Safe and Efficient Strategies for Updating Firewall Policies

Zeeshan Ahmed, Abdessamad Imine, Michaël Rusinowitch

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Safe and Efficient Strategies for Updating Firewall Policies

Zeeshan Ahmed and Abdessamad Imine and Michaël Rusinowitch

N° 6940

May 2009

Thème SYM

ISSN 0249-6399

ISSN 0249-6399 ISRN INRIA/RR--6940--FR+ENG

Safe and Efficient Strategies for Updating Firewall Policies

Zeeshan Ahmed* and Abdessamad Imine† and Michaël Rusinowitch‡

Thème SYM — Systèmes symboliques
Projet CASSIS

Rapport de recherche n° 6940 — May 2009 — 19 pages

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. Much research has already addressed policy specification, conflict detection, and optimization but very little research is devoted to firewall policy deployment. Only recently, some researchers have proposed deployment strategies for two important classes of policy editing languages. In this report, we show that these strategies have serious flaws leading to security breaches. Then we provide correct, efficient and safe algorithms for both classes of languages. Our experimental results show that these algorithms are very fast and can be used safely even for deploying very large policies.

Key-words: Firewall Policy Management, Firewalls, Network Security.

* INRIA Nancy Grand Est, UMR 7503 (ahmedzee@loria.fr).
† INRIA Nancy Grand Est & Univ. Nancy 2, UMR 7503 (imine@loria.fr).
‡ INRIA Nancy Grand Est, UMR 7503 (rusi@loria.fr).

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Stratégies Sûres et Efficaces pour la Mise-à-jour des Politiques de Pare-Feu

Résumé : Actuellement, les politiques de pare-feu (en anglais firewall) peuvent contenir de milliers de règles et ce à cause de la taille énorme et la structure complexe des réseaux modernes. De ce fait, ces politiques nécessitent des outils automatiques fournissant un environnement convivial pour spécifier, configurer et déployer en sûreté une politique cible. Beaucoup de travaux de recherche ont traité de la spécification des politiques, la détection des conflits et le problème d'optimisation, mais très peu de travaux se sont intéressés au déploiement de politiques. Ce n'est que récemment, certains chercheurs ont proposé des stratégies de déploiement pour les deux importantes catégories d'édition de politiques. Dans ce rapport, nous montrons que ces stratégies sont erronées et pourraient mener à des failles de sécurité. Ensuite, nous fournissons deux algorithmes corrects, efficaces et sûrs pour les classes d'édition de politiques. Nos résultats expérimentaux montrent que ces algorithmes sont trés rapides et peuvent être utilisés en toute sûreté, même pour le déploiement de politiques dont la taille est très importante.

Mots-clés : Gestion des Politiques de Pare-Feu, Pare-Feu, Sécurité des Réseaux.

Contents

1 Introduction

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 [18]. 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 [24]. Due to intervening nature of firewall rules, correct configuration and *deployment* of such large policies is a very difficult and error-prone. A policy deployment is the process by which the running policy is replaced by the target policy.

To ease the burden on firewall administrators, many firewall management tools such as Cisco Security Manager [2], Juniper Networks' Netscreen Security Manager [6], and Check Point SmartCenter [1] have been developed. These tools provide a user-friendly environment to specify, configure and automatically deploy a target policy. Much research has addressed policy specification [10,7,16], conflict detection [19, 15, 9], and optimization [20, 17]. However, very little research has been done on firewall policy deployment.

A firewall policy deployment should have following characteristics [24]: correctness, confidentiality, safety, and speed.

Correctness: A deployment is correct if it successfully implements the target policy on the firewall. After a correct deployment the target policy becomes the running policy. Correctness is an essential requirement for any deployment.

Confidentiality: Confidentiality refers to securing the communication between a management tool and a firewall. Due to the sensitive nature of information transmitted during a deployment, the communication between management tool and firewall should be confidential. Confidentiality can be achieved by using encrypted communication protocols such as SSH [23] and SSL [22].

Safety: 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.

Speed: A deployment should be done in the shortest time, so that the desired state of affairs is achieved as quickly as possible. A deployment algorithm should have a good running time, so that it is applicable even for large policies. Also, the algorithm 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 [24].

Different firewalls support different policy editing commands. The set of policy editing commands that a firewall supports is called its policy editing language. In [24], the authors classify policy editing languages into two representative classes, Type I and Type II, and provide deployment algorithms for both types of languages. To the best of our knowledge, it is the first work that addresses deployment safety and efficiency.

In this paper, we analyse the algorithms provided in [24] and show that these algorithms have serious flaws. We present an improved safety formalization that can be used as a basis for formulating safe deployment strategies (Section 3). We provide two linear algorithms for Type I deployment

(Section 4). The first algorithm is most-efficient and it ensures that either no legal traffic is rejected or no illegal traffic is permitted. While the second algorithm is safe but it generates some extra rules to ensure deployment safety. We also give an approximatively 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 Firewall Background

A firewall is a perimeter security device that filters packets that traverse across the boundaries of a secured network. The filtering decision is based on a firewall policy defined by 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. Majority of firewalls filter traffic according to first-match semantics, that is when a packet p arrives, it is compared against the rules top-down until a matching rule is found and the process is repeated for the following packet. All policies have 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. Majority 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.* A rule is 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 [24]. However, the following five fields are most commonly used: protocol type, source IP address, source port, destination IP address and destination port [12].

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.

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. As discussed in the Introduction, a deployment must be correct and should satisfy the following three characteristics: confidentiality, safety, and speed.

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. Typically, a firewall uses a subset of the following editing commands [24]:

(*app r*)	appends rule *r* at the end of *R*
(*del r*)	deletes *r* from *R*
(*del i*)	deletes the rule at position *i* from *R*
(*ins i r*)	inserts *r* at position *i*
(*mov i j*)	moves the ith rule to the jth position in *R*

Policy editing languages can be classified into two representative classes [24]: Type I and Type II.

Type I Editing. Type I editing 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 [24], therefore it is complete. Most older firewalls and some recent firewalls,such as FWSM 2.x [2] and JUNOSe 7.x [6], only support Type I editing.

Type II Editing. Type II languages allow 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 [24], therefore it is both complete and safe. 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 [14] and Enterasys Matrix X [3].

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, the most-efficient Type I deployment uses the minimum number of append and delete commands, similarly a most-efficient Type II deployment uses the minimum number of 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. Therefore, 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 may permit malicious traffic to pass through the firewall that 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 particulary 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 [5] and Slammer [4], 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 serious work on deployment safety is presented in [24], and a safe deployment formalization is presented. The formalization defines a safe deployment as follow; 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. to 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 the initial and the target policies. A policy is *safe* iff it is both denial-safe and permission-safe. In the rest of 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 $I = R_0, R_1, ..., R_{i-1}, R_n = T$ with R_{i+1} is derived by applying an editing command to R_i [24].

Let P(x) denotes the set of packets permitted by Policy x and D(x) denotes the set of packets denied by Policy x. Then, mathematically we can define that Policy x is safe w.r.t. policy I and policy T as follows:

$$Safe(x,I,T) \longleftrightarrow (P(I) \cap P(T)) \subseteq P(x) \subseteq (P(I) \cup P(T))$$
$$Safe(x,I,T) \longleftrightarrow (D(I) \cap D(T)) \subseteq D(x) \subseteq (D(I) \cup D(T))$$

Definition 3.1 ***Partial-Safe Deployment.*** *A deployment is partial-safe if it is either permission-safe or denial-safe but not both w.r.t. initial and final policies.*

Theorem 1 *A deployment that satisfies the following two conditions is safe:*
(a) Every running policy R *only contains rules from* I *and* T*, and every rule common to both* I *and* T *is present in R. (b) Let r_1 and r_2 be two rules in* I *or* T *or both, such that r_2 always appears after r_1, then r_2 appears after r_1 in R.*

Proof. The first condition states that a rule can appear in R only if it belongs to I or T, and the rules common to both I and T are always present in R. The second condition states that whenever r_1 and r_2 simultaneously appear in R, the mutual order of r_1 and r_2 in R is according to either I or T.

Let r be a rule that it is present in both I and T, and p_1 be a packet that hits r in both policies. Clearly, no rule r' exists before r in I or T that matches p_1. According to condition(a), r is always present in R. According to condition(b), r' cannot occur before r. Therefore p_1 will always hit r in R. If p_1 is permitted by r, then p_1 is always permitted during the deployment and vice versa. Let p_2 be a packet that does not match any rule in I and T. According to condition (a), only rules from I and T can appear in R. Therefore, p_2 will never hit a rule in R. Hence, R is safe w.r.t. I and T, and the deployment is safe. ■

Theorem 2 *A deployment which only utilizes the rules of* I *and* T*, and satisfies the condition(b) of Theorem 1 is partial-safe.*

Proof. We first consider a firewall policy with a deny-all default rule at the end, i.e. all packets that do not match to any rule are rejected by default. Let us consider a rule r_1, which denies a packet p_1, is present in both I and T. Let us consider a rule r_2, which accepts p_1, is present in either I or T or both. Assume that p_1 hits r_1 in both I and T. This implies that if r_2 appears in I or T, it appears after r_1. Now, if R contains only rules from I and T and the condition(b) of Theorem 1 is satisfied then r_2 cannot appear before r_1 in R during the deployment. Therefore, if r_1 is present in R, then p_1 hits r_1 and it is denied. On the other hand, if r_1 is not present in R, then r_2 is also not present in R and p_1 does not match any rule in R and it is denied by the default rule. Thus, any packet that is denied by I and T is also denied by R and hence the deployment is safe.

Following the same reasoning, it can be proved that a firewall with a permit-all default rule is *denial-safe*. ■

4 Type I Deployment

4.1 Problems with Previous Algorithms

Two algorithms are presented in [24] for Type I deployment. It is assumed that both the initial policy I and the final one T are stored in separate arrays. The first deployment algorithm follows a simple scheme but it is not most-efficient. The algorithm is comprised of two phases. In phase 1, starting from the first rule in T, the algorithm appends each rule in T to the end of R . If the rule is already present in R, then it is first deleted the appended back at the end of R. In phase 2, starting from the last rule in I, every rule in I that is not in T is deleted from R. It is claimed in [24] that the deployment is safe except for intervals when rules that are present in both I and T are deleted and appended back. However, we have found that this algorithm is not safe for other situations as well.

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	a. deny tcp 10.1.1.0/24 any
c. permit tcp 10.1.0.0/16 any	c. permit tcp 10.1.0.0/16 any
d. permit tcp 192.168.2.0/24 any	d. permit tcp 192.168.2.0/24

(a) Initial and Target policies.

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

(b) Running policy.

Figure 1: Example of Policy Deployment

For example, consider the initial and target policies given in Figure 1.(a). 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 apply the algorithm,then after two steps we get the resulting running policy R as shown in Figure 1.(b).

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. Clearly, the condition(b) of Theorem 1 is violated and R is not safe w.r.t. I and T. Hence, the deployment by the given algorithm is not safe even when common rules are appended back to R.

The second algorithm proposed for Type I editing in [24] is called SCANNINGDEPLOYMENT . It is claimed that this algorithm is a most efficient Type I algorithm. However, it can be easily shown

that the algorithm is not correct even in simple cases. When this algorithm is used for deployment, the longest prefix of T that is a subsequence of I is deleted from the final running policy i.e. T is not deployed correctly. For example, consider the by applying SCANNINGDEPLOYMENT on I and T with the same rules as in Figure 1.(a) we have:

I	T
a	b
b	c
c	d

$\longrightarrow$ SCANNINGDEPLOYMENT $\longrightarrow$

R
d

Clearly, the deployment is not correct. Also, the rules b and c that are present in both I and T are missing from R, and the condition(a) of Theorem 1 is not satisfied. Consider a packet p_2 with source address 192.168.1.1. Clearly, p_2 is permitted by both I and T, but R denies it. This a serious problem because the final running policy will deny legal traffic until a future deployment is done correctly.

SCANNINGDEPLOYMENT comprises of two phases. In the first phase, a stack and a hash table is used to store all the rules that are candidates of being deleted in the next phase. At the end of Phase I, the hash table should contain only hashes of rules that are not present in T. In the second phase, the algorithm pops a rule from stack and check if it is in hash table. If it is present in hash table that means it is in I but not T and should be deleted. However, the algorithm is not formulated correctly and at the end of phase I, hash table contain entries for rules in T. As a consequence, these rules are deleted from the final running policy and T is not deployed correctly.

4.2 Our Solutions for Type I Deployment

Algorithm 1 presents the corrected version of SCANNINGDEPLOYMENT, in which the rules in T are deleted from the hash table at the end of phase I. These corrections are shown in bold (lines 19-23). Recall that two types of security problems may arise during an unsafe deployment: (a) Rejection of legal traffic. (b) 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 [24]. In Algorithm 2, called PARTIALSAFEDEPLOYMENT, we give a most-efficient algorithm that provides a *partial-safe* deployment, that is it can avoid either situation (a) or (b) but not both. For firewall policies with permit-all semantics, the algorithm ensures that situation (a) will never occur. Similarly, for firewall policies with deny-all semantics situation (b) 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 (b) 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.

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 R. The algorithm begins by finding the longest prefix T' of T that is a subsequence of I and all rules of I that are not in T', starting from first rule in I, are pushed

to stack and added to hash table (lines 3-9). Next, starting from the first rule in T that is not in T', each rule r is taken and placed at correct position in R. If r is present in hash table, this implies that r is present in R and needs to be deleted first. If In this case, all rules in I that are not in T' and occurs after r in I are deleted from R (lines 12-17). Then r is deleted from R and appended back at the end. This ensures that all rules, which do not appear before r in I or T, never appear before r in R. Thus, the condition of Theorem 2 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 20-21), 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 $n = max(|I|,|T|)$.

```
1: CORRECTSCANNINGDEPLOYMENT(I,T)
2: // Phase I:
3: i ← 1
4: for t ← 1 to sizeOf(T) do
5:     while i ≤ sizeOf(I) and I[i] ≠ T[t] do
6:         stack.push (I[i])
7:         hash.add (I[i])
8:         i ← i+1
9:     end while
10:    if i > sizeOf(I) then
11:        if hash.contains(T[t]) then
12:            hash.remove (T[t])
13:            IssueCommand (del T[t])
14:        end if
15:        IssueCommand (app T[t])
16:    end if
17: end for
18: // Phase II:
19: for t ← 1 to sizeOf(T) do
20:    if hash.contains(T[t]) then
21:        hash.remove (T[t])
22:    end if
23: end for
24: if sizeOf(I) ≥ sizeof(T) and sizeOf(T)> 0 then
25:    i ← i+1
26: end if
27: for j ← sizeOf(I) downto i do
28:    IssueCommand (del I[j])
29: end for
30: while NOT stack.empty do
31:    r ← stack.pop()
32:    if hash.contains(r) then
33:        IssueCommand (del r)
34:    end if
35: end while
```

Algorithm 1: Corrected version for Scanning Deployment proposed by [24].

```
1: PARTAILSAFEDEPLOYMENT(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[i])
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 2: Partial Safe Deployment.

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 [24]. 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 the 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 Algorithm 3, we provide a safe strategy for Type I deployment by splitting r into r_1 and r_2. An obvious scheme to split r into semantically equivalent rules is to to find the first field f in r that can be split into fields f_1 and f_2, then generate two rules r_1 and r_2 such that:

$$r_1.f = f_1,\ r_2.f = f_2, \text{ and } r_1.f \cup r_2.f = r.f$$

All other fields of r_1 and r_2 remains identical to the corresponding fields of r.

The algorithm begins by finding the longest prefix T' of T that is a subsequence of I. Then, for each rule r in T that needs to be deleted from R and appended back later on, two rules r_1 and r_2 are appended first to R, then r is deleted from R and appended back at the end of R. Finally, r_1, r_2 and all rules in I that are not in T are deleted from R. The algorithm follows the same logic as in PARTIALSAFEDEPLOYMENT (see Algorithm 2), except that before deleting a common rule from R, semantically equivalent rules are appended to R. Thus, both conditions of Theorem 1 are satisfied, and the deployment is safe. The algorithm has a linear running time, but it cannot be considered most-efficient because it generates two extra rules for each rule r common to both I and T such that r is not in T'. However, the deployment is safe at each step, and at the end T is deployed correctly. Let c_1 represents the number of rules in I but not T, and c_2 represents the number of rules in T but not not I, then the total number of commands generated by algorithm is equal to $6|I| + c_2 - 6|T'| - 5c_1$.

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 $|I| - |LCS(I,T)|$ [13]. 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 Problems with Previous Algorithms

Two algorithms are proposed in [24] for type II deployment. The first algorithm is a greedy two-phase algorithm called TWOPHASEDEPLOYMENT, while the second algorithm is a most-efficient algorithm called SANITIZEIT. It is claimed in [24] that TWOPHASEDEPLOYMENT is correct and safe. However, it can be shown that it is not correct even for very simple deployments. Consider the application of TWOPHASEDEPLOYMENT to I and T given in Figure 1.(a):

I	**T**
a	c
b	b
c	a

$\longrightarrow$ TWOPHASEDEPLOYMENT $\longrightarrow$

R
a
c
b

```
1: SAFET1DEPLOYMENT(I,T)
2: // U holds the original rules that are broken and need to be appended back later on
3: // V holds the rules r1 and r2 that are to be deleted later on
4: i ← 1
5: j ← 1
6: isStarted ← FALSE
7: for i ← 1 to sizeOf(T) do
8:     while i ≤ sizeOf(I) and I[i] ≠ T[t] do
9:         stack.push(I[i])
10:        hash.add(I[i])
11:        i ← i+1
12:    end while
13:    if i ≤ sizeOf(I) then
14:        i ← i+1
15:    else
16:        if NOT isStarted then
17:            if hash.contains(T[t]) then
18:                isStarted ← TRUE
19:                r ← T[t]
20:                U.append(r)
21:                // Split rule r into sub-rules r1 and r2
22:                V.append(r1)
23:                V.append(r2)
24:                IssueCommand (app r1)
25:                IssueCommand (app r2)
26:            else
27:                IssueCommand (app r)
28:            end if
29:        else
30:            r ← T[t]
31:            U.append(r)
32:            // Split rule r into sub-rules r1 and r2
33:            V.append(r1)
34:            V.append(r2)
35:            IssueCommand (app r1)
36:            IssueCommand (app r2)
37:        end if
38:    end if
39: end for
40: // Phase II
41: while NOT stack.empty do
42:    r ← stack.pop()
43:    if hash.contains(r) then
44:        IssueCommand (del r)
45:    end if
46: end while
47: for i ← 1 to sizeOf(U) do
48:    IssueCommand(app U[i])
49: end for
50: for j ← sizeOf(V) downTo 1 do
51:    IssueCommand (del V[j])
52: end for
```

Algorithm 3: Safe Type I Deployment.

Obviously, the final running policy is different from *T*, and hence the deployment is not correct. This a serious problem because the final running policy will remain unsafe until a future deployment is done correctly. SANITIZEIT is a most-efficient and safe Type II algorithm that depends upon the optimal edit sequence D computed by *diff* algorithm [24]. The deployment sequence D is given as an input to SANITIZEIT. An important question is how to compute the set of rules Δ from D. The

diff algorithm generates a sequence of *ins* and *del* commands, and the rule's position specified in an operation depends upon the sequence of previous operations. Therefore, it is not clear that how to directly determine the rule concerned by a particular operation. This computation might change the overall complexity of the algorithm. For example, consider the following running policy and the set of insert and delete operations performed on it.

R	del 3	ins 1,c	ins 3,d	del 4
a	a	c	c	c
b	b	a	a	a
c		b	d	d
			b	

The sequence of rules concerned by these operations is (c,d,b). But to actually determine the rule concerned by a particular operation, one must know the current state of R.

5.2 Our Solution for Type II Deployment

The above problems motivate us to provide a correct, safe and most-efficient O(n log n) algorithm, called EFFICIENTDEPLOYMENT (see Algorithms 4 and 5). The main phases in EFFICIENTDEPLOYMENT can be summarized as follow:

1. Construction of Intermediate Target policy T_2
2. Perform Move up and Insert operations
3. Perform Move Down and Delete operations

The algorithm performs a most-efficient deployment by generating exactly one editing command for each rule in I and T that is not in the longest common sequence of I and T LCS(I,T), represented by L in the Algorithm. Computation of LCS is a classical problem that can be solved in $O(nlogn)$ time [11]. Let r be a rule that is present in both I and T but not L, and r' be the first rule of L that appears after r in T, then whether r is to be moved up or moved down can be determined as follows:

if ($IndexOf(r',I) < IndexOf(r,I)$) **then** (Move Up r) **else** (Move Down r) **end if**

In the first phase (lines 3-27), the algorithm constructs an intermediate target policy T_2 such that every rule to be moved up or inserted in R is placed correctly relative to the elements of L, while the rules to be moved down or deleted from R preserve their relative position in I w.r.t elements of L. As the order of rules to be deleted or moved down is preserved between the initial policy I and T_2, therefore LCS(I,T_2) consists of all the rules in L, the rules to be moved down and the rules that are in I but not in T. Similarly, LCS(T_2,T) consists of all the rules in L, the rules to be moved up and the rules that are in T but not in I. $LCS(I,T_2)$ is represented by L_2, while $LCS(T_2,T)$ is represented by L_3 in the algorithm. The role of T_2 is to simplify the calculation of rules' positions as the commands are issued over the running policy.

The construction of T_2 is very simple, and is done as follows. A pointer *cpI* is maintained in I, that represent points to current rule in I that is a candidate to be appended to T_2. Starting from first rule in T, any rules to be moved up or inserted are appended to T_2. If a rule r' that is in L is found, then I is traversed from current pointer until r' is found and all rules that are not in T_2 are appended to T_2. Also, all rules in I that are to be moved down or deleted are pushed into a stack. This phase takes O(n) time and space, where $n \leq |I| + |T|$.

```
1: EFFICIENTDEPLOYMENT(I,T)
2: // Phase I: Construction of T2
3: S1 ← empty stack , L ← LCS(I,T)
4: r' ← L.first , cpI ← 1
5: for t ← 1 to sizeOf(T) do
6:     if T[t] = r' then
7:         while I[cpI] ≠ T[t] do
8:             if I[cpI] ∉ T2 then
9:                 T2.append(I[cpI]) , L2.append(I[cpI])
10:            end if
11:            if I[cpI] ∉ T then
12:                S1.push(I[cpI])
13:            end if
14:            cpI ← cpI + 1
15:        end while
16:        T2.append(r') , L2.append(r')
17:        r' ← L.next
18:    else
19:        if T[t] ∉ I then
20:            T2.append(T[t])
21:        else
22:            if indexOf(T[t],I) > indexOf(r',I) AND indexOf(r',T) ≤ sizeOf(I) then
23:                T2.append(T[t])
24:            else
25:                S1.push(T[t])
26:            end if
27:        end if
28:    end if
29:    for i ← cpI to sizeOf(I) do
30:        if I[i] ∉ T2 then
31:            S1.push(I[i])
32:        end if
33:    end for
34: end for
35: // Phase II: Move up and Insert operations
36: r' ← L2.first , inserts ← 0 , cursor ← indexOf(r',I)
37: for t ← 1 to sizeOf(T2) do
38:    if T2 = r' then
39:        indexOf(r',R) ← indexOf(r',I) + inserts + Moveups(r',FALSE)
40:        L3.append(r')
41:        if NOT L2.end then
42:            r' ← L2.next
43:            curosr ← indexOf(r',I) + inserts + Moveups(r',FALSE)
44:        else
45:            cursor ← cursor + 1
46:        end if
47:    else
48:        if T2[t] ∉ I then
49:            IssueCommand (ins cursor,T2[t])
50:            inserts ← inserts + 1 , cursor ← cursor + 1, L3.append(T2[t])
51:        else
52:            if indexOf(T2[t],I) > indexOf(r',I) then
53:                IssueCommand (mov indexOf(T2[t],I) + MoveUps(T2[t],TRUE) + inserts, cursor)
54:                cursor ← cursor + 1 , L3.append(T2[t]
55:            end if
56:        end if
57:    end if
58: end for
59: //Phase III: Move Down and Delete operations
60: while NOT S1.empty do
61:    r ← S1.pop()
62:    if r ∉ T then
63:        r.pos ← indexOf(r,T2) + 1
64:        D.append(r)
65:    else
66:        r' ← L3.next
67:        cursor ← indexOf(r',T2) − MoveDowns(l,FALSE) + 1
68:        r.pos ← indexOf(r,T2) - MoveDowns(r,TRUE)
69:        IssueCommand (move r.pos, cursor)
70:    end if
71: end while                                                    INRIA
72: for d ← 1 to sizeOf(D) do
73:    r ← D[d]
74:    IssueCommand (del r.pos)
75: end for
```

Algorithm 4: Efficient Type II Deployment.

```
1: MoveUps(r, add)
2: /* The fucntion MoveUps takes two arguments r and add. The function returns the number of rules N, which are already moved up, having
   initial position below the initial position of r. If add=TRUE, then r is added to the list M for future rule comparison, otherwise it is not
   added to the array M. The value of N is determined using Binary search */
3: start ← 1 , last ← sizeOf(T2) , count ← 0
4: i ← (start + last) ÷ 2 // Integer Division
5: while i ≠ indexOf(r, T2) do
6:     if i < indexOf(r, T2) then
7:         if add = TRUE then
8:             M[i] ← M[i] + 1
9:         end if
10:        start ← i + 1
11:    else
12:        count ← count + M[i]
13:        last ← i − 1
14:    end if
15:    i ← (start + last) ÷ 2
16: end while
17: if add = TRUE then
18:    M[i] ← M[i] + 1
19: else
20:    count ← count + M[i]
21: end if
22: return count
23:
24:
25: MoveDowns(r, add)
26: // Same as MoveUps, except that statements 8 and 12 are interchanged.
```

Algorithm 5: MoveUps and MoveDowns Functions.

In the next phase, T_2 is considered as the target policy. All the rules to be inserted or moved up are placed at their correct position according to T_2 (lines 30-48). After the completion of this phase, T_2 becomes the running policy. Starting from first rule in T_2, the algorithm traverses T_2 and perform move up and insert operations. As a rule r is encountered in T_2, its current position in R is computed and updated by using the following relation:

Current Position of r in R← r's position in I + M + N

where, M is the number of rules already inserted in R and N is the number of rules that are already moved up that initially appear below r in I. The value of N is determined in $\log|T|$ steps in the function *MoveUps* (lines 67-84). All rules in L, and the rules moved up or inserted to R are placed in L_3.

In the last phase T_2 is converted to T (lines 51-63). Editing commands are generated for rules to be moved down or deleted from R and the current position of each rule r is determined and updated using the following equation:

Current Position of r in R← r's position in T_2 - U

where U is the number of rules already moved down having initial position above the initial position of r. The value of U is determined in $\log|T|$ steps in a way similar to M except that counts are maintained for rules that are already moved down. Phase II and III takes O(k$\log k$) time in total, where k is equal to $|T_2| - |LCS(I,T)|$

5.3 Safety and Correctness of EFFICIENTDEPLOYMENT

It can be shown that EFFICIENTDEPLOYMENT is both safe and correct. Firstly, we show that T_2 is safe w.r.t. I and T. Starting from the first rule in T, any rule to be inserted or moved up is immediately appended at the end of T_2. Once a rule r is appended to T_2, no other rule can be placed before r in T_2. This means that the order of rules to be moved up or inserted is according to T. When a rule r' of L is encountered in T, then I is traversed until r' is found in I, and all rules to be moved down or deleted are appended to T_2. Again, once a rule to be moved down or deleted is appended to T_2, no rule can be appended before it. Thus, the order of rules to be moved down or deleted is according to the order in I. Hence, the condition (b) of Theorem 1 is satisfied. Also, it is evident that the $T_2 = I \cup T$, therefore the condition (a) of Theorem 1 is also satisfied and T_2 is safe w.r.t. I and T.

In Phase II, T_2 is traversed from the start and editing commands are generated for rules to be moved up or inserted in R. As these rules follows the order of T, this implies that a rule in T that is to be moved up or inserted is placed correctly before any rules that appear after it in T. Similarly, in the Phase III, editing commands are generated for all the elements to be moved down or deleted from R. This phase perform operation in reverse order, that is a rule that appear towards the end of T_2 is placed correctly before any rules that appear before it. Thus, at any moment during the deployment, the rules in R follows the order in I or T. Hence, the condition (b) of Theorem 1 is satisfied.

Initially, R is equal to I. During phase II, all the rules in T that are not in I are inserted in R and no rule is deleted from R. This means that at the end of Phase I, R is equal to $I \cup T$ and condition(a) of Theorem 1 is satisfied. In Phase III, all rules that are in I but not in T are deleted from R and no new rule is inserted in R. Thus, at the end of phase III, R contains only rule in T and the ordering of rule is exactly the same as in T as described in the previous paragraph. Therefore, R becomes T and the first condition(a) of Theorem 1 is also satisfied. Hence, the deployment is correct and safe.

6 Performance Evaluation

To evaluate the performance of EFFICIENTDEPLOYMENT, we try to follow the same set of test cases as in [24]. 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. The results of each test on policies 1-4 are given in the table below. 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 [24] for computing a safe deployment. All times are represented in seconds.

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	.00783	.01200	.01646	.02300	.03585	.04400	.08027	.24200
Test 2	.00704	.01200	.01813	.02800	.04721	.04900	.08116	.28300
Test 3	.00684	.03800	.01859	.04900	.03826	.07000	.08409	.32500
Test 4	.00684	.04000	.01837	.20500	.03713	1.3820	.08247	12.582
Test 5	.00696	.07000	.01687	.38700	.03454	4.3920	.08761	26.983

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 [24], 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.

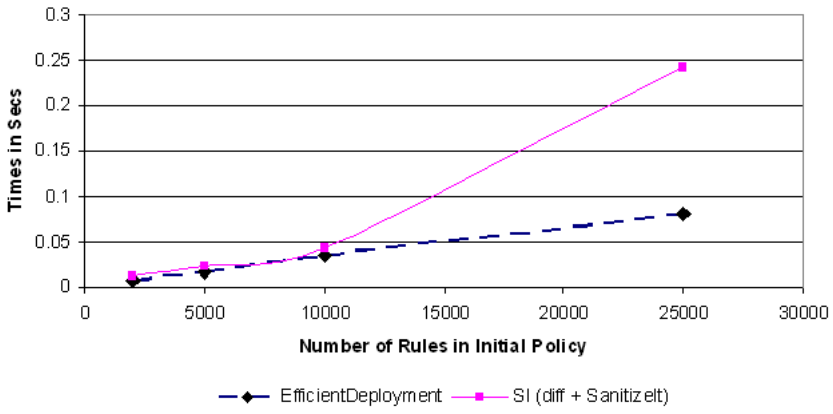

Figure 2: Comparison of EFFICIENTDEPLOYMENT and SanitizeIT for Test 1

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 shown that recent approaches [24]to firewall policy deployment contain critical errors. Indeed, these approaches can introduce temporary security holes that permit illegal traffic and/or interrupt network services by blocking legal traffic during a deployment. We have proposed 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 two correct algorithms: the first one is efficient and partial-safe. The second one is safe but it is less-efficient as it generates some extra rules in order to ensure safety. For Type II policy editing languages, we have presented an approximatively 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.

References

[1] Check Point SmartCenter. http://www.checkpoint.com/products/smartcenter/.

[2] Cisco Security Manager. http://www.cisco.com/en/US/products/ps6498/index.html.

[3] Entrasys Matrix X Core Router. http://www.entrasys.com/products/routing/x/.

[4] F-Secure. Malware information pages: Slammer. http://www.f-secure.com/v-descs/mssqlm.shtml.

[5] F-Secure. Malware information pages: Worm:w32/downadup.al. http://www.f-secure.com/v-descs/worm_w32_downadup_al.shtml.

[6] Juniper Network and Security Manager. http://www.juniper.net/us/en/local/pdf/datasheets/1100018-en.p

[7] E. Al-Shaer and H. Hamed. Modeling and Management of Firewall Policies. *Network and Service Management, IEEE Transactions on*, 1(1):2–10, April 2004.

[8] M. Anwar, M. Zafar, and Z. Ahmed. A Proposed Preventive Information Security System. In *International Conference on Electrical Engineering, 2007. ICEE '07.*, pages 1–6, 2007.

[9] F. Baboescu and G. Varghese. Fast and Scalable Conflict Detection for Packet Classifiers. In *ICNP*, pages 270–279, 2002.

[10] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. Firmato: A Novel Firewall Management Toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[11] L. Bergroth, H. Hakonen, and T. Raita. A Survey of Longest Common Subsequence Algorithms. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[12] S. Cobb. ICSA Firewall Policy Guide v2.0. Technical report, NCSA Security White Paper Series, 1997.

[13] G. Cormode, S. Muthukrishnan, and S. C. Sahinalp. Permutation Editing and Matching via Embeddings. In *ICALP*, pages 481–492, 2001.

[14] M. Englund. Securing systems with host-based firewalls. In *Sun BluePrints Online*, September 2001.

[15] Z. Fu, S. F. Wu, H. Huang, K. Loh, F. Gong, I. Baldine, and C. Xu. IPSec/VPN Security Policy: Correctness, Conflict Detection, and Resolution. In *POLICY*, pages 39–56, 2001.

[16] M. G. Gouda and A. X. Liu. Firewall Design: Consistency, Completeness, and Compactness. In *ICDCS*, pages 320–327, 2004.

[17] H. Hamed and E. Al-Shaer. Dynamic rule-ordering optimization for high-speed firewall filtering. In *ASIACCS*, pages 332–342, 2006.

[18] S. Karen and H. Paul. Guidelines on Firewalls and Firewall Policy. *NIST Recommendations*, SP 800-41, July, 2008.

[19] A. X. Liu. Change-impact analysis of firewall policies. In *ESORICS*, pages 155–170, 2007.

[20] J. Qian. ACLA: A framework for Access Control List (ACL) Analysis and Optimization, booktitle = Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues of the New Century. page 4, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[21] L. Qiu, G. Varghese, and S. Suri. Fast firewall implementations for software and hardware-based routers. In *International Conference on Network Protocols*, pages 155–170, 2001.

[22] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[23] T. Ylönen. SSH: secure login connections over the internet. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[24] C. C. Zhang, M. Winslett, and C. A. Gunter. On the Safety and Efficiency of Firewall Policy Deployment. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 33–50, Washington, DC, USA, 2007. IEEE Computer Society.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399