Safe and Efficient Strategies for Updating Firewall Policies*

Zeeshan Ahmed[1], Abdessamad Imine[2], and Michaël Rusinowitch[1]

[1] INRIA Nancy Grand Est
{ahmedzee,rusi}@loria.fr
[2] INRIA Nancy Grand Est & Nancy-Université, France
imine@loria.fr

Abstract. Due to the large size and complex structure of modern networks, firewall policies can contain several thousand rules. The size and complexity of these policies require automated tools providing a user-friendly environment to specify, configure and safely deploy a target policy. When activated in online mode, a firewall policy deployment is a very difficult and error-prone task. Indeed, it may result in self-Denial of Service (self-DoS) and/or temporary security breaches. In this paper, we provide correct, efficient and safe algorithms for two important classes of policy editing. Our experimental results show that these algorithms are fast and can be used safely even for deploying large policies.

Keywords: Firewall Policy Management, Firewalls, Network Security.

1 Introduction

Motivation. A firewall is an essential component of any network security infrastructure. Network firewalls are devices or systems that control the flow of traffic between networks employing different security postures [17]. The network traffic flow is controlled according to a firewall policy. The large size and complexity of modern networks result in large and complex firewall policies. Firewall policies containing 10K rules are not uncommon and firewalls configured with as many as 50K rules exist [22].

A policy *deployment* is the process by which the running policy is replaced by a new policy. In most mission-critical network-based applications (such as Voice-OverIP and online e-commerce), the *deployment should be performed in online mode in order to keeping these applications available and accessible*. Different firewalls support different policy editing commands: inserting a new rule, appending a new rule at the end, deleting a rule and moving a rule from one position to another one. Due to intervening nature of firewall rules, correct deployment of such large policies is a very difficult and error-prone task.

For example, consider the initial policy I (the running policy) and the target policy T (the policy to be deployed) given in Figure 1. To obtain T, if we delete rules a and c and next we insert these rules at right positions the deployment will be slow. A deployment

* This work has been supported by the INRIA ARC 2010 ACCESS and FP7-ICT-2007-1 Project No.216471 AVANTSSAR.

S. Katsikas, J. Lopez, and M. Soriano (Eds.): TrustBus 2010, LNCS 6264, pp. 45–57, 2010.

should be *efficient* i.e. it should issue the minimum number of commands to accomplish the deployment. A slow deployment is unpleasant for users and may partly defeat the purpose of deployment [22]. In our example, instead of four commands, we have only to issue two move commands (either moving a and c or moving b and d).

Now consider a packet p with source address of 10.1.1.1. Clearly, rule a denies p, while rule c accepts it. It is evident that both I and T deny p. If we first move a, then we get the intermediate running policy $R = [b, c, d, a]$. Now rule c appears before rule a, while it appears after a in both I and T. The running policy accepts p, which is denied by both I and T. A deployment is *safe* if no legal packet is rejected and no illegal packet is accepted during the deployment. A naive deployment strategy may result in self-Denial of Service (self-DoS) and/or temporary security breaches. Deployment safety is a new and challenging area of research.

I	**T**
a. deny tcp 10.1.1.0/24 any	b. permit ip 192.168.1.0/24 any
b. permit ip 192.168.1.0/24 any	d. permit tcp 192.168.2.0/24
c. permit tcp 10.1.0.0/16 any	a. deny tcp 10.1.1.0/24 any
d. permit tcp 192.168.2.0/24 any	c. permit tcp 10.1.0.0/16 any

Fig. 1. Example of Policy Deployment

Related Work. Much research has addressed policy specification [10, 7, 15], conflict detection [18, 14, 9], and optimization [20, 16]. However, very little research has been done on firewall policy deployment. To the best of our knowledge, the work presented in [22] is the first that addresses deployment safety and efficiency. In [22], the authors classify policy editing languages into two representative classes, Type I and Type II, and provide deployment algorithms for both types of languages. In [6], it is shown that these algorithms have serious flaws related to efficiency and safety properties.

Contributions. In this paper, we present a safety formalization that can be used as a basis for formulating safe deployment strategies (Section 3). We provide a linear algorithm for Type I deployment (Section 4). This algorithm is most-efficient and it ensures that either no legal traffic is rejected or no illegal traffic is permitted. We also give an approximately linear, most-efficient and safe algorithm for Type II languages (Section 5). Finally, we present experimental results of our Type II algorithm, and give conclusions.

2 Overview of Firewalls

A firewall is a perimeter security device that filters packets that traversing across the boundaries of a secured network. The filtering decision is based on a *policy* defined by the network administrator. A firewall policy is an ordered list of rules. A firewall rule r defines an action, typically *accept* or *reject*, for the set of packets matching its criteria. Most of firewalls filter traffic according to first-match semantics. When a packet p arrives, it is compared against the rules in a top-down fashion until a matching rule is found and the process is repeated for the following packet. All policies admit a hidden

match-all default rule at the end. Therefore, when a packet does not match a rule in the policy, then the default action is followed. In most firewalls, the default rule is *deny-all*, however a *permit-all* default rule is also possible. A rule is given with a set of fields, where each field can have an atomic value or a range of values. It is possible to use any field of IP, UDP, or TCP headers [22]. However, the following five fields are most commonly used: protocol type, source IP address, source port, destination IP address and destination port [11].

Any field in packet's header can be used for the matching process. However, the same five fields are most commonly used. In a packet, each of these fields has an atomic value. If all the fields of a packet p match with the corresponding fields of a rule r, then p is accepted or rejected according to the decision field of r. If p does not match to any rule in policy, then the default match-all rule is applied. Most of firewalls do not allow identical rules in the same policy. *Therefore, we assume this restriction and do not allow duplication of rules within a policy.*

3 Policy Deployment

Policy deployment is the process by which policy editing commands are issued on firewall, so that the target policy becomes the running policy.

3.1 Policy Editing Languages

A network administrator or a management tool issues commands on firewall to transform the running policy R into the target policy T. The set of commands that a firewall supports is called its policy editing language. Policy editing languages can be classified into two representative classes [22]: Type I and Type II.

Type I Editing. This type supports only two commands, append and delete. Command *(app r)* appends a rule r at the end of the running policy R, unless r is already in R, in which case the command fails. Command *(del r)* deletes r from R, if it is present. As Type I editing can transform any running policy into any target policy, therefore it is complete. Most older firewalls and some recent firewalls, such as FWSM 2.x [1] and JUNOSe 7.x [5], only support Type I editing.

Type II Editing. This type allows random editing of firewall policy. It supports three operations: *(ins i r)* inserts rule r as the ith rule in running policy R, unless r is already present; *(del i)* deletes ith rule from R; *(mov i j)* moves the ith rule to the jth in R position. Type II editing can transform any running policy into any target policy without accepting illegal packets or rejecting legal packets. It is obvious that for a given set of initial and target policies, a Type II deployment normally uses fewer editing commands than an equivalent Type I deployment. Examples of Type II editing firewalls include SunScreen 3.1 Lite [13] and Enterasys Matrix X [2].

3.2 Deployment Efficiency

A deployment is *most-efficient* if it utilizes the minimum number of editing commands in a given language, to correctly deploy a target policy on a firewall. Therefore for a given

deployment scenario, a most-efficient Type I (resp. Type II) deployment uses the minimum number of *append* and *delete* commands (resp. *insert*, *delete* and *move* commands). Usually a policy editing command takes constant time, and the variation in deployment time is negligible for different types of commands. Accordingly, the most-efficient deployment minimizes the overall deployment time. Deployment efficiency for Type I and Type II languages are discussed in more detail in Sections 4 and 5 respectively.

3.3 Deployment Safety

A deployment is *safe* if no security hole is introduced and no legal traffic is denied at any stage during the deployment. A temporary security hole permits malicious traffic to pass through the firewall and this may cause serious damage to the network infrastructure. Similarly, rejection of legal traffic during deployment may interrupt critical operations and result in serious losses. This is like inflicting a self-DoS attack and hence it is intolerable in mission-critical networks, even for a short duration of time.

Deployment safety is particularly important in cases where many changes are to be made to a large firewall policy. In such cases, a deployment can last up to several minutes, which may provide sufficient opportunity to a malicious party to exploit a vulnerability. Fast spreading worms, such as Conficker [4] and Slammer [3], can infect million of systems across the globe within minutes. Furthermore, a skilled hacker can use automated tools to continuously probe for vulnerabilities and instantly exploit these as they appear during an unsafe deployment.

The first fundamental work on deployment safety is presented in [22] where a safe deployment formalization is presented. The formalization defines a safe deployment as follows: Policy A is *denial-safe* w.r.t. policies B and C iff every packet that A denies is also denied by B or C. A deployment is denial-safe iff at every moment during the deployment the running policy is denial-safe w.r.t. the initial and the target policies. Similarly, policy A is *permission-safe* w.r.t. policies B and C iff every packet that A permits is also permitted by B or C. A deployment is permission-safe iff at every moment during the deployment the running policy is permission-safe w.r.t. to initial and the target policies.

Definition 1. ***Deployment Safety.*** *A policy is* safe *iff it is both denial-safe and permission-safe. A deployment is* partial-safe *if it is either permission-safe or denial-safe but not both w.r.t. initial and final policies.*

In the rest of this paper, we denote the initial policy by I and the target policy by T. A firewall has a new running policy every time an editing command is applied. Thus deployment can be viewed as a sequence of running policies $I = R_0, R_1, ..., R_{n-1}, R_n = T$, where R_{i+1} is derived by applying an editing command to R_i. Let $P(R)$ denotes the set of packets permitted by Policy R and $D(R)$ denotes the set of packets denied by Policy R. Formally, we can define that R is safe w.r.t. policies I and T as follows:

$$Safe(R, I, T) \equiv (P(I) \cap P(T)) \subseteq P(R) \subseteq (P(I) \cup P(T))$$
$$Safe(R, I, T) \equiv (D(I) \cap D(T)) \subseteq D(R) \subseteq (D(I) \cup D(T))$$

This is semantic characterization for the safety deployment as it is based on the set of packets. It should be noted that it is very hard to verify $Safe(R, I, T)$ because it depends on the data contained in all fields of every policy rule. Unlike [22], we define a *sufficient* condition to verify whether or not a deployment is safely performed. Indeed, Theorem 1 gives syntactic characterization for the deployment safety because all rules within policies are considered as a black box. Let r_e represent the hidden default rule at the end of policy (i.e $r_e = deny-all$ or $r_e = permit-all$). For a rule r, α_r represents the set of rules that precede r in I, and β_r represents the set of rules that precede r in T.

Theorem 1. *An intermediate policy R is safe w.r.t. I and T, if the following conditions hold for all rules r_i in $I \cup T \cup \{r_e\}$: (i) If $r_i \in I \cap T$, then r_i is preceded by an improper superset of α_{r_i} or β_{r_i}. (ii) If $r_i \in I$ (or $r_i \in T$) but $r_i \notin I \cap T$ and r_i appears in R, then r_i is preceded by an improper superset of α_{r_i} (or β_{r_i}).*

Due to the space limit, the proof of Theorem 1 is given in [6].

Example 1. Consider the following initial and target policies described as a list of rules:

$$I = [a, b, c] \text{ and } T = [d, e, f]$$

Let R_1 and R_2 be intermediate policies where R_1 = [] and R_2 = $[b, e]$. Note that $\alpha_{r_e} = \{a, b, c\}$ and $\beta_{r_e} = \{d, e, f\}$. The intermediate policy R_1 may not be safe because an empty set of rules precedes r_e, which is neither an improper superset of α_{r_e} nor β_{r_e}. Also, b appears only in I, and $\alpha_b = \{a\}$. But the set of rules that precede b in R_2 is not an improper superset of α_b. Hence, R_2 may not be safe w.r.t. I and T.

Example 2. Consider the following initial and target policies:

$$I = [b, c] \text{ and } T = [d, c]$$

such that b and d permit a packet p, while c denies p. Obviously, both I and T permit p. There are two cases where p is rejected by an intermediate policy R: (i) when both b and d are not in R i.e. $R_1 = [c]$, and; (ii) when c appears at the first position in R i.e. $R_2 = [c, b]$. However, $\alpha_{r_e} = \{b, c\}$ and $\beta_{r_e} = \{d, c\}$. Therefore, b or d (or both) must precede r_e in R and case (i) cannot occur in safe deployment. Furthermore, $\alpha_c = \{b\}$ and $\beta_c = \{d\}$. Therefore, b or d (or both) must also precede c in R and case (ii) is also not possible in a safe deployment. The four possible safe running policies w.r.t. I and T are:

$$R_3 = [b, c]$$
$$R_4 = [d, c]$$
$$R_5 = [b, d, c]$$
$$R_6 = [d, b, c]$$

Theorem 2 gives syntactic characterization to verify the partial-safety. Due to the space limit, the proof of this theorem is given in [6].

Theorem 2. *An intermediate running policy R is partial-safe w.r.t. I and T, if the conditions of Theorem 1 hold for all rules r_j in $I \cup T$.*

Example 3. Let r_e be the hidden default rule at the end of policy. Consider the following policies where b and d deny a packet p, while c and e accept p:

$$I = [b, c, r_e = deny - all]$$

$$T = [d, e, r_e = deny - all]$$

Clearly, both I and T deny p. The only case where p is accepted by a running policy is when c or e appear as first rule in R. However, $\alpha_c = \{b\}$ and $\beta_e = \{d\}$. Therefore, c appears in R only when it is preceded by b. Similarly, if e appears in R, then it must be preceded by d. Therefore, this case in not possible. This also implies that if both b and d are not in R, then c and e cannot be in R. In this case, p matches r_e and it is denied. Hence, R is permission-safe. Alternatively, consider the same I and T, and the case where both b and d permit p, while c and e deny p. In this case, if both b and d are not in R, then p matches r_e and it is falsely denied. Hence, R is not denial-safe.

4 Type I Deployment

Recall that two types of security problems may arise during an unsafe deployment: (i) Rejection of legal traffic; (ii) Creation of temporary security holes. To be safe, a firewall policy deployment must avoid both types of problems. However, safe deployment is not always possible by using only the rules of I and T and Type I editing commands [22]. In Algorithm 1, called PARTIALSAFEDEPLOYMENT, we give a most-efficient algorithm that provides a *partial-safe* deployment; that is it can avoid either situation (i) or (ii) but not both. For firewall policies with permit-all semantics, the algorithm ensures that situation (i) will never occur. Similarly, for firewall policies with deny-all semantics situation (ii) is avoided.

It is worth mentioning that some types of security threats cannot be dealt by firewalls alone and additional security mechanism such as Intrusion Detection and Prevention System (IDPS) [8] may be required. If the situation (ii) temporarily arises during a deployment, an IDPS can be configured to block the illegal packets that may pass through the firewall. Therefore, in the presence of an IDPS, a firewall policy with permit-all semantics can avoid both types of problem.

It is assumed that both the initial policy I and the final one T are stored in separate arrays and the running policy R is initiallay equal to I. The algorithm is efficient, as it deploys the target policy using the minimum number of Type I editing commands. The algorithm selectively deletes all rules that are in I but not in T, in reverse order and appropriately append rules to running policy R. The algorithm begins by finding the longest prefix T' of T that is a subsequence of I. Starting from first rule in I, all rules of I – but not in T' – are then pushed to the stack and added to the hash table (lines $3 - 11$). Next, starting from the first rule in T that is not in T', each rule r is taken and placed at a correct position in R. If r is present in the hash table, this implies that r is present in R and needs to be deleted first. In this case, all rules in I that are not in T' and occur after r in I are deleted from R (lines $13 - 20$). Then r is deleted from R and appended back at the end. This ensures that r appears in R, only if it is preceded by an improper superset of α_r or β_r (see Sub-section 3.3). Thus, the condition of Theorem 2

```
 1: PARTIALSAFEDEPLOYMENT(I,T)
 2: // Find longest prefix T' of T such that T' ⊆ I
 3: j ← 1
 4: for i ← 1 to sizeOf(I) do
 5:    if I[i] = T[j] then
 6:       j ← j + 1
 7:    else
 8:       stack.push(I[i])
 9:       hash.add (I[i])
10:    end if
11: end for
12: // Place each rule of T that is not in T' at correct position
13: for t ← j to sizeOf(T) do
14:    if hash.contains(T[t]) then
15:       repeat
16:          y ← stack.pop()
17:          IssueCommand (del y)
18:          hash.delete(y)
19:       until y = T[t]
20:    end if
21:    IssueCommand (app T[t])
22: end for
23: // Delete all rules in I that are not in T
24: while NOT stack.empty do
25:    IssueCommand (del stack.pop())
26: end while
```

Algorithm 1. Partial Safe Deployment

is satisfied and the algorithm PARTIALSAFEDEPLOYMENT is *partial-safe*. Finally, the stack contains rules that are in I but not T and therefore must be deleted from R. After the deletion of these rules (lines 24-26), R becomes T. Hence, the deployment is also correct. Let $|X|$ represents the total number of rules in policy X , then $|I| + |T| - 2|T'|$ editing commands are generated by the algorithm. The algorithm takes $O(n)$ time and space,where $n = max(|I|, |T|)$.

Due to the limited set of operations and the restriction that repetition of rules is not allowed, not all deployments can be done safely using Type I languages. For instance, if $I = [a, b]$ and $T = [b, a]$ the deployement can never be safe. The restriction, that all rules must be distinct, can be overcome by using semantically equivalent rules or by breaking a rule r into sub-rules r_1 and r_2, such that $r_1 \cup r_2 = r$. Two rules r_1 and r_2 are considered semantically equivalent, if both rules match exactly the same set of packets. The union $r_1 \cup r_2$ provides a semantic equivalence to r. Regardless of firewall policy architecture, it is always possible to split a rule with a multi-value field into several rules [21]. In [6], we presented an algorithm providing a safe strategy for Type I deployment by splitting r into equivalent r_1 and r_2. Due to the space limit, this algorithm is not presented here.

5 Type II Deployment

Type II deployment allows for random modification of a running policy. Therefore, for a given set of *I* and *T*, a safe Type II deployment usually utilizes less editing commands than an equivalent Type I deployment. If *I* and *T* have identical set of rules, then *T* can be considered as a permutation of *I*. In this case, the optimal edit sequence preserves a Longest Common Subsequence LCS(*I*,*T*) of the two sequences, and the optimal edit sequence have length equal to $|I| - |LCS(I,T)|$ [12]. That is, a move command has to be generated for each rule that is not in LCS(*I*,*T*). In the general case, where *I* has some rules that are not in *T* and *T* has some rules that are not in *I*, a command has to be generated to insert/delete each such rule. Therefore, the optimal edit sequence will have a length of $|I| + |T| - c - |LCS(I,T)|$, where c is the number of rules common to both *I* and *T*.

5.1 Deployment Algorithm

In this section we present a correct, safe and most-efficient near linear running time type II deployment algorithm called EFFICIENTDEPLOYMENT (see algorithms 2 and 3). For a most-efficient deployment, we need to issue exactly one command for each rule that is in $I \cup T$ but not in one of the longest common subsequences of I and T [19]. EFFICIENTDEPLOYMENT issues exactly $|I \cup T| - |LCS(I,T)|$ commands to transform I to T. Firstly, the algorithm issues commands to selectively move rules upwards and to insert rules that are in T but not in I. An array T_2 is maintained that facilitates in the calculation of positional parameters for *ins*, *del* and *mov* operations. After commands are issued for all the rules to be moved upwards and inserted, T_2 becomes the running policy.

EFFICIENTDEPLOYMENT maintains two variables i and t that point to the rules in I and T respectively that are currently under consideration to be appended to T_2. The rules still to be moved upwards and inserted are also appended to an array χ and later on commands are generated for these rules, as described below.

The algorithm starts by traversing T from $T[t = 1]$. If $T[t]$ is neither in T_2 nor in $LCS(I,T)$, then it is appended to T_2 and χ (lines 9-10). While if $T[t]$ is in $LCS(I,T)$ but not in T_2, then I is traversed by incrementing i until $I[i] = T[t]$ and any rule $I[i]$ that is not in T_2 is appended to T_2 and pushed to the stack (lines 12-23). Then, $T[t]$ is appended to T_2 and the variable t is incremented. Next, a command is issued for each rule r in χ and then r is deleted from χ (lines 26-37). If r is not in I, then an $insert$ command is issued and the variable M is incremented, otherwise a $move$ command is issued that places r closer to the beginning of R. Each rule to be moved upwards that appear after r in I but before r in R causes the r to be shifted down one position in R. The number of such rules, N, is calculated by using binary search technique in the function $Count$ and stored in the array C_1. Similarly the number of rules, M, inserted above r causes r to be shifted down M position. Therefore, the current position of r in R is the sum of its initial position in I, M, and N. The value of N is calculated in $O(log|I|)$ steps.

```
1: EFFICIENTDEPLOYMENT(I,T)
2: i ← 1, L ← LCS(I,T)
3: C1 ← empty array of size sizeOf(I)
4: C2 ← empty array of size sizeOf(T)
5: T2 ← empty_array, χ ← empty_array
6: for t ← 1 to sizeOf(T) + 1 do
7:    if T[t] ∉ T2 then
8:       if T[t] ∉ L then
9:          χ.append(T[t])
10:         T2.append(T[t])
11:      else
12:         while I[i] ≠ T[t] do
13:            if I[i] ∉ T2 then
14:               T2.append(I[i])
15:               if I[i] ∈ T then
16:                  stack.push(I[i], count(indexOf(I[i], T), C2)
17:               else
18:                  stack.push(I[i], 0)
19:                  V ← V + 1
20:               end if
21:            end if
22:            i ← i + 1
23:         end while
24:         T2.append(I[i])
25:         i ← i + 1
26:         if sizeOf(χ) > 0 then
27:            j ← 1
28:            while sizeOf(χ) > 0 do
29:               if χ[j] ∉ I then
30:                  issueCommand(insert indexOf(χ[j], T2), χ[j])
31:                  M ← M + 1
32:               else
33:                  issueCommand(move indexOf(χ[j], I) +
                     count(indexOf(χ[j], I), C1) + M, indexOf(χ[j], T2))
34:               end if
35:               χ.delete(χ[j])
36:               j ← j + 1
37:            end while
38:         end if
39:      end if
40:   end if
41: end for
42: //MoveDowns and Delete
43: while NOTstack.empty do
44:    [r, U] ← stack.pop()
45:    if r ∉ T then
46:       issueCommand(delete indexOf(r, T2))
47:       V ← V − 1
48:    else
49:       issueCommand(move indexOf(r, T2), indexOf(r, T) + U + V + 1)
50:    end if
51: end while
```

Algorithm 2. EFFICIENTDEPLOYMENT algorithm

```
1: Count(nodePos, arr) {
2: start ← 1, last ← sizeOf(arr)
3: adjust ← 0
4: mid ← Trunc((start + last)/2)
5: while mid ≠ nodePos AND start ≤ last do
6:    if mid > nodePos then
7:       adjust ← adjust + arr[mid]
8:       last ← mid − 1
9:    else
10:      start ← mid + 1
11:      arr[mid] ← arr[mid] + 1
12:   end if
13:   mid ← Trunc((start + last)/2)
14: end while
15: arr[mid] ← arr[mid] + 1
16: return(adjust + arr[mid] − 1) }
```

Algorithm 3. Count function

After all the rules in χ are processed, the traversal is resumed at T[t] and the steps described in the previous paragraph are repeated until $T[|T|+1]$ is reached. We assume that $I[|I|+1] = T[|T|+1] = r_e$, where r_e is the default rule at the end of each policy. After traversal is finished T_2 becomes the running policy.

Finally, a command is issued for each rule s in stack (lines 43-51). If s is not in T, then a $delete$ command is issued, otherwise a $move$ command is issued. The current position of s is its index in T_2. The final position for $move$ command is the sum of index of s in T, the rules still need to be deleted (V), and the number of rules (U) that appear before s in I but after s in T. The value of U is determined in $O(log|T|)$ steps in the function $Count$ and stored in array C_2. The total running time for issuing all commands is $O(n + x\ log\ n)$, where x is the number of rules to be moved and $n = max(|I|, |T|)$. The running time can be further improved if a balance tree such as AVL tree is used to compute positional parameters leading to a running time complexity of $O(n + x\ log\ x)$.

5.2 Safety and Correctness of EFFICIENTDEPLOYMENT

Recall that α_x represents the set of rules that precede a rule x in I, and β_x represents the set of rules that precede x in T. EFFICIENTDEPLOYMENT starts by traversing T from the beginning and if a rule $r = T[t]$ is encountered, which is not in T_2, it is immediately appended to T_2.

If a rule $\Sigma = T[t]$ is encountered that is in $LCS(I, T)$, then i is incremented until $I[i] = \Sigma$ and any rule $s = I[i]$ that is not in T_2 is appended to T_2. If s is not in T_2, this means that either s does not appear in T or it appears after Σ in T and therefore must be moved downwards. Similarly r is not in $LCS(I, T)$ and it appears before Σ in T, this implies that either r is not present in I or it appears after Σ in I and therefore must be moved upwards.

As all rules that appear before r in T are appended to T_2 before r, therefore r is preceded by an improper superset of β_r in T_2. Also, all rules that appear before Σ in I and/or T precede Σ in T_2. In other words, Σ is preceded by $\alpha_\Sigma \cup \beta_\Sigma$ in T_2. Similarly,

some rules in T and all rules in I that precede s are appended to T_2 before s, therefore s is preceded by an improper superset of α_s in T_2.

Initially I is the running policy. The algorithm starts by issuing commands for rules to be inserted and move upwards. The final position of a move (or insert) operation for r is $indexOf(r, T_2)$. Therefore, after the move operation r is preceded by an improper superset of β_r. Since r is moved upwards, so it still precedes the rules that appear below it in I. Therefore according to Theorem 1 (see Sub-section 3.3), R remains safe w.r.t. I and T.

After commands are issued for all the rules to be moved upwards and inserted, T_2 becomes the running policy and s is preceded by an improper superset of α_s. However, some rules in β_s may still appear below s in R. The correct position of s is calculated as described in previous section, which causes s to be preceded by an improper superset of $\alpha_s \cup \beta_s$. Therefore according to theorem 1, R remains safe w.r.t. I and T. When all the rules, to be moved down and deleted, above s are processed then s is preceded by exactly β_s. Thus, after commands are issued for all rules to be moved downwards and deleted, each rule x is preceded by exactly β_x. In other words, each rule in R is preceded by exactly the same set of rules that precedes it in T; this implies that I is converted to T and hence the deployment is safe and correct.

6 Performance Evaluation

To evaluate the performance of EFFICIENTDEPLOYMENT, we try to follow the same set of test cases as in [22]. We use four firewall policies with 2000, 5000, 10000, and 25000 rules. For each policy, we perform five different tests. In a most-efficient deployment test 1, test 2, test 3,test 4 and test 5 requires 10, 500, 1000, 60%, and 90% of commands respectively to convert initial policy to the target policy. Note that these percentages are taken from the initial policy. The algorithm is implemented in C++, and all tests are performed on *Dell Precision 370* with Intel Pentium IV 2.0 Ghz processor and 1 GB of RAM[1]. The results of each test on policies 1-4 are given in Table 1. The time taken by EFFICIENTDEPLOYMENT is specified in the column ED, while the column SI specifies the total time taken by *diff* and SANITIZEIT algorithm given in [22] for computing a safe deployment. All times are shown in seconds.

It is clear that EFFICIENTDEPLOYMENT takes a fraction of second to calculate safe and most-efficient deployment for policies as large as Policy 4. Also, EFFICIENTDEPLOYMENT generates a most-efficient and safe deployment much faster than the SanitizeIt algorithm. However, as no details are given about nature of changes in [22], it might not be appropriate to directly draw conclusion for tests 2-5. For example, consider Test 5 on Policy 4, 90% edit distance means 22500 commands need to be issued to turn *I* to *T*. If 22,500 insert commands are required that means *T* has 47,500 rules, while if 22,500 delete commands are required then *T* has only 2500 rules. Therefore, reliable comparison can only be done if size of *I* and *T* used in [1] is known, so that policies of same size could be used for testing EFFICIENTDEPLOYMENT. However, Test 1 involves only 10 changes and it can be used to compare the two algorithms, as shown in figure 2.

[1] An executable program is available at
`http://webloria.loria.fr/~imine/program.zip`

Table 1. Performance tests

Tests	Policy 1 (2,000)		Policy 2 (5,000)		Policy 3 (10,000)		Policy 4 (25,000)	
	ED	SI	ED	SI	ED	SI	ED	SI
Test 1	.00635	.01200	.01542	.02300	.03023	.04400	.07567	.24200
Test 2	.00662	.01200	.01622	.02800	.03111	.04900	.07580	.28300
Test 3	.00697	.03800	.01641	.04900	.03154	.07000	.07592	.32500
Test 4	.00703	.04000	.01757	.20500	.03518	1.3820	.08943	12.582
Test 5	.00732	.07000	.01859	.38700	.03722	4.3920	.09421	26.983

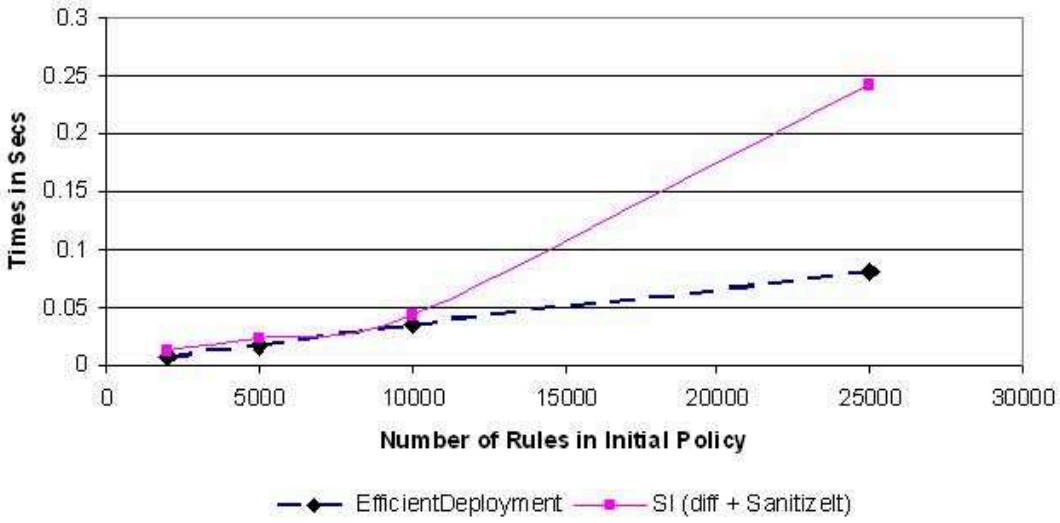

Fig. 2. Comparison of EFFICIENTDEPLOYMENT and SANITIZEIT for Test 1

From the curve illustrated in Figure 2, it can be concluded that EFFICIENTDEPLOYMENT is more efficient than SANITIZEIT and the running time is close to linear. Furthermore, SANITIZEIT appears to have a polynomial running time. This effect is more notable in case of test 5 and Policy 4, where SI takes almost 27 secs to compute a deployment sequence.

7 Conclusion

Firewall policy deployment safety is a new and area of research. In this paper, we have presented a formalization for deployment safety and used this formalization as a basis to provide safe and efficient algorithms for both Type I and Type II languages. We have proposed for type I policy editing languages a correct algorithm that is efficient and partial-safe. For Type II policy editing languages, we have presented an approximately linear, most-efficient and safe algorithm. Our experimental results showed that this algorithm does not add any overhead and it is practical even for very large policies. In future work, we plan to investigate the deployment problem in two kinds of firewalls: stateful firewalls and distributed firewalls.

References

1. Cisco Security Manager,
`http://www.cisco.com/en/US/products/ps6498/index.html`

2. Entrasys Matrix X Core Router, http://www.entrasys.com/products/routing/x/
3. F-Secure. Malware information pages: Slammer, http://www.f-secure.com/v-descs/mssqlm.shtml
4. F-Secure. Malware information pages: Worm:w32/downadup.al, http://www.f-secure.com/v-descs/worm_w32_downadup_al.shtml
5. Juniper Network and Security Manager, http://www.juniper.net/us/en/local/pdf/datasheets/1100018-en.pdf
6. Ahmed, Z., Imine, A., Rusinowitch, M.: Safe and Efficient Strategies for Updating Firewall Policies. Research Report RR-6940, INRIA (2009), http://webloria.loria.fr/~imine/rep2009.pdf
7. Al-Shaer, E., Hamed, H.: Modeling and Management of Firewall Policies. IEEE Transactions on Network and Service Management 1(1), 2–10 (2004)
8. Anwar, M., Zafar, M., Ahmed, Z.: A Proposed Preventive Information Security System. In: International Conference on Electrical Engineering, ICEE '07, pp. 1–6 (2007)
9. Baboescu, F., Varghese, G.: Fast and Scalable Conflict Detection for Packet Classifiers. In: ICNP, pp. 270–279 (2002)
10. Bartal, Y., Mayer, A.J., Nissim, K., Wool, A.: Firmato: A Novel Firewall Management Toolkit. In: IEEE Symposium on Security and Privacy, pp. 17–31 (1999)
11. Cobb, S.: ICSA Firewall Policy Guide v2.0. Technical report. NCSA Security White Paper Series (1997)
12. Cormode, G., Muthukrishnan, S., Sahinalp, S.C.: Permutation Editing and Matching via Embeddings. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 481–492. Springer, Heidelberg (2001)
13. Englund, M.: Securing systems with host-based firewalls. In: Sun BluePrints Online (September 2001)
14. Fu, Z., Wu, S.F., Huang, H., Loh, K., Gong, F., Baldine, I., Xu, C.: IPSec/VPN Security Policy: Correctness, Conflict Detection, and Resolution. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, pp. 39–56. Springer, Heidelberg (2001)
15. Gouda, M.G., Liu, A.X.: Firewall Design: Consistency, Completeness, and Compactness. In: ICDCS, pp. 320–327 (2004)
16. Hamed, H., Al-Shaer, E.: Dynamic rule-ordering optimization for high-speed firewall filtering. In: ASIACCS, pp. 332–342 (2006)
17. Karen, S., Paul, H.: Guidelines on Firewalls and Firewall Policy. NIST Recommendations, SP 800-41 (July 2008)
18. Liu, A.X.: Change-impact analysis of firewall policies. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 155–170. Springer, Heidelberg (2007)
19. Myers, E.W.: An o(nd) difference algorithm and its variations. Algorithmica 1(2), 251–266 (1986)
20. Qian, J.: 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, Deventer, The Netherlands, p. 4. Kluwer, B.V. (2001)
21. Qiu, L., Varghese, G., Suri, S.: Fast firewall implementations for software and hardware-based routers. In: International Conference on Network Protocols, pp. 155–170 (2001)
22. Zhang, C.C., Winslett, M., Gunter, C.A.: On the Safety and Efficiency of Firewall Policy Deployment. In: SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy, Washington, DC, USA, pp. 33–50. IEEE Computer Society, Los Alamitos (2007)