evaluation

(it took more than 15 s and almost 150 kilobytes of memory for the analysis of 70 rules with 90% of overlaps).

We can notice, however, that even if the theoretical complexity of the third case should bound close to O(p^n), where p is the number of attributes, and n the number of rules, we were far from this complexity, and our implementation scaled well with the increase of rules. We further verified that although the complexity of Algorithm 4 is determined by the complexity of splitting rules, the dynamic removal of anomalies, and the distribution of overlaps between rule attributes, significantly reduces the execution complexity.

We measured, in a second phase of our evaluations, the average time and memory space consumption when processing our inter-component audit algorithms through a progressive increment of security rules, components and networks. The configuration of every component was previously analyzed with our intra-component audit process, and any possible anomaly and/or overlap between rule attributes was previously removed.

The results of these measurements are plotted in Fig. 8a and b as three different curves, according to the three following topologies: (1) two subnetworks with two firewalls and one NIDS; (2) four subnetworks with three firewalls and two NIDSs; and (3) six subnetworks with four firewalls and two NIDSs. These same three topologies were also utilized for measuring the average time and memory space when performing the aggregation process defined in Sect. 5.3. The results of these last measurements are plotted in Figs. 9a and b, respectively.

From Figs. 8a and b we see that it took less than 2 s and 200 kilobytes of memory for the analysis of 70 security rules distributed between three components and two subnetworks; and almost 5 s and 260 kilobytes of memory for the analysis of the same number of rules distributed between six components and six subnetworks. The analysis of those same scenarios, but through the aggregation process specified in Sect. 5.3 increased both processing time and memory space consumption. More specifically, it took almost 7 s and 310 kilobytes of memory the aggregation of the 70 security rules distributed between six components and six subnetworks. We consider this increase reasonable, since it is due to the rewriting of policies performed at the beginning and ending stages of the aggregation process—specified in lines 3 and 36 of Algorithm 10.

Clearly, the results presented in this section indicate strong requirements of both processing time and space memory. However, we consider that these requirements are acceptable considering that all our approaches are performed off-line and they do not affect the performance of any component or network. Furthermore, we want to recall that the implementation of our proposal has been done by using a high level scripting language. We expect that the use of a more efficient language will considerably improve these results.

7 Conclusions

We presented in this paper a set of mechanisms for the managing of anomalies on distributed network security policies. More precisely, our proposal is intended for the discovery of anomalies in network security policies deployed over *firewalls* and *network intrusion detection systems* (NIDSs). Our approach was presented in two main blocks. We first presented, in Sect. 4, a set of algorithms for the management of anomalies within the configuration of single security components. We then presented, in Sect. 5, a set of algorithms for the management of anomalies between the configuration of different security components implementing a single, but distributed, security policy.

The advantages of our proposal are the following. First, our intra-component transformation process verifies that the resulting rules are completely independent between them. Otherwise, each rule considered as useless during the process is reported to the security officer, in order to verify the correctness of the whole process. Second, we can perform a second rewriting of rules, generating a configuration that only contains positive rules if the component default policy is negative, and negative rules if the default policy is positive. Third, the network model presented in Sect. 3 allows us to determine which components are crossed by a given packet knowing its source and destination, as well as other network properties. Thanks to this model, our approach better defines all the set of anomalies studied in the related work, and it reports, moreover, two new anomalies (*irrelevance* and *misconnection*) not reported, as defined in our work, in none of the other approaches.

The implementation of our approach in a software prototype, moreover, demonstrates the applicability of our work. We discussed this implementation, based on a scripting language [9], and presented an evaluation of its performance. Although the results of our experiments showed strong processing time and memory space requirements, we consider them reasonable and expect that the use of a more efficient implementation language will improve our initial evaluation.

As further work, we are currently working on an extension of our proposals in the case where the security architecture will also include virtual private network (VPN) tunnels and IPv6 devices, as well as those scenarios where there exist a cooperation between routing and tunneling policies. In parallel to this work, we are also studying how to extend our approach to the analysis of stateful policies.

Acknowledgements This work was supported by funding from the French Ministry of Research, under the *ACI DESIRS* project; and the Spanish Ministry of Science and Education, under the project *CONSOLIDER CSD2007-00004 "ARES"*.

References

1. Abou el Kalam, A., Baida, R.E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miège, A., Saurel, C., Trouessin, G.: Organization based access control. In: IEEE 4th International Workshop on Policies for Distributed Systems and Networks, pp. 120–131 Lake Come, (2003)
2. Adiseshu, H., Suri, S., Parulkar, G.: Detecting and resolving packet filter conflicts. In: 19th Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 3, pp. 1203–1212, Tel-Aviv, (2000)
3. Alfaro, J.G., Cuppens, F., Cuppens-Boulahia, N.: Aggregating and deploying network access control policies. In: 1st Symposium on Frontiers in Availability, Reliability and Security (FARES), 2nd International Conference on Availability, Reliability and Security (ARES2007), Vienna, pp 532–539 (2007)
4. Alfaro, J.G., Cuppens, F., Cuppens-Boulahia, N.: Management of exceptions on access control policies. In: 22nd IFIP TC-11 International Information Security Conference (IFIPsec2007), South Africa, May 2007, pp. 97–108. IFIP, Springer, Kluwer (2007)
5. Al-Shaer, E.S., Hamed, H.H.: Discovery of policy anomalies in distributed firewalls. In: IEEE INFOCOM'04, vol. 4, pp. 2605–2616, Hong Kong (2004)
6. Al-Shaer, E.S., Hamed, H.H., Masum, H.: Conflict classification and analysis of distributed firewall policies. IEEE J. Select. Areas Commun. **23**(10), 2069–2084 (2005)
7. Al-Shaer, E.S., Hamed, H.H.: Taxonomy of conflicts in network security policies. IEEE Commun. Magazine **44**(3), 134–141 (2006)
8. Bartal, Y., Mayer, A., Nissim, K., Wool, A.: Firmato: a novel firewall management toolkit. In: IEEE Symposium on Security and Privacy, pp. 17–31, Oakland (1999)
9. Castagnetto, J., et al.: Professional PHP Programming. Wrox Press Inc, ISBN 1-86100-296-3 (1999)
10. Chapman, D., Fox, A.: Cisco Secure PIX Firewalls. Cisco Press, Dublin (2001)
11. Cheswick, W.R., Bellovin, S.M., Rubin, A.D.: Firewalls and Internet Security: Repelling the Wily Hacker, 2nd edn. Addison-Wesley, (2003)
12. Cisco Systems, Inc.: Cisco Security Manager Product Information. [Online]. Available from: http://cisco.com/go/csmanager
13. Cuppens, F., Cuppens-Boulahia, N., Alfaro, J.G.: Detection and removal of firewall misconfiguration. In: Proceedings of the 2005 IASTED International Conference on Communication, Network and Information Security, vol. 1, pp. 154–162, (2005)
14. Cuppens, F., Cuppens-Boulahia, N., Alfaro, J.G.: Misconfiguration management of network security components. In: Proceedings of the 7th International Symposium on System and Information Security, Sao Paulo (2005)
15. Cuppens, F., Cuppens-Boulahia, N., Sans, T., Miege, A.: A formal approach to specify and deploy a network security policy. In: Second Workshop on Formal Aspects in Security and Trust, pp. 203–218, Toulouse (2004)
16. Gupta, P.: Algorithms for routing lookups and packet classification. PhD Thesis. Department of Computer Science, Stanford University (2000)
17. Hassan, A., Hudec, L.: Role based network security model: a forward step towards firewall management. In: Workshop on Security of Information Technologies, Algiers (2003)
18. Kurland, V.: Firewall builder. White Paper (2003)

19. Liu, A.X., Gouda, M.G.: Complete redundancy detection in firewalls. In: 19th Annual IFIP Conference on Data and Applications Security (DBSec-05), pp. 196–209, Storrs, (2005)
20. MITRE Corp.: Common Vulnerabilities and Exposures. [Online]. Available from: http://cve.mitre.org/
21. Northcutt, S.: Network Intrusion Detection: An analyst's Hand Book, 3rd edn. New Riders Publishing (2002)
22. Open Security Foundation.: Open Source Vulnerability Database. [Online]. Available from: http://osvdb.org/
23. Reed, D.: IP Filter. [Online]. Available from: http://coombs.anu.edu.au/~avalon/ip-filter.html
24. Roesch, M.: Snort: lightweight intrusion detection for networks. In: 13th USENIX Systems Administration Conference, Seattle (1999)
25. Sandhu, R., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Comput. **29**(2), 38–47 (1996)
26. Skybox Security, Inc.: Security Risk Management and Network Change Management Solution from Skybox Security
27. Welte, H., Kadlecsik, J., Josefsson, M., McHardy, P., et al.: The netfilter project: firewalling, nat and packet mangling for linux 2.4x and 2.6.x. [Online]. Available from: http://www.netfilter.org/
28. Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C., Mohapatra, P.: FIREMAN: a toolkit for FIREwall Modeling and ANalysis. In: IEEE Symposium on Security and Privacy, pp. 199–213, Oakland (2006)