Detection of firewall configuration errors with updatable tree

Tarek Abbes, Adel Bouhoula, Michaël Rusinowitch

International Journal of Information Security manuscript No.
(will be inserted by the editor)

Detection of Firewall Configuration Errors with Updatable Tree

Tarek Abbes · Adel Bouhoula · Michaël Rusinowitch

Received: date / Accepted: date

Abstract The fundamental goals of security policy are to allow uninterrupted access to the network resources for authenticated users and to deny access to unauthenticated users. For this purpose, firewalls are frequently deployed in every size network. However, bad configurations may cause serious security breaches and network vulnerabilities. In particular, conflicted filtering rules lead to block legitimate traffic and to accept unwanted packets. This fact troubles administrators who have to insert and delete filtering rules in a huge configuration file.

We propose in this paper a quick method for managing a firewall configuration file. We represent the set of filtering rules by a firewall anomaly tree (FAT). Then, an administrator can update the FAT by inserting and deleting some filtering rules. The FAT modification automatically reveals emerged anomalies and helps the administrator to find the adequate position for a new added filtering rule. All the algorithms presented in the paper have been implemented and computer experiments show the usefulness of updating the FAT datastructure in order to quickly detect anomalies when dealing with a huge firewall configuration file.

T. Abbes
Higher Institute of Electronic and Communication of Sfax, Tunisia
Tel.: +216-21240520
E-mail: abbes.tarek@gmail.com

A. Bouhoula
Higher School of Communication of Tunis, Tunisia
E-mail: adel.bouhoula@supcom.rnu.tn

M. Rusinowitch
Laboratory of Research in Computer Science and its Applications, France
E-mail: rusi@loria.fr

Keywords Firewall · Configuration update · Conflicts · Tree

1 Introduction

The Internet, and all its accompanying complications, has become integral to our lives. The security problems besetting the Internet are legendary and have been daily annoyances to many users. A firewall prevents the dangers of the Internet from spreading to the internal network. It offers many advantages such as restricting outgoing and incoming connections at a controlled point and preventing known malwares. Hence, the firewall is considered a core element in network security. However, as Rubin at al. stated in [29], "The single most important factor of your firewall's security is how you configure it.".

Firewalls are generally placed at the frontier between a private network, a demilitarized zone (DMZ) and the public network. They control the traffic from and toward these different zones by means of a set of filtering rules called access control list or ACL. The rules order is very important in ACL since it is tightly coupled with the intended security policy.

As the network services and communication means evolve, the number of ACL entries continuously raises. As a consequence, there are more conflicts between filtering rules. El-Shaer reports in [2] that all tested firewall configurations performed by different administrators, include many anomalies. Among the participants, 9 of them are expert. Moreover, Wool confirms in 2010 his main study's findings in 2004 that firewalls are still poorly configured, and a rule set's complexity is still positively correlated with the number of detected configuration errors. Finally, previous security facts out-

line the success of some worms and virus like Blaster [12], Sapphire [26] and Conficker [11] which are easily blocked with a well managed firewall. The number of suspect ports used by known trojans and reported on simovits site [13] reaches 878.

Therefore, the process of configuring the firewall becomes difficult and error prone. A new inserted rule may overlap with an existing one, which leads to a conflict or a redundancy depending on the rules actions (accept or deny). On one hand, rules are considered conflicted, if they define distinct actions. This type of anomaly leads to block an authorized network traffic or to allow unwanted packets. On the other hand, rules are redundant if they execute the same actions. The increase of the redundancy cases affects the firewall performance.

We propose in this paper a static analysis method to deal with the firewall anomalies problem. Our approach intends to detect anomalies in a firewall configuration file on one side, and to determine the consequences of adding or deleting filtering rules on the other side. The main idea is to represent the filtering rules on a minimal tree called firewall anomaly tree (FAT), then to supervise leaves. If there are several rules sharing the same path on the tree then we detect anomalies.

The advantages of our approach are the following. First of all, the filtering rules have a variable number of fields (dimensions) to match the packets. Secondly, unlike related works that are presented in Section II, our approach does not process the fields in a predefined fixed order. Instead, it selects fields whose values can differentiate between filtering rules. In this case, we simplify the FAT construction. The third benefit is the possibility to dynamically update the FAT by adding or deleting filtering rules. In this way, the administrator tests the modification of a firewall configuration. Even more, the FAT suggests to the administrator the adequate position to insert new rules.

Our previous work in [1] allows the FAT construction using an inference system, however without being able to update the Tree in case of a modification in the configuration file. In order to resolve this problem, our current work enhances the node datastructure and proposes a set of new algorithms to maintain the FAT.

The rest of this paper is organized as follows. Section 2 starts with an analysis of some related work. We present in Section 3 a framework for classifying firewall anomalies. We explain this classification with one example. Section 4 shows our preprocessing phase on firewall filtering rules. Then, we present in Section 5 a set of algorithms for constructing and updating the FAT. We devote Section 6 to detect and prevent firewall anomalies. We expose in Section 7 the experimental results. Finally we conclude the paper in Section 8.

2 Related works

Over the past few years, the study of firewall anomalies has received a large attention. Hamed and Al-Shaer give in [3] a comprehensive classification of security policy conflicts that might exist in a single or several security devices. Qian et al. present in [27] another framework to automate ACL analysis. They formally define the relations between 2 rules. Then they give sufficient conditions to detect redundancy and inconsistency. Ferraresi et al. [18] develop a new firewall anomalies classification of based on the severity concept. Besides, they propose two automatic conflict resolution algorithms.

In the context of anomalies detection, Al-Shaer et al. construct in [4] a policy tree to represent filtering rules and simultaneously highlight common paths which denote anomalies. The nodes in the tree represent some protocols fields, that come in a fixed order (protocol, source address, etc.), while edges are labeled by the values taken by these protocols fields. Hence, two different addresses with the same prefix are twice depicted despite the common part. The discovery of anomalies is done on the basis of pairs of filtering rules, which can be a long process in case of a long firewall configuration file. The authors extend their work in [5] to reveal policy anomalies in distributed firewalls. They propose a software tool called the "Firewall Policy Advisor" that simplifies the management of filtering rules.

In several related works, the binary decision tree is optimized to get a binary decision diagram (BDD). Yuan et al. present FIREMAN, a toolkit for modeling and analyzing firewall [33]. The authors define some sufficient conditions to detect 3 types of misconfigurations that are policy violation (with respect to a white and a black lists), inconsistency (shadowing, generalization and correlation cases) and inefficacy (redundancy and verbosity situations). Besides, Gouda and Lui employ the BDD to design a consistent, complete and compact firewall configuration [21]. They present a sequence of 5 algorithms going from reducing BDD until generating a simplified equivalent BDD. Again, the nodes represent protocols fields, while edges are the fusion a several value ranges to obtain a reduced firewall configuration. In another work, Liu et al. [25] give an effective SQL-like query language, called "Structured Firewall Query Language" (SFQL), for describing firewall queries. They also propose an algorithm for processing queries, that employs as core datastructure, the BDD. Hu et al. [23] use the BDD datastructures to convert a list of rules into a set of disjoint network spaces. They adopt a grid representation in order to identify policy anomalies and resolve them. Rezvani et al. [28] present two detection algorithms and a tool based on BDD in

order to discover firewall anomalies. Besides, they give two algorithms useful to resolve anomalies and decrease the number of rules without changing the policy.

Jeffrey et al. show in [24] that extra complex datastructures provided by BDDs are not necessary for analyzing only the firewall configuration. It suffices to employ search algorithms for Boolean satisfiability (SAT solvers). The authors ensure two properties in the firewall configuration: reachability (no shadowed rule) and no cyclicity (absence of loops). Ben Youssef et al. [10] verify the conformance of firewall configurations with respect to the security policies using a satisfiability solver modulo theories (SMT). They extend their work in order to support the case of distributed firewalls [9].

Cuppens et al. present in [16,17] an audit process to manage intra-firewall policy anomalies. By using relationships between the attributes of filtering rules (such as coincidence, disjunction, and inclusion), they succeed to detect and remove the configuration anomalies.The datastructure used in their work is a linked-list of initial size n, where n is the number of filtering rules. Each element is an associative array with the strings *condition*, *decision*, *shadowing*, and *redundancy* as keys to access each necessary value. The authors expand their approach to detect anomalies in network security policies deployed over firewalls and network intrusion detection systems [6]. Their approach has the advantage to analyze the whole set of rules and not only the relationship between two rules. Their implementation scales well with the increase of rules despite a theoretical complexity close to $O(m^n)$ where m is the number of attributes, and n is the number of rules.

Joaquin Garcia et al. develop in [19] a management tool called Mirage for the analysis and the deployment of network security policies. They conduct two types of analysis, a bottom-up analysis of already deployed network security equipments and a top-down refinement of a security policy towards the network equipments configurations. The two strategies allow the discovery and/or the prevention of inconsistencies in a single or between several network security components.

Basile et al. [8] propose a formal model to represent the policy in many security devices. Their formalism allows the identification and the removal of inconsistencies and anomalies. Besides, the modeled policy can be translated toward distinct resolutions strategies (e.g. First, Last or Most Specic matching) without changing the policy semantics. Eronen et al. propose in [15] an expert system to analyze firewall rules. The system knowledge base is expressed with a Prolog based programming language. The inference engine uses a constraint logic programming approach to determine which rules are candidates under specified conditions. The existence of many candidates rules may reveal anomalies.

The works cited above consider the numeric value of each protocol field and can be viewed as High-level approaches. Low-level packet filters decompose each value into a bit string. This approach have also received a lot of attention during the last decade. Gupta gives in [22] an extensive survey on such techniques. Eppstein et al. present a geometric approach that represents n 2-tuples filtering rules as n prioritized rectangles [14]. The detection of conflicts is performed $O(n^{3/2})$ in time and $O(n)$ in space. Baboescu and George et al. use a multidimensional binary tree to represent filtering rules [7]. Thanasegaran et al. [32] try to reduce the high memory and computation time consumption required by a geometrical approach. They propose a topological approach called BISCAL (Bit-vector based spatial calculus) to detect the conflicts in the firewall policies. The main idea is to perform a space division on each dimension then to compute some characteristic vectors and their Cartesian product.

We can conclude from this study that different approaches have been proposed to detect firewall anomalies. The tree data structure is first employed in [4,5], then the BDD is widely used [21,23,25,28,33]. Some of these works adopt a high level strategy in order to reveal conflicts. They use the concept of disjunction and inclusion of value ranges. These operations have to be implemented in an efficient way. Other works opt for a binary representation. As mentioned in [30], an optimization to this scheme consists in using multibit trees. If we have n filtering rules having a maximum size of W bits and a stride with length k bits , then the complexity of the lookup is $O(\frac{W}{k})$. However the memory consumption increases exponentially with k and is equal to $O(2^k n \frac{W}{k})$. Srinivasan et al. [31] use dynamic programming to determine, for a given prefix distribution, the optimal strides that minimize the memory consumption and guarantee a worst-case search time. Optimization is useful if the entries of the firewall configuration do not change at all or change very little.

In our work we pursue an intermediate-level approach by processing each field, byte per byte ($k = 8$). Besides, we sort the bytes looking for a rapid mismatch between filtering rules. Finally, our approach is scalable and allows the FAT update.

3 Firewall Anomalies specification

We define in this section the different forms of anomalies between firewall rules. For this purpose, we formalize a firewall ltering rule, then we introduce the notion of "domain".

3.1 Firewall filtering rule domain

A firewall filtering rule applies an action (accept, drop, etc.) on a network packet if a set of conditions are satisfied. These conditions, grouped in a header part, specify generally the belonging of some fields of the packet, to an interval or a set of values. If the firewall is stateful, the conditions are related to previous events such as a prior establishment of a TCP connection [20].

Definition 1 (Filtering Rule) A filtering rule R assigns an action A to a header H composed by a list of conditions. Formally it has the form:

$R : \langle f_1 \in D_1 \rangle \wedge \ldots \wedge \langle f_m \in D_m \rangle \Rightarrow \langle action = a \rangle$

We define $R[f_i]$ the set D_i, and $R[action]$ the action a.

We mean by a firewall filtering domain, the set of packets that are matched by the rule. Formally, we have:

Definition 2 (Domain) Let R be a filtering rule having the form: $\langle f_1 \in D_1 \rangle \wedge \ldots \wedge \langle f_m \in D_m \rangle \Rightarrow \langle action = a \rangle$. We define Dom(R) as the set $\{(x_1, x_2, \ldots, x_m) \mid \forall i \in [1..m] : x_i \in D_i\}$.

3.2 Firewall anomalies cases

The firewall anomalies appear when several rules match the same packet. We identify an incoherence when two overlapped rules define distinct actions. If these rules have the same action, we speak about redundancy. We can explain anomalies by the intersection of firewall filtering rules domains. We schematize in Fig. 1 all possible intersection cases. We suppose that rule R_i comes before rule R_j whose domain is shaded. The intersections can be:

- Partial: the first rule R_i matches only some packets matched by the second rule R_j and R_j has the same relation with R_i (Fig 1-c). In this case, we have either a partial redundancy (R_i and R_j define the same action) or a correlation (R_i and R_j define distinct actions). Formally we have:

$$\left\{ \begin{array}{l} Dom(R_i) \cap Dom(R_j) \neq \emptyset \\ Dom(R_i) \not\subset Dom(R_j) \\ Dom(R_i) \not\supset Dom(R_j) \end{array} \right\}$$

- Complete: R_i and R_j share the same domain (Fig 1-b). We have either a full redundancy (R_i and R_j define the same action) or a full incoherence (R_i and R_j define distinct actions). Formally we have:

$$\{\, Dom(R_i) = Dom(R_j) \,\}$$

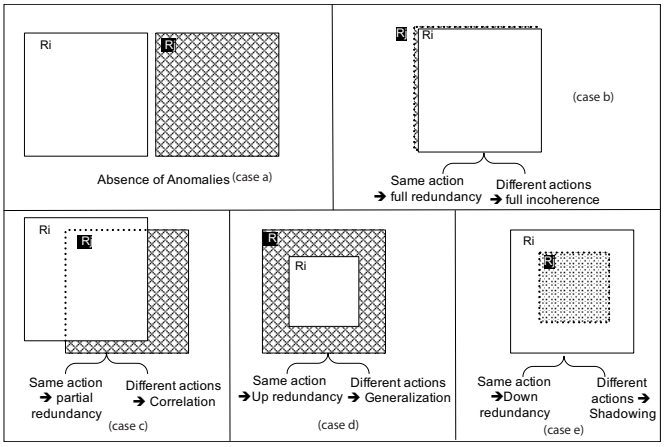

Fig. 1 Firewall filtering rules anomalies

- Complete for the up domain: the second rule R_j matches all packets already matched by the first rule R_i (Fig 1-d). If R_i and R_j define the same action, the anomaly is called up redundancy, else it is a generalization. Formally we have:

$$\{\, Dom(R_i) \subsetneq Dom(R_j) \,\}$$

- Complete for the down domain: the first rule R_i matches all packets matched by the second rule R_j (Fig 1-e). This is a serious anomaly case since the second rule is unnecessary. If R_i and R_j define the same action, the anomaly is called down redundancy, else it is a shadowing. Formally we have:

$$\{\, Dom(R_j) \subsetneq Dom(R_i) \,\}$$

Example 1 We show in Table 1 an example of a firewall configuration. IP addresses are represented using the CIDR notation. We can observe 6 types of anomalies :

- R1 is up redundant to R2.
- R1 and R4 are partial redundant.
- R2 and R3 are correlated.
- R3 is a generalization of R1.
- R4 is down redundant to R2.
- R4 is shadowed by R3.

Table 1 Firewall configuration

N	(addr, port) src	(addr, port) dst	action
R_1	(192.168.1.0/24, 80)	(10.0.0.0/8, *)	accept
R_2	(192.168.0.0/23,80)	(*, *)	accept
R_3	(192.168.0.0/22,80)	(10.0.0.0/8, *)	deny
R_4	(192.168.0.0/23,80)	(10.0.0.0/9, *)	accept

4 Firewall filtering rules preprocessing

We are interested in this section in representing firewall filtering rules in an adequate form. As noted in section 3, firewall anomalies are due to overlaps between filtering rules. Besides, we notice that that if two rules do not intersect in one dimension then they are coherent. Thus, we will transform our firewall filtering rules in such a manner that we start looking for no intersections. The idea is to first consider fields having single values over intervals.

4.1 Firewall filtering rules normalization

Our strategy involves representing all the fields of the filtering rules into a unique tree with a constant step equal to one byte. We suppose that the header part of a filtering rule can be expressed using 4 types of value.

1. a bit string prefix (e.g. $src\_ip = 192.168.0.0/23$)
2. a simple value (e.g. $dst\_port \in \{80\}$)
3. a set of values (e.g. $dst\_port \in \{80, 110\}$)
4. an interval (e.g. $dst\_port \in [0..1024]$)

Whatever the value types used in the header part of a filtering rule, we transform the latter into a sequence of couples called masked bytes and having the form $(byte, mask)_b$. The mask identifies the number of fixed bits in the byte. Thus, $(0, 8)_b$ is equivalent to the single value 0, $(0, 7)_b$ represents the set {0,1} and $(0, 0)_b$ is equivalent to the interval [0..255]. We can distinguish two types of masked bytes:

- A Complete masked byte with a mask equal to 8. It represents a unique value and creates less overlaps with other sets of values. For simplicity, we call it a **complete byte**.
- A partial masked byte with a mask length lower than 8. It is equivalent to a range, so it causes overlaps between rules. We call it a **partial byte**.

As an example we take the field "Source IPv4 address" which is coded into 4 bytes. We have four possible transformations:

1. The bit string prefix 192.168.0.0/25 (type 1) is represented as $(192, 8)_b, (168, 8)_b, (0, 8)_b, (0, 1)_b$,.
2. The simple value 192.168.0.1 (type 2) is represented as $(192, 8)_b, (168, 8)_b, (0, 8)_b, (1, 8)_b$.
3. The set of values {192.168.0.2, 192.168.0.3, 192.168.0.4} is transformed into $(192, 8)_b, (168, 8)_b, (0, 8)_b, (2, 7)_b$ and $(192, 8)_b, (168, 8)_b, (0, 8)_b, (4, 8)_b$.
4. The interval [192.168.0.32..192.168.0.63] is transformed into $(192, 8)_b, (168, 8)_b, (0, 8)_b, (32, 3)_b$.

4.2 Firewall filtering rules path

The normalization of the filtering rules converts them to sequences of masked bytes. In order to easily determine anomalies, we arrange these sequences, thus we obtain rules paths. Indeed, our goal is to plot firewall filtering rules on a tree called "Firewall Anomaly Tree" or FAT. We determine overlaps between rules if there are at a final node more than one candidate rule. The FAT branches are labeled by masked bytes. The complexity of the construction is due to the partial bytes (mask <8) since they represent ranges and may overlap with each other. However if such overlap occurs in one dimension (source IP address), it may not be the case in another dimension (source port) due to distinct complete bytes. Therefore, we postpone the processing of the partial bytes over the complete bytes. The rule path construction introduces some disorder in the sequence of masked bytes. In order to maintain the original positions of bytes, we associate to each byte a couple of integers called position. Besides, we call an "element" the combination of a masked byte and its position.

Definition 3 (position) A position associated to a byte is a couple of $(dim, ord)_o$ where dim refers to a field, ord represents the byte order in the field and o denotes a position.

Definition 4 (element) An element is a couple of a masked byte and a position having the form: $((byte, mask)_b, (dim, ord)_o)$

Example 2 The first element of rule R1 depicted in Table 1 is $((192, 8)_b, (1, 1)_o)$.

To obtain a unique rule path for each filtering rule, we introduce a partial order that organizes the sequence of elements.

Definition 5 (Partial order $\preccurlyeq$) let two elements $e_1 = ((v_1, m_1)_b, (d_1, o_1)_o)$ and $e_2 = ((v_2, m_2)_b, (d_2, o_2)_o)$. We define the partial order $\preccurlyeq$ as

$$e_1 \preccurlyeq e_2 \text{ iff } \begin{cases} \bullet m_2 < m_1 = 8 & \textbf{or} \\ \bullet((m_1 = m_2 = 8) \; or \; (m_1 < 8 \; and \; m_2 < 8)) & \\ \quad and \; (d_1 < d_2) & \textbf{or} \\ \bullet((m_1 = m_2 = 8) \; or \; (m_1 < 8 \; and \; m_2 < 8)) & \\ \quad and \; (d_1 = d_2) \; and \; (o_1 < o_2) & \end{cases}$$

Intuitively, the relation $\preccurlyeq$ implements our strategy to first consider complete bytes. We use it in the following to define the path of a filtering rule.

Definition 6 (Rule path) The path of a filtering rule "R" is a well ordered sequence of elements using the partial order $\preccurlyeq$. It is noted ***rule_path(R)***.

Example 3 Using Table 1, we have *rule_path(R_4)*= ((192,8)$_b$,(1,1)$_o$), ((168,8)$_b$,(1,2)$_o$), ((10,8)$_b$,(2,1)$_o$), ((0,8)$_b$,(3,1)$_o$), ((80,8)$_b$,(3,2)$_o$), ((0,7)$_b$,(1,3)$_o$), ((0,1)$_b$,(2,2)$_o$)); At the dimension 3, we have a source port that needs 2 bytes and is equal to 80.

5 Construction and update of Firewall Anomaly Tree

In order to detect firewall misconfigurations, we are based on a tree representation called Firewall Anomaly Tree (FAT). The main idea is to plot the paths of the filtering rules on a tree, then to inspect common paths which reveal the presence of overlaps. We devote this section to firstly define the FAT datastructure and then, the algorithms to update the tree.

5.1 Tree Datastructure

The FAT datastructure is mainly composed of interconnected nodes. A node represents one position extracted from the path of one or several filtering rules. Besides it includes some output edges used to interconnect children nodes.

Definition 7 (Node) A node N is an n-tuple (id, pos, edge[0..256], type, P, S, T, bptr, bval) where:

- id is a node identifier
- pos is the current position used to issue output edges from N
- type is the node type which can be F (Final) for leaves nodes and C (Complete), P (Partial) or PC (Partial Complete) for intermediate nodes. The type C is chosen if all the labels of the output edges are complete bytes, P (partial) if all the labels are partial and PC in the other cases
- edge [0..256] is an array representing output edges from N
- P is a set of candidate filtering rules, called primary rules
- S is a set of filtering rules such that Dom(S) $\supsetneq$ Dom(P). We call them secondary rules
- T is a set of filtering rules such that Dom(P) $\cap$ Dom(T) $\neq \emptyset$ and Dom(P) $\not\subset$ Dom(T) and Dom(T) $\not\supset$ Dom(P). We call them tertiary rules
- bptr is a back pointer to the parent node of N
- bval is the edge label between N and its parent node.

In particular the root node has an identifier 1, a set P containing all filtering rules, two empty sets S and T and void values btpt and bval. The fields pos and edge have to be computed using auxiliaries functions (subsection 5.2).

The identifier "id" and the sets "P", "S" and "T" are useful to find anomalies. To compute these sets, we need to have a link to parent node ("bptr"). Besides, the "pos" and the "edge" table are useful to construct the FAT.

5.2 Auxiliaries functions

In order to construct a child node, we have to create a new node (using *getNewNode* function) and assign a new identifier to the *id* field (using *getNewID* function). The fields *bptr* and *bval* can be easily computed if we know the edge from the parent node to the child node. The remaining fields are calculated by the following auxiliaries functions. We classify them into three categories: "computing the node's position", "calculating the edges' labels and the node's type" and "computing the candidate rules".

5.2.1 Computing the node's position

We introduce two functions to compute a node position. The *minpos* function is only used on the root node in order to extract the first position given a set of rules. The *nextpos* function is employed on the other nodes in order to compute for a set of rules, the position that follows the previous one.

Definition 8 (*minpos* function) Let SR be a set of filtering rules. We define the *minpos* function to extract the first position from the rule paths in SR.
$minpos(SR) = \{(dim_0, ord_0)_o \mid \exists R_i \in SR \text{ and } \exists e_0 = ((byte_0, mask_0)_b, (dim_0, ord_0)_o) \in rule\_path(R_i) \text{ such that } \forall R_j \in SR,\ \forall e \in rule\_path(R_j),\ e_0 \preccurlyeq e\}$

The *nextpos* function takes a set of rules and a position as arguments. It computes the next position if it exists, otherwise it returns an empty set $\emptyset$.

Definition 9 (*nextpos* function) Let SR be a set of filtering rules and pos_1 a position extracted from an element e_1. The *nextpos* function computes the position that follows pos_1. Formally, $nextpos(SR, pos_1) =$
- $(dim_2, ord_2)_o$ *if* $\exists R_i \in SR$ *and* $\exists e_2 = ((byte_2, mask_2)_b, (dim_2, ord_2)_o) \in rule\_path(R_i) \mid \forall R_j \in SR, \forall e \in rule\_path(R_j),$ *if* $e_1 \preccurlyeq e,$ *then* $e_1 \preccurlyeq e_2 \preccurlyeq e$
- $\emptyset$ *otherwise*

Example 4 Using the rules depicted in Table 1, we have:

- *minpos*({R1,R2,R3,R4})=(1,1)$_o$
- *nextpos*({R4},(1,2)$_o$)=(2,1)$_o$
- *nextpos*({R1,R4},(1,2)$_o$)=(1,3)$_o$
- *nextpos*({R1,R4},(2,2)$_o$)=$\emptyset$

5.2.2 Calculating the edges' labels and the node's type

We define the *proj* function in order to extract from a set of filtering rules (precisely set P of primary candidate rules) and at a given position, all possible bytes. Returned values will define labels of the output edges from the current node (the field *edge[]* in a node datastructure).

Definition 10 (proj function) Let SR be a set of filtering rules and $(dim, ord)_o$ a position. Then the *proj* function is defined as follows:
$proj(SR, (dim, ord)_o) = \{(byte, mask)_b \mid \exists R \in SR \, and \, \exists e = ((byte, mask)_b, (dim, ord)_o) \in rule\_path(R)\}$

Example 5 proj({R1,R2,R3,R4},$(1,1)_o$)=$\{(192,8)_b\}$;

Following the projection, we can decide the type of the node. In fact, if we obtain an empty set of masked bytes, we assign the type F (or "Final") to denote a leaf node. If all masked bytes returned by the *proj* function are complete, then the node has the type C (or "Complete"). However, if the masked bytes are partial, we attribute the type P (or "Partial"). In the other cases we have a mixture of complete and partial bytes, and we assign the type PC. In this situation, we will only consider the complete bytes in order to issue output edges. The processing of partial bytes is postponed in other nodes connected by an edge labeled ϵ.

5.2.3 Computing the candidate rules

We define other auxiliaries functions to compute the sets of candidates rules P, S and T in a node. The set P denotes to the primary candidate rules. It is calculated using the *cand* function.

Definition 11 (cand function) Let SR be a set of filtering rules, $(dim, ord)_o$ a position and $(byte, mask)_b$ a masked byte. The *cand* function is defined as :
$cand(SR, (dim, ord)_o, (byte, mask)_b) = \{R \mid R \in SR \, and \, \exists e = ((byte, mask)_b, (dim, ord)_o) \in rule\_path(R)\}$

We employ two functions to compute the sets S and T in a Node. Given an edge label l (having the form $(byte, mask)_b$) and a position, we use the *subcand* function to find all rules whose masked bytes are subset of l. The *supcand* function is used to calculate the candidate rules whose masked bytes are superset of l. Formally, *subcand* and *supcand* functions are defined as follows:

Definition 12 (subcand and supcand functions) Let SR be a set of filtering rules, $(dim, ord)_o$ a position and $l = (byte, mask)_b$ a masked byte then:
• $subcand(SR, (dim, ord)_o, (byte, mask)_b) = \{R \mid R \in SR \, and \, \exists e_1 = ((byte_1, mask_1)_b, (dim, ord)_o) \in rule\_path(R) such \, that \, (byte_1, mask_1)_b \subset (byte, mask)_b\}$
• $supcand(SR, (dim, ord)_o, (byte, mask)_b) = \{R \mid R \in SR \, and \, \exists e_2 = ((byte_2, mask_2)_b, (dim, ord)_o) \in rule\_path(R) such \, that \, (byte_2, mask_2)_b \supset (byte, mask)_b\}$

Example 6
cand({R1,R2,R3,R4},$(1,1)_o$,$(192,8)_b$)={R1,R2,R3,R4};
subcand({R2,R3},$(1,3)_o$, $(0,6)_b$) = {R2};
supcand({R1,R2,R3,R4},$(1,3)_o$, $(1,8)_b$) = {R2,R3,R4}

5.3 Development of nodes

We define a function "*Develop*" that is, when applied to a node, will generate all children nodes. Recursively, each created child node is developed using the same function until reaching the final nodes (leaves). Since each node represents a single position from a candidate filtering rule, final nodes correspond to last positions of the candidate rules paths.

The *Develop* function is described in Algorithm 1. It is composed of four main phases: compute position, projection, typing and unfolding. If the projection gives a mixture of complete and partial bytes, the node type is *PC* and we only consider complete bytes (Algorithm 1-Instruction α). The processing of partial bytes is postponed in other nodes (Algorithm 1-part β). Besides, Algorithm 1 uses the function *Candidate* (Algorithm 2) which combines the auxiliaries functions *cand*, *supcand*, *subcand* previously defined.

The development of the root node is sufficient to construct the FAT, and therefore to detect anomalies using Algorithm 10 discussed in Subsection 6.1. The FAT construction is rapid since several rules are processed simultaneously at the same position. Besides, we can stop early the FAT construction if we encounter a node having one candidate rule (Algorithm 1-Instruction γ). In this case, we don't have conflicts between primary, secondary and tertiary rules. We call this strategy "Cut". It simplifies the FAT but prevents its update. However, often administrators need to update the firewall configuration in order to react against recent threats and adapt to new services. We present in the next Subsections two operations for adding and eliminating filtering rules on an existing FAT. For this purpose, we will define two functions *insert()* and *delete()*.

5.4 Firewall rule insertion

We describe in Algorithm 3 the *insert()* function which traverses an existing FAT in order to include the path of a new filtering rule. Hence, the administrator can build the FAT by inserting one by one the filtering rules.

```
Function Develop
Input: Node=(−, −, −, −, P, S, T, bptr, bval)
Where Node has the form:
(d, pos, type, edge[ ], P, S, T, bptr, bval)
Output: Firewall Anomaly Tree (FAT)
begin
    Let id= Node.id ; pos=Node.pos ; type =
    Node.type ; edge [ ] = Node.edge[ ] ; P = Node.P ;
    S = Node.S ; T = Node.T ; bptr = Node.bptr ;
    bval=Node.bval

    id= getNewId()
     / *** Phase 1: Compute position ***/
    if (bptr == Null) then pos=minpos(P) else
    pos=nextpos(P, bptr.pos)

     / *** Cut Strategy (γ) when no need to
    update the FAT ***/
    if ( | P ∪ S ∪ T |== 1) then pos=FINAL   (γ)
    if (pos ≠ FINAL) then
        / *** Phase 2: Projection ***/
        V=Proj(P, pos)
        / *** Phase 3: Typing ***/
        if (V is complete) then
            type=C
        else
            if (V is partially complete) then
                type=PC
                V= V-{masked bytes}     (α)
            else
                type=P
            end
        end
        / *** Phase 4: Unfolding ***/
        Let V = {val_1, ..., val_n}
        for 1 ≤ i ≤ n do
            NewNode_i = getNewNode()
            edge[hash(val_i)]=NewNode_i
            NewNode_i.bptr = Node
            NewNode_i.bval = val_i
            Candidate (NewNode_i)
            Develop (NewNode_i)
        end
        if (type==PC) then
            /*PART β: Postpone partial bytes*/
            NewNode_{n+1} = newnode()
            Node.edge[256]=NewNode_{n+1}
            NewNode_{n+1}.bptr = Node
            NewNode_{n+1}.bval = ϵ
            NewNode_{n+1}.P = P \ ⋃_{1≤i≤n} NewNode_i.P
            NewNode_{n+1}.S=S
            NewNode_{n+1}.T=T
            Develop (NewNode_{n+1})
        end
    else
        type=F
    end
end
```

Algorithm 1: Node development

```
Function Candidate
Input: Node=(−, −, −, −, −, −, −, −, bptr, bval)
Where Node has the form:
(id, pos, type, edge[ ], P, S, T, bptr, bval)
Output: Node=(, −, −, −, P, S, T, bptr, bval)
begin
    Let prev=Node.bptr; bP=prev.P; bS=prev.S;
    bT=prev.T; bval=Node.bval; bpos=prev.pos
    Node.P=Cand(bP, bpos, bval)
    Node.S=Supcand(bP ∪ bS, bpos, bval)
    Node.T={Subcand(bS ∪ bT, bpos, bval) ∪
    Supcand(bT, bpos, bval) }
end
```

Algorithm 2: Computing candidate rules

Algorithm 3 describes the *insert()* function. The main challenge is to find on the FAT, the adequate position where to include the path of the new firewall rule noted *NR*. For this purpose, we extract the first position of the rule *NR* (noted *rpos*). Besides we traverse the FAT, node by node, starting from the root node and using the elements extracted from *rule_path(NR)*. The position of the visited current node of the FAT is noted *cpos*. We can distinguish three parts in Algorithm 3 depending on the values of *rpos* and *cpos*. For more clarity, the algorithm of each part is given separately.

In the first case, *rpos* is smaller than *cpos* (Case A in Algorithm 3). We construct a new node and we insert it before the current node of the FAT (Algorithm 4). We employ the function *develop()* to include the rule path.

In the second case, *rpos* is greater than *cpos* (Case B in Algorithm 3). We take the edge ϵ from the current node of the FAT. If it does not exist, we construct it (Algorithm 5) and we employ the function *develop()* to include the rule path.

In the final case, *rpos* is equal to *cpos* (Case C in Algorithm 3). We pursue the traversing of FAT as long as we find an output edge from the current node of FAT with a label equals to the masked byte of the new rule *NR*. If the condition does not hold, we construct a new node interconnected to the current node by a new edge labeled with the masked byte of *NR* at the position *rpos* (Algorithm 6). Then, we employ the function *develop()* to add the rule path.

Besides, we have to update the nodes of the existing FAT by incorporating, when it is feasible, the new added rule *NR* as a candidate rule. On the one hand, a new rule *NR* is a primary rule in the nodes whose paths are prefixes of the new rule path. This insertion is done in the first instructions of the Algorithms 5 and 6 (Instruction ρ). On the other hand, a new rule NR is considered as a secondary or a tertiary candidate rule in a node *up_node*, respectively if the masked byte of NR in-

```
Function insert
Input: NR ; Root
Output: Updated FAT
begin
    prev = NULL
    current=Root
    if (current==NULL) then
        Root=getNewNode ()
        Root.P={NR}
        Root.bptr=NULL
        Develop (Root)

    else
        stop=0
        foreach elt of rule_path(NR) do
            rpos=elt.pos ; cpos=current.pos

            /* rule position is smaller than current
            node position */
            if (rpos < cpos ) then
             | /*see Case A*/
            end

            /* rule position is greater than current
            node position */
            if (rops > cpos) then
             | /*see Case B*/
            end

            /* rule position is equal to current node
            position */
            if (rpos == cpos ) then
             | /*see Case C*/
            end
            if (stop ==1) then exit ()
        end
    end
end
```

Algorithm 3: Rule insertion in existing FAT

```
Case A of FAT update
begin
    NewNode=getNewNode ()
    if (prev≠NULL) then
        /* case A.1 */
        prev.edge[current.bval]=NewNode
        NewNode.bptr=prev
        NewNode.bval=current.bval

    else
        /* case A.2 */
        Root= NewNode
    end
    NewNode.edge[ϵ]=current
    current.bptr=NewNode
    current.bval = ϵ
    Candidate (NewNode)
    Develop (NewNode)
    stop=1
end
```

Algorithm 4: Case A : $rpos < cpos$

```
Case B of FAT update
begin
    current.P = current.P ∪ {NR}        (ρ)
    prev = current
    if (current.edge[ϵ] ≠ NULL) then
        /* case B.1 */
        current = current.edge[ϵ]

    else
        /* case B.2 */
        NewNode=getNewNode ()
        current.edge[ϵ] = NewNode
        NewNode.bptr=NewNode
        NewNode.bval=ϵ
        Candidate (NewNode)
        Develop (NewNode)
        stop=1
    end
    update (prev,NR)
end
```

Algorithm 5: Case B : $rpos > cpos$

```
Case C of FAT update
begin
    current.P = current.P ∪ {NR}        (ρ)
    prev=current
    if (current.edge[elt.val] ≠ NULL) then
        /* case C.1 */
        current = current.edge[elt.val]

    else
        /* case C.2 */
        NewNode=getNewNode ()
        current.edge[elt.val] = NewNode
        NewNode.bptr=current
        NewNode.bval=elt.val
        Candidate (NewNode)
        Develop (NewNode)
        stop=1
    end
    update (prev,NR)
end
```

Algorithm 6: Case C : $rpos = cpos$

cludes or is included in the input edge label of *up_node*. The *update()* function is described in algorithm 7.

We present in Fig. 2 the FAT update upon the insertion of three filtering rules R5, R6 and R7. The header parts of the filtering rules have four dimensions, source and destination addresses and source and destination ports. Let us note that tcp and udp ports are represented in 2 bytes. We employ Algorithm 6 for adding rule R5 since we match the case rpos=cpos=$(1,1)_o$. The insertion of R6 requires the application of Algorithm 5 because we find the case rpos = $(2,1)_o$ > cpos=$(1,1)_o$. Finally, the filtering rule R7 shows an example of rpos = $(1,4)_o$ < cpos=$(2,1)_o$, hence the use of Algorithm 4.

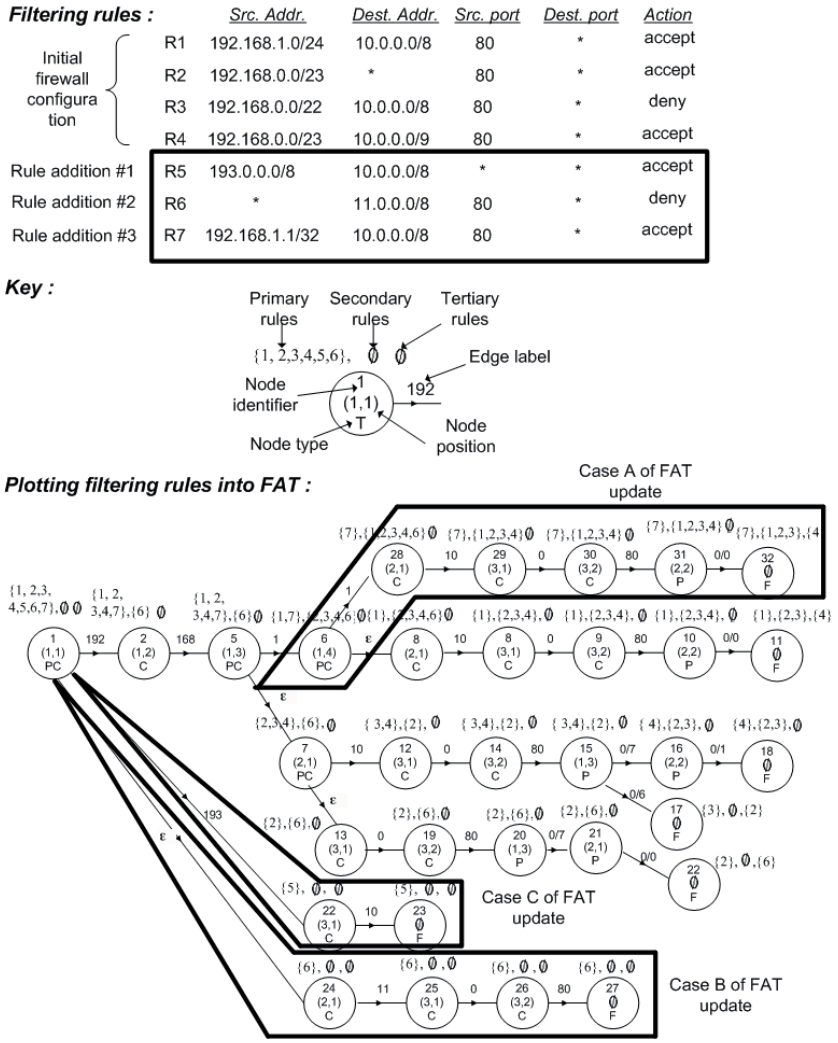

Fig. 2 FAT update

5.5 Firewall rule elimination

We present in this Subsection the procedure to eliminate a firewall filtering rule *DR* from the FAT. The main goal is to remove the rule path of *DR* which is provided by Algorithm 8. The *delete()* function traverses the FAT using the elements of *rule_path(DR)*. At each visited node, if there are no other filtering rules sharing the rule path with *DR*, then we assign to that node the type F which marks the end of the path. If the condition does not hold, we only remove *DR* from the primary candidates rules set of the node.

Besides, we have to remove the identifier of the rule *DR*, from the sets of secondary and tertiary candidate rules of all nodes in the FAT. For this purpose, we use the function *clean*, described in Algorithm 9 and is first invoked in Algorithm 8.

6 Firewall Anomalies discovery and complexity

We are interested in this section, in detecting and preventing firewall anomalies, essentially those considered severe (full redundancy, shadowing and down redundancy). The FAT construction allows the administrator discovering all anomalies. The prevention is triggered when inserting a new filtering rule. It gives the administrator the suitable place, where to add a new rule in order to avoid severe anomalies.

6.1 Firewall anomalies detection

In order to determine firewall anomalies, we have to inspect the leaf nodes of the FAT. Redundant rules are discovered if there are several primary candidate rules in a leaf node. The set of secondary candidate rules contains filtering rules whose domains include the domain of the primary candidate rule. The rules order is

```
Function update
Input: Node ; NR
begin
    foreach val such that (Node.edge[val]≠NULL)
    do
        /*Adding NR as a secondary candidate rule*/
        if NR ∈ Node.P ∪ Node.S then
            if proj (NR,Node.pos)⊃ val then
                up_node = Node.edge[val]
                up_node.S= up_node.S ∪ { NR }
                update (up_node,NR)
            end
        end

        /*Adding NR as a tertiary candidate rule*/
        if NR ∈ Node.S ∪ Node.T then
            if proj (NR,Node.pos)⊂ val then
                up_node = Node.edge[val]
                up_node.T= up_node.T ∪ { NR }
                update (up_node,NR)
            end
        end
    end
end
```

Algorithm 7: Update FAT nodes

important for deciding the type of the anomaly. If the identifier of a secondary candidate rule is lower than the identifier of the primary rule, we detect a shadowing or down redundancy depending on the rules actions However, if a secondary candidate rule comes after the primary rule, we encounter a generalization or an up redundancy. Eventually, the domain of tertiary candidate rule overlaps with that of the primary rule. Hence, we detect a correlation or a partial redundancy depending on the rules actions. We present in Algorithm 10 a detailed description of the anomalies detection process.

```
Function delete
Input: Root ; DR
begin
    current=Root
    foreach elt of rule_path(DR) do
        current.P= current.P - {DR}
        if (current.P = ∅) then
            current.edge[DR.elt.val]=NULL
            current.type=F
            exit ()

        else
            if (current.pos==DR.elt.pos) then
                if (current.type==P) then
                    foreach val' such that
                    current.edge[val']≠NULL do
                        if (val'⊊ DR.elt.val) then
                            c_node=current.edge[val']
                            clean (c_node)
                        end
                    end
                end
                current=current.edge[DR.elt.val]

            else
                /* current.pos < DR.elt.pos */
                foreach val' such that
                current.edge[val']≠NULL do
                    if (val'≠ ϵ) then
                        c_node=current.edge[val']
                        clean (c_node)
                    end
                end
                if (|current.edge[ϵ].P|==1) then
                    current.edge[ϵ]=NULL
                else
                    current=current.edge[ϵ]
                end
            end
        end
    end
end
```

Algorithm 8: Elimination of a rule from the FAT

6.2 Firewall Anomalies Prevention

The anomalies prevention is ensured when inserting a new rule NR. The inspection of the FAT's leaf nodes shows that NR appears only one time as a primary candidate rule (discussion 1). Besides, NR may occur as a secondary (discussion 2) or tertiary (discussion 3) candidate rule in other leaf nodes. In the following, our discussion is based on the sets P, S, T of primary, secondary and tertiary candidate rules of a leaf node.

- Discussion 1 = $NR \in P$: if $card(P) > 1$ then NR is full redundant to other rules in P and it is useless. If $card(P) = 1$, the domain of NR is included in the domains of the rules present in S. To avoid shadowing and down redundancy, NR must come before the rules in S. Formally, $order(NR) < min(order(S))$.
- Discussion 2 = $NR \in S$: In this situation, the domain of NR embraces the domains of the rules present in P. To avoid shadowing and down redundancy, the rule NR must come after the rules in P. Formally, $order(NR) > max(order(P))$.
- Discussion 3 = $NR \in T$: We have in this situation, a correlation or a partial redundancy between NR and the rules in P. Let us assume that $P = \{R_1, ..., R_k\}$ where R_i are sorted by their order of appearance in the configuration file. We note by Dom_i the domain formed by $\{R_1, ..., R_i\}$, i.e. $Dom_i = \bigcup_{j=1}^{i} Dom(R_j)$. We have to find the smallest $i^* \in \{1..k\}$ such that $Dom(NR) \subset Dom_i^*$. To avoid shadowing and down redundancy, NR must appear before rule R_i^*, which means that $order(NR) < i^*$.

```
Function clean
Input: Node ; DR
begin
    if DR ∈ Node.S then
        Node.S = Node.S -{DR}
    end
    if DR ∈ Node.T then
        Node.T = Node.T -{DR}
    end
    foreach val such that Node.edge[val]≠NULL do
        /* will stop at leaf node */
        clean (Node.edge[val])
    end
end
```

Algorithm 9: Clean FAT nodes

```
begin
    foreach leaf node L in the FAT do
        if | L.P ∪ L.S ∪ L.T |> 1 then
            foreach p in L.P do
                foreach p* in L.P do
                    if order(p*) < order(p) then
                        if action(p*)=action(p) then
                        p is full redundant to p*
                        else p is full incoherent to p*
                    end
                end
                foreach s ∈ L.S do
                    if order(s) < order(p) then
                        if action(s)=action(p) then
                        p is down redundant to s
                        else p is shadowed by s
                    else
                        if action(s)=action(p) then
                        p is up redundant to s
                        else s is a generalization to p
                    end
                end
                foreach t ∈ T do
                    if action(t)=action(p) then p is
                    partial redundant to t
                    else p is a correlation of t
                end
            end
        else No anomaly
    end
end
```

Algorithm 10: Anomalies discovery with FAT

6.3 Soundness and Completeness

Theorem 1 (Soundness) *Our detection algorithms are sound.*

Proof Let $R1$ and $R2$ be two conflicting rules. According to Algorithm 10, $R1$ and $R2$ are candidates rules in a leaf node, elt be L_n. We suppose that $R1 \in L_n.P$ and $R2 \in L_n.P \cup L_n.S \cup L_n.T$. We distinguish 3 cases:

- $\{R1, R2\} \subset L_n.P$. According to Algorithm 2 which uses the function *cand*() (Definition 11) we deduce that:
  - $\{R1, R2\} \subset (L_n.prev).P = L_{n-1}.P$
  - $R1$ and $R2$ share the same byte at $L_{n-1}.pos$.

  By using the recurrence, we conclude that $R1$ and $R2$ share all the bytes from the root node to the node L_n. Hence, $R1$ and $R2$ are redundant.
- $R1 \in L_n.P$ and $R2 \in L_n.S$. According to Algorithm 2 which uses the function *supcand*() (Definition 12) we deduce that:
  - $R1 \in (L_n.prev).P = L_{n-1}.P$ and $R2 \in L_{n-1}.P \cup L_{n-1}.S$
  - The masked byte of $R1$ at $L_{n-1}.pos$ is included in the masked byte of $R2$ at the same position.

  By using the recurrence, we conclude that $Dom(R1) \subsetneq Dom(R2)$. Hence, the anomaly exists.
- $R1 \in L_n.P$ and $R2 \in L_n.T$. According to Algorithm 2 which uses the function *subcand*() (Definition 12) we deduce that:
  - $R1 \in (L_n.prev).P = L_{n-1}.P$ and $R2 \in L_{n-1}.P \cup L_{n-1}.S \cup L_{n-1}.T$
  - The masked byte of $R1$ at $L_{n-1}.pos$ includes the masked byte of $R2$ at the same position.

  By using the recurrence, we conclude that $Dom(R1) \cap Dom(R2) \neq \emptyset$. Hence, the anomaly exists. □

Theorem 2 (Completeness) *Our detection algorithms are complete.*

Proof The completeness of our detection algorithms is achieved thanks to the partial order (Definition 5). All bytes of the filtering rules are sorted according to their dimensions (fields) and types (masked or complete). So all rules are represented in the FAT.

The *Develop()* method (Algorithm 1) finishes when the current position is "FINAL". Besides, the *insert()* method (Algorithm 3) defines 3 cases (A, B and C), all of them use the function *Develop()*. Finally, the *delete()* (Algorithm 4) converges since it eliminates some nodes and candidate filtering rules from a finite Tree. □

6.4 Complexity Study

We analyze in this Subsection, the complexity of our two construction strategies. We assume that the firewall configuration file contains n rules with m dimensions. If we subdivide each dimension into several blocks with a size of one byte, we can obtain up to d blocks per rule. Concerning space complexity, it is linear $O(n)$, since (1) each path must have at least one primary candidate rule (2) each rule is a primary candidate rule on a unique path on the FAT (3) edges are defined based on masked

bytes of primary candidate rules. We devote the rest of this Subsection to compute time complexity.

Develop Strategy

Using Algorithm 1, we notice that the construction time depends on three operations: the computation of the next position , the projection with the node typing and the unfolding (the loop "for"). We model this period time by $f_d(n)$ where d is the number of positions to be processed and n is the number of primary candidate rules.

$t_{develop\_strategy} = f_d(n)$
$= t_{position\_calculation} + t_{projection\&typing} + t_{unfolding}$
$= n + n + \alpha[t_{candidate} + f_{d-1}(\frac{n}{\alpha})]$

α is the number of child nodes derived from the node under processing. It varies between 1 and n. The maximum entropy (disorder) is obtained when α is equal to 2. $t_{candidate}$ is the time needed to execute Algorithm 2 on each child node. It depends on all candidates rules (primary, secondary and tertiary) at the parent node. Their numbers decrease when developing the nodes. For this reason, we note $t_{candidate}$ by C_i where i is the level of the development.

$$\begin{aligned} t_{develop\_strategy} = f_d(n) &= 2.n + 2[C_1 + f_{d-1}(\tfrac{n}{2})] \\ &= 2.n + 2[C_1 + 2.(\tfrac{n}{2}) + 2[C_2 + f_{d-2}(\tfrac{n}{4})]] \\ &= 2.n.log_2n + (\textstyle\sum_{i=1}^{log_2n} 2^i.C_i) + n.f_{d-log_2n}(1) \\ &= 2.n.log_2n + (\textstyle\sum_{i=1}^{log_2n} 2^i.C_i) + n.[2 + f_{d-log_2n-1}(1)] \\ &= 2.n.log_2n + (\textstyle\sum_{i=1}^{log_2n} 2^i.C_i) + n.[2.(d - log_2n)] \\ t_{develop\_strategy} &= 2.n.d + (\textstyle\sum_{i=1}^{log_2n} 2^i.C_i) = O(n.log_2n) \end{aligned}$$

Let us note that we can reduce the amount $2.n.d$ by choosing a block size larger than a byte. However, we obtain a larger C_i since we get more candidates rules, due to overlaps, at each stride.

Insert Strategy

We assume having a FAT composed of n filtering rules. We would like to add a new filtering rule numbered $n+1$. We have to employ Algorithms 3, 4, 5, 6 and 7. There are at most d positions on The FAT. We note by $a(d)$ the time needed to find the right position where we will develop the new rule. Besides, we refer by $u(n)$, the time period to update candidates rules on the existing FAT of n rules. Analytically, we have:

$$\begin{aligned} t_{insert\_n+1} &= t_{access} + t_{develop} + t_{update\_n} \\ &= a(d) + f_{d-a(d)}(1) + u(n) \\ &= a(d) + 2.(d - a(d)) + u(n) \\ t_{insert\_n+1} &= 2.d + u(n) - a(d) = O(n) \end{aligned}$$

If we construct the whole FAT using the insert strategy, we obtain the following complexity:

$t_{insert\_strategy} = \sum_{i=1}^{n} t_{insert\_i}$
$t_{insert\_strategy} = 2.n.d + \sum_{i=0}^{n-1}(u(i) - a(d)) = O(n^2)$

We notice that adding a new rule on an existing FAT is fast. If we neglect the amount $2.d - a(d)$, the period time for inserting a new rule depends essentially on $u(n)$. In addition, the two construction strategies share the same amount $2.n.d$. They differ on the way to compute candidates rules. This operation is less solicited with the develop strategy ($n.log_2n$ instead of n^2).

7 Experimental results

We develop the set of algorithms presented in this paper using the C language. These algorithms do not impose a specific number or type of fields in the filtering rules. On the contrary, we employ the concept of rule path that represents a concatenation of an arbitrary number of bit blocks. These blocks are extracted from the attributes of the filtering rules and are sorted to avoid, as much as possible, the processing of overlaps. Due to its flexibility, we think that our approach can be generalized to many other problems (packet routing, signature based intrusion detection, etc.).

We conduct our experiments on a laptop Intel Core i3 processor, 4GB RAM, running Microsoft Windows Seven. We perform two types of experiments. First of all, we execute our program on a short size firewall configuration supposed to contain all anomalies types. Our objective is to ensure the detection of all incoherences. Afterwards, we carry out several experiments with a variable size configuration file. We evaluate the behavior of our verification tool in overload conditions.

7.1 Detection results on a short configuration

In order to verify the results of our algorithms, we enhance the configuration file given in Fig. 2, by 3 rules R8, R9 and R10. Table 2 summarizes all firewall filtering rules.

We provide in Fig. 3 a summary of our firewall configuration analysis. Our algorithm reveals 22 anomalies, among them there are 2 shadowing and 3 down redundancies considered as severe errors. We report in Table 3 a detailed listing about these anomalies. We notice that R4 and R7 are not useful, they are shadowed by R3. In addition the default filtering rule (R10) produces a lot of minor anomalies having the types generalization or up redundancy.

Table 2 Simple Firewall configuration

N	Src Addr.	Dest Addr.	Src Port	Dest Port	Action
R1	192.168.1.0/24	10.0.0.0/8	80	*	Accept
R2	192.168.0.0/23	*	80	*	Accept
R3	192.168.0.0/22	10.0.0.0/8	80	*	Deny
R4	192.168.0.0/23	10.0.0.0/9	80	*	Accept
R5	193.0.0.0/8	10.0.0.0/8	*	*	Accept
R6	*	11.0.0.0/8	*	*	Accept
R7	192.168.1.1/32	10.0.0.0/8	80	*	Accept
R8	10.0.0.0/8	172.16.0.0/12	80	*	Accept
R9	193.0.0.0/25	*	80	*	Deny
R10	*	*	*	*	Deny

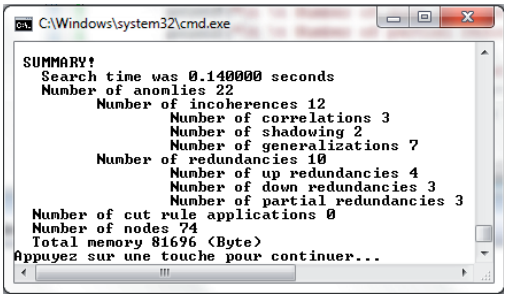

Fig. 3 Program execution on a simple configuration file

Table 3 Anomalies in a simple firewall configuration

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
R1		UR		PR						
R2						C				
R3	G	C								UR
R4		DR	S							
R5										
R6										UR
R7	DR	DR	S	PR						
R8										
R9					C	PR				UR
R10	G	G		G	G		G	G		

Keys: C: Correlation; S: Shadowing; G: Generalization
UR: Up Redundancy; DR: Down Redundancy; PR: Partial Redundancy

7.2 Detection results on a large configuration

We get several filtering rules from 21 firewalls that are deployed in different institutes and universities. The size of the configuration files varies between 15 and 112 rules. Some firewalls are deployed between the perimeter network (DMZ) and the outside network, while others are implemented between internal network segments. We find the two operation modes: routed and transparent modes. The configurations aim to enforce network policies. They authorize some ordinary data traffic (http, smtp, ssh,...) and some control traffic (routing protocols, vpn protocols, snmp,...). They also secure the boundaries between different zones, and filter malicious softwares, undesirable applications, and abnormal traffic (spoofing, dos,...). We rewrite local IP addresses in these configuration files to appear protecting the same network. Then, we combine the filtering rules to obtain a huge configuration file, able to evaluate our approach in overload conditions.

We construct our FAT by two possible ways. The first method takes all filtering rules and applies the *develop()* function with the Cut strategy. The second method constructs the FAT, rule by rule, using the *insert()* function. During our experiments, we supervise the number and the nature of discovered anomalies, the FAT construction time and the memory consumption.

In Fig. 4, we inspect the variation of discovered anomalies with respect to the number of filtering rules. We notice that the number of incoherences in a firewall rises rapidly with the complexity of configuration. We detail in Fig. 5 the nature of these anomalies. We notice that the partial intersection of two rules domains is the main cause leading to correlation and partial redundancy. Shadowing and down redundancy, which are considered as severe faults in a configuration, may also appear in a huge firewall configuration.

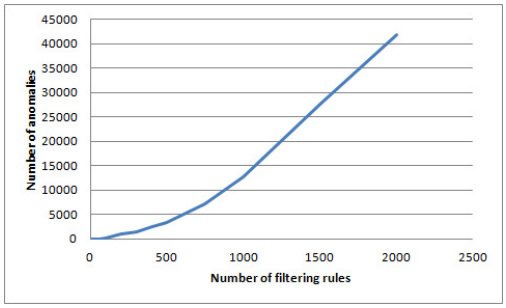

Fig. 4 Anomalies increase with respect to filtering rules

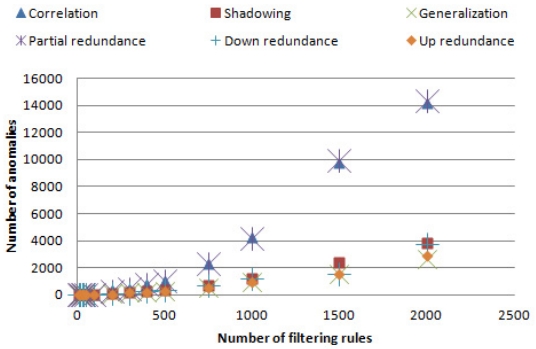

Fig. 5 Nature of discovered anomalies

We measure in a second phase of our evaluation, the FAT construction using the two aforementioned methods (Fig. 6). As expected, the *develop()* method is faster

since it processes in one step, all candidate rules on each node at a given position. However when using the *insert()* method we have to visit the same node, as many times as the new inserted rules, are candidate in that node. The *develop()* method fits a static firewall configuration. Nevertheless, it is better to rely on the second construction method if the content or the number of filtering rules frequently changes . Indeed, Fig. 6 shows that the update of a FAT representing 2000 rules, takes only 0,59 seconds.

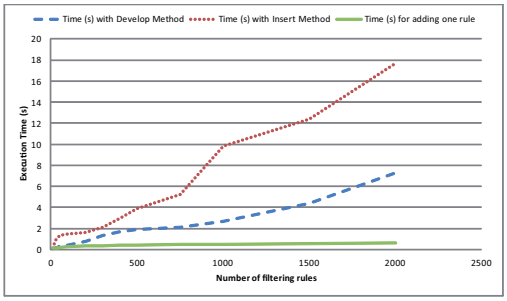

Fig. 6 Time execution for FAT construction and update

Our final experiment evaluates the memory consumption needed to represent the FAT (Fig. 7). While the two construction methods requires almost the same amount of memory, we can notice that the *develop* method has a small advantage. This is due to the application of **Cut** strategy in the first construction process. In fact, Fig. 8 shows the number of nodes generated by the two methods and the number of Cut strategy invocations.

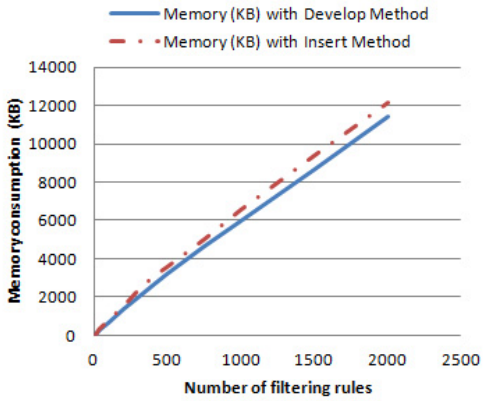

Fig. 7 Memory consumption with the two construction methods

8 Conclusion

The efficiency of a firewall depends on the defined security policy. However, a detailed configuration implies

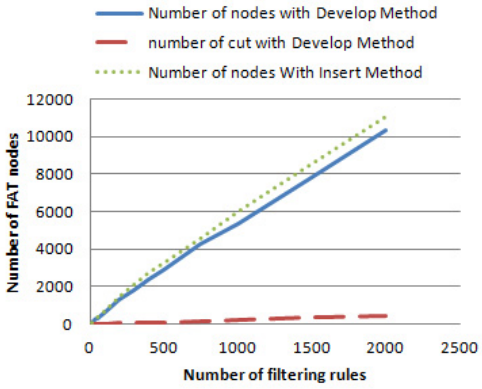

Fig. 8 Nodes generation with the two construction methods

the presence of many filtering rules which increases the risk of incoherence. A manual verification of a firewall configuration seems to be difficult. Administrators have to rely on automatic tools to check up and maintain their firewalls.

We propose in this paper a set of algorithms to represent filtering rules in a tree datastructure called FAT (Firewall Anomaly Tree) and to update it according to the evolution of the firewall configuration. Our strategy for building the tree does not process separately the value each field of the filtering rules. Instead, the value is divided into a set of complete bytes, and a partial byte that represents an interval. Our strategy postpones the processing of the partial bytes over the complete bytes. Besides, the construction can be achieved by treating all rules together, or one by one. On one hand, we show that the first method is faster and it is more suitable for a first verification of a firewall configuration. On the other hand, the second method is appropriate for continuously monitoring the configuration file since it allows a fast tree update. By means of FAT, the administrator can discover all anomalies and choose the adequate position to insert a new filtering rule.

As a future work, we can apply our strategy to discover anomalies between multiple firewalls or with heterogeneous security equipments (Intrusion Detection / Prevention System, Honey Pot, Security gateway). Our second objective is to exploit the tree datastructure to find an adequate permutation that reduces the number of incoherences.

References

1. T. Abbes, A. Bouhoula, and M. Rusinowitch. An inference system for detecting firewall filtering rules anomalies. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 2122–2128, 2008.
2. E. Al-Shaer and H. Hamed. Modeling and management of firewall policies. *IEEE Transactions on Network and Service Management*, vol. 1, no. 1, pages. 2-10, 2004.

3. E. Al-Shaer and H. Hamed. Taxonomy of conflicts in network security policies. *IEEE Communications Magazine*, vol. 44, no. 3, pages 134-141, March 2006.
4. E.S. Al-shaer and Hazem H. Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *Proceedings of IFIP/IEEE Eighth International Symposium on Integrated Network Management*, pages 17–30, 2003.
5. E.S. Al-shaer and Hazem H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of IEEE INFOCOMM*, pages 2605–2616, 2004.
6. J.G. Alfaro, N. Cuppens-Boulahia, and F. Cuppens. Complete analysis of configuration rules to guarantee reliable network security policies. *International Journal of Information Security*, vol. 7, no. 2, pages 103-122, March 2008.
7. F. Baboescu and G. Varghese. Fast and scalable conflict detection for packet classifiers. In *Proceedings of the 10th IEEE International Conference on Network Protocols*, ICNP '02, pages 270–279, Washington, DC, USA, 2002. IEEE Computer Society.
8. C. Basile, A. Cappadonia, and A. Lioy. Network-level access control policy analysis and transformation. *IEEE/ACM Transactions on Networking (TON)*, vol. 20, no. 4, pages 985-998, August 2012.
9. N. BenYoussef and A. Bouhoula. Automatic conformance verification of distributed firewalls to security requirements. In *Proceedings of the 2010 IEEE Second International Conference on Social Computing*, SOCIALCOM '10, pages 834–841, 2010.
10. N. BenYoussef, A. Bouhoula, and F. Jacquemard. Automatic verification of conformance of firewall configurations to security policies. In *Proceedings of the 14th IEEE Symposium on Computers and Communications, ISCC 2009*, pages 526–531, 2009.
11. CERT Coordination Center. Conficker worm targets microsoft windows systems. http://www.us-cert.gov/cas/techalerts/TA09-088A.html, April 2009.
12. CERT Coordination Center. CERT Advisory CA-2003-20 W32/Blaster worm. http://www.cert.org/advisories/CA-2003-20.html, August 2003.
13. Simovits Consulting. Trojan list sorted on trojan port. http://www.simovits.com/trojans/trojans.html.
14. D. Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 827–835, Philadelphia, PA, USA, 2001.
15. P. Eronen and J. Zitting. An expert system for analyzing firewall rules. In *Proceedings of the 6th Nordic Workshop on Secure IT Systems (NordSec 2001)*, pages 100–107, 2001.
16. N. Cuppens-Boulahia F. Cuppens and J.G. Alfaro. Detection and removal of firewall misconfiguration. In *Proceedings of the International Conference on Communication, Network and Information Security, IASTED'05*, volume 1, pages 154–162, 2005.
17. N. Cuppens-Boulahia F. Cuppens and J.G. Alfaro. Misconfiguration management of network security components. In *Proceedings of the 7th International Symposium on System and Information Security*, pages 154–162, 2005.
18. S. Ferraresi, S. Pesic, L. Trazza, and A. Baiocchi. Automatic conflict analysis and resolution of traffic filtering policy for firewall and security gateway. In *Proceeding of the IEEE International Conference on Communications, ICC '07*, pages 1304–1310, 2007.
19. J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, and S. Preda. Mirage: a management tool for the analysis and deployment of network security policies. In *Proceedings of the 5th international Workshop on data privacy management, and 3rd international conference on Autonomous spontaneous security*, pages 203–215, 2011.
20. M. Gouda and A. Liu. A model of stateful firewalls and its properties. In *Proceedings of the IEEE International Conference on Dependable Systems and and Networks*, pages 128–137, 2005.
21. M. Gouda and X. Liu. Firewall design: Consistency, completeness, and compactness. In *Proceedings of the 24th International Conference on Distributed Computing Systems, ICDCS'04*, pages 320–327, 2004.
22. P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network Magazine of Global Internetworking*, vol. 15, no. 2, pages 24-32, 2001.
23. H. Hu, G.L Ahn, and K. Kulkarn. Detecting and resolving firewall policy anomalies. *IEEE Transactions on Dependable Secure Computing*, vol. 9, no. 3, pages 318-331, May 2012.
24. A. Jeffreyand and T. Samak. Model checking firewall policy configurations. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks*, POLICY '09, pages 60–67, 2009.
25. A. Lui and M. Gouda. Firewall policy queries. *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 6, pages 766-777, 2009.
26. D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the sapphire/slammer worm. http://www.caida.org/publications/papers/2003/sapphire/sapphire.html, 2003.
27. J. Qian. Acla: A framework for access control list (acl) analysis and optimization. In *Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues of the New Century*, page 4, 2001.
28. M. Rezvani and R. Aryan. Analyzing and resolving anomalies in firewall security policies based on propositional logic. In *Proceedings of 13th IEEE International Multitopic Conference, INMIC 2009*, pages 1–7, 2009.
29. A.D. Rubin, D. Geer, and M.J. Ranum. *Web Security Sourcebook: A Complete Guide to Web Security Threats and Solutions*. Wiley Computer Publishing, 1997.
30. M. A. Ruiz-Sanchez, E . W. Biersack and W. Dabbous Survey and taxonomy of IP address lookup algorithms. In *Journal IEEE Network: The Magazine of Global Internetworking*, vol. 15, Issue 2, page 8-23, March 2001.
31. V. Srinivasan, V. and G. Varghese Fast address lookups using controlled prefix expansion. In *Journal ACM Trans. Comput. Syst.*, vol. 17, no. 1, page 1-40, February 1999.
32. S. Thanasegaran, Y. Yin, Y. Tateiwa, Y. Katayama, and N. Takahashi. A topological approach to detect conflicts in firewall policies. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, IPDPS 2009*, pages 1–7, 2009.
33. L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the IEEE Symposium on Security and Privacy, SP '06*, pages 199–213, 2006.