On the Safety and Efficiency of Firewall Policy Deployment

Charles C. Zhang Marianne Winslett Carl A. Gunter
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue
Urbana, IL 61801, USA
{cczhang, winslett, cgunter}@cs.uiuc.edu

Abstract

Firewall policy management is challenging and error-prone. While ample research has led to tools for policy specification, correctness analysis, and optimization, few researchers have paid attention to firewall policy deployment: the process where a management tool edits a firewall's configuration to make it run the policies specified in the tool. In this paper, we provide the first formal definition and theoretical analysis of safety *in firewall policy deployment. We show that naive deployment approaches can easily create a temporary security hole by permitting illegal traffic, or interrupt service by rejecting legal traffic during the deployment. We define* safe *and* most-efficient *deployments, and introduce the* shuffling theorem *as a formal basis for constructing deployment algorithms and proving their safety. We present efficient algorithms for constructing most-efficient deployments in popular policy editing languages. We show that in certain widely-installed policy editing languages, a safe deployment is not always possible. We also show how to leverage existing* `diff` *algorithms to guarantee a safe, most-efficient, and monotonic deployment in other editing languages.*

1 Introduction

The entangled cyberspace of the Internet and intranets has never been a safe place, but it would have been much worse without firewalls. A firewall, also called a border protection device, sits on the border of networks, controls traffic between different trust zones, and serves as the first line of defense against unauthorized or malicious accesses.

The process of configuring firewall policies is difficult and error prone. Studies have shown that it is likely that the majority of real world firewall policies have configuration errors [36]. The size and complexity of network topologies are still increasing, and so are the size and complexity of firewall policies. Policies containing 10K rules are not uncommon in commercially deployed firewalls, and we have seen a firewall configured with 50K rules. Manually configuring such policies has clearly become mission impossible even for guru network administrators.

To address the policy specification problem, policy provisioning and management have received a great deal of attention [16, 26, 21], as have conflict detection [13, 20, 15, 14, 38] and optimization [31, 22, 11]. In parallel with this academic research, firewall management tools such as Cisco Security Manager [4], Juniper Networks' Netscreen-Security Manager [8], and Check Point SmartCenter [1] have gained popularity with network administrators. Management tools provide intuitive graphical user interfaces (GUIs) to ease policy specification, and give convenient utilities for error detection and rule optimization. When a network administrator is satisfied with the policy configured through the GUI, he instructs the management tool to *deploy* it. The management tool then translates the needed policy changes into a format recognized by the firewall, typically in the form of lines of commands in text, and sends them to the firewall so that the new policy becomes the running policy.

A management tool aims to achieve four goals when deploying firewall policies: *correctness*, *confidentiality*, *safety*, and *speed.*

Correctness: A deployment is correct if the firewall device's running policy is replaced by the target policy configured through the GUI. Correctness is the most important goal. Some management tools take additional steps to verify the correctness of a completed deployment, e.g., by comparing checksums of the running policy and the target policy.

Confidentiality: Very often a management tool is a standalone application that uses certain transport protocols to communicate with the firewall device on a different network, including exchanging credentials for authentication and sending policy changes. Given the critical role that firewalls play, the communication between a management tool and its managed firewalls needs to be confidential: successful eavesdropping on any sensitive information in a deployment session gives an attacker the potential to seriously compromise the network's security. Deployment confidentiality can be readily addressed by existing cryptographic communication protocols such as SSH [37] and SSL [19].

Safety: A deployment is safe if it does not cause the firewall to drop legal traffic or permit illegal traffic during deployment. Though common sense tells administrators to focus on the correctness of firewall policies, an unsophisticated deployment approach can easily cause traffic interruption or a temporary security hole during deployment of even the most perfect policy. We give examples of unsafe deployments in section 3.

Speed: Slow deployments are unpleasant for users and can also raise security issues: very often a firewall policy change needs to be deployed immediately to close a hole for illegal access or to open access for highly urgent traffic. If a time-critical deployment is unnecessarily slow, it partly defeats the purpose of the deployment and jeopardizes the network's security.

In this paper, we make several contributions to the correctness, safety, and efficiency of firewall policy deployments. To the best of our knowledge, we provide the first formal definition and theoretical analysis of safety in policy deployment. In particular, the shuffling theorem we propose can be used as a basis for constructing deployment algorithms and proving their safety (section 4). We categorize firewall policy editing languages into type I and type II (section 3), give linear algorithms to calculate most-efficient type I deployments and prove that not all policies have a safe type I deployment (section 5). We show that the most-efficient type II deployment problem can be solved using existing algorithms, provide an efficient algorithm to sanitize any type II deployment, and argue that every policy has a type II deployment that is both safe and most-efficient (section 6). We present performance results from our sanitization algorithm implementation (section 7), discuss our approaches and give recommendations for firewall design (section 8), comment on related work (section 9), and finally give conclusions(section 10).

2 Firewall Background

Network administrators typically view the network as being divided into zones, with the hosts in the same zone being trusted to similar degrees. Common zones include the Internet, the demilitarized zone (DMZ), and the intranet, with increasing levels of trust. A typical firewall policy permits traffic going from hosts with higher trust levels to hosts of lower trust levels (e.g., from an intranet host to an Internet host), and denies traffic going in the other direction. Yet this is not always the case. Common public services such as web sites are typically put on the DMZ and made available to hosts from the Internet. Real world firewall policies are always ad hoc. For example, it is very typical for a firewall policy to deny a certain service to a list of specific hosts which are considered malicious.

A firewall controls traffic by examining the contents of network packets, which is why a firewall is also called a packet filtering device. Five packet fields are most commonly used for traffic filtering: protocol type, source IP address, source port, destination IP address, and destination port. We encode these as a 5-tuple: $\langle prot, src\_ip, src\_port, dst\_ip, dst\_port\rangle$.[1] In every packet, each of the five fields assumes a specific value, such as $\langle TCP$, 192.168.5.7, 1352, 10.1.1.1, 23$\rangle$. We call a 5-tuple with every field assigned a specific, indivisible value a *flow*. Multiple packets can have the same 5-tuple and hence belong to the same flow. Fields other than those in the 5-tuple, e.g., IP TOS (Type of Service) and TTL (Time to Live) values, are occasionally used in some firewalls, which will increase the dimensionality of the tuples in our flow definition, but does not affect the discussion in this paper.

A firewall *rule* r specifies an action, typically *accept* or *deny*, on a filtering set, which is a set of flows. If a packet p belongs to a flow in r's filtering set, we say that p *matches* r. A filtering set is usually specified using the same 5-tuple format as a flow, except that each field can assume either a specific value or a range of values; e.g., if the *src_ip* field is a network `192.168.1.0/24`, any host address on this network matches this field. A firewall *policy* is an ordered list of rules with a first-match semantics: when a new packet p comes in, the firewall checks p against its rules one by one, starting from the first rule and stopping when it reaches the first rule r that matches p. We say that p *hits* r in this case, and p is either permitted or denied as specified

[1] Packets using a protocol other than TCP/UDP may have the *src_port* and *dst_port* fields undefined or redefined, though these fields will still be encoded as integer values; e.g., the source and destination ports can be used to accommodate ICMP message types when the protocol is ICMP. This recoding will not affect our discussion.

```
a.  permit TCP 192.168.1.1 12.3.4.0/24 80
b.  deny IP 10.1.1.0/24 any
c.  permit UDP 172.20.0.0/16 any 123
d.  deny IP 10.1.2.0/24 76.54.32.1
e.  permit IP 10.0.0.0/8 any
```

Policy α

```
a.  permit TCP 192.168.1.1 12.3.4.0/24 80
f.  deny IP 10.1.1.1 any
c.  permit UDP 172.20.0.0/16 any 123
g.  permit IP 10.0.0.0/16 any
h.  permit IP 10.1.0.0/16 any
```

Policy β

Figure 1. Two Firewall Policies

by the action of r.[2]

If no matching rules are found, then a hidden, default *match-all* rule is applied with a default action. In this paper, we use *closed* firewall policies, meaning that the default match-all rule at the end of every policy is a *deny-all* rule that denies every packet. Our results in this paper apply to policies with a default permit-all rule as well.

As with most firewalls, we do not allow the same rule to appear more than once within a policy, since only the first occurrence is meaningful and the rest are always *shadowed* and never used by the firewall to accept or deny packets.

Figure 1 gives two sample firewall policies based on the PIX firewall language [3]. In this language, *any* means all hosts and the absence of a port value means all ports; e.g., rule a permits all TCP flows going from any port of host `192.168.1.1` to the port 80 of all hosts on the `12.3.4.0/24` network. The symbols such as a are for illustration purposes only. Although we use the PIX syntax in this example and the real-world policy deployment example in section 3, our discussion in this paper is independent of any specific policy syntax, as long as a policy consists of an ordered list of rules with first-match semantics and each rule specifies which packets to accept and denies the rest.

3 Policy Deployment

Similar to the user interface evolution of operating systems, early firewalls supported only a shell-like command line interface (CLI). As shown in Figure 1, a firewall rule can be typed in as a command line, and firewall policies can be viewed as text composed of command lines. Given the CLI tradition of firewalls and list-based policy structure, even today many firewalls still provide only CLI oriented interfaces for management tools, although encoded in a more modern format such as XML. From a firewall's perspective, a management tool is not fundamentally different from a human user.

[2]Other semantics are possible, such as the last-match semantics used by BSD Packet Filter. First-match semantics, however, is by far the most popular in firewall policies. Deployment problems in last-match semantics can be addressed in the same style as we propose in this paper, although the details will necessarily differ.

3.1 Policy Editing Languages

A management tool deploys a user's target policy by sending editing commands to transform the firewall's current policy. Modern firewalls typically use a subset of the following editing commands, *append* (`app` r), *delete* (`del` r), *numbered delete* (`del` i), *insert* (`ins` i r), and *move* (`mov` i j), where r stands for a rule, and i and j are position numbers. Not all firewalls support all these commands. The set of supported editing commands defines a firewall's policy editing language. While there are other variations on the market, we define type I and type II policy editing languages, which we consider the most representative. If a deployment uses only type I (resp. type II) commands, we call it a type I (resp. II) deployment.

Type I Editing Type I editing supports two editing commands, `app` and `del`. Command "`app` r" appends a rule r to the end of the running policy R, unless r is already in R, in which case the command fails. "`del` r" deletes r from R, if it is present. Type I editing is inefficient as it does not allow random editing of policy lines. The combination of `app` and `del`, however, is *complete* in that it can transform any initial policy into any target policy. A brute force approach is to delete all rules in the initial policy, then append one by one all the rules in the target policy. Type I editing started in the past when firewall policies were small and easy to edit, yet manages to survive as the editing language in the industry's relatively recent offerings like FWSM 2.x [2] and JUNOSe 7.x [7], which still have many installations.

Type II Editing To overcome the inefficiency of type I editing, newer firewalls have introduced type II editing by adding the `ins`, `del` and `mov` commands, which support rule position numbers. "`ins` i r" inserts r into the running policy R so that it becomes the ith rule, provided that r is not already in R prior to the insertion. "`del` i" deletes the ith rule. "`mov` i j" moves the ith rule to a new position so that it becomes the jth rule. Type II editing is both complete and efficient. Examples of Type II editing firewalls include SunScreen 3.1 Lite [18] and Enterasys Matrix X [5].

The effect of "`mov` i j" can be achieved by "`del` i" followed by "`ins` j r". This, however, does not mean `mov` is just syntactic sugar, as there is still a difference between the two approaches. The first deployment goes through only two states, with r at either position i or j. The second deployment has a third state between `del` and `ins`, where r is absent. There are firewalls that support `ins` and `del`, but not `mov`, which we will discuss in section 8.

3.2 Deployment Efficiency

As network communication cost and CLI processing time are directly proportional to the number of editing commands sent by a management tool, an effective way to speed up the deployment is to minimize the number of editing commands. We say a deployment is *most-efficient* for policy I and T iff it consists of the smallest possible number of editing commands in a given editing language to transform I into T. We consider only the total number of commands because the variation in deployment time for different types of commands is typically negligible, as most deployment time is spent in network transit and other fixed per command maintenance costs. Later we will show that a most-efficient type I (resp. type II) deployment has the smallest number of `apps` (resp. `ins`) and the smallest number of `dels` among all type I (resp. type II) deployments, if we count one `mov` as one `del` plus one `ins` in type II editing. Consequently, a most-efficient deployment can be expected to take close to the minimum possible deployment time.

In this paper, we require the running policy after deployment to be identical to the target policy. If the target policy is redundant and can be reduced to a smaller size while still semantically equivalent, a management tool can always apply optimization techniques prior to deployment, so that the target policy submitted to the deployment task is already "optimal". Policy optimization usually aims for redundancy removal. In contrast, theoretically it is possible to "optimize" the target policy just to accelerate the deployment, e.g., rewrite the target policy to make it syntactically closer to the initial policy. While we are not aware of any management tool that practices this kind of optimization, it can be applied prior to policy deployment and is orthogonal to our discussion of most-efficient deployment.

3.3 Unsafe Deployments

Two types of traffic safety anomalies can occur during an unsafe deployment: traffic interruption caused by dropping legal traffic and security holes caused by temporarily permitting illegal traffic. Suppose we need to deploy policy β in Figure 1 to a firewall running policy $R = \alpha$. Consider IP packet p_1 with source IP address `10.1.1.1` and p_2 with `10.1.2.3`. We consider p_1 illegal because both b in α and f in β deny it, and p_2 legal because both e in α and h in β accept it. We do not need to edit the shared rules a and c in R, but we need to delete b, d, and e, and insert f, g, and h at the right positions. If we delete b first, R becomes $[a, b, c, d]$, which permits the illegal packet p_1 as it matches d. If we delete d first, R becomes $[a, b, c]$, which denies the legal packet p_2 as it matches none of the rules and gets denied by default.

In the above example, the deployment can finish quickly and the two safety anomalies will disappear. But depending on the size of the deployment load, the network bandwidth, and the firewall's processing power, real-world deployments can take anywhere from a couple of seconds to tens of minutes, which may provide sufficient opportunity for safety anomalies to be exploited to pose serious threats. In 2003, the fast-spreading Sapphire/Slammer worm [28] had a peak scanning rate of over 55 million hosts per second, and managed to infect 90% of all vulnerable hosts across the Internet within 10 minutes. Similarly, malicious parties can have precompiled attacks that constantly probe for vulnerabilities caused by unsafe deployments. They can also use traffic analysis and other techniques to detect and predict deployment schedules, to make such attacks more effective.

4 Safe Deployment Formalization

Definition 1 (Denial-safe)**.** *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 policy and the target policy.*

We say that the running policy *falsely* denies a packet during a deployment if this packet is accepted by both the initial and target policies. Denial-safe deployments do not have false denials. On the other hand, a deployment that is not denial-safe can falsely deny legal packets and cause traffic interruption.

Traffic interruption is intolerable in mission-critical networks and should be avoided whenever possible. For

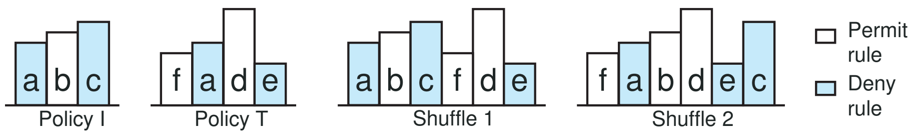

Figure 2. Policy Shuffle

example, the network may be carrying real-time control and data for physical systems such as medical monitoring devices or power generators, where an interruption of service can have life-threatening implications. In a less critical scenario, if an ISP's firewall policy deployment temporarily disconnects all through traffic for tens of seconds, it inconveniences its customers and may cause them financial loss. As a special case, if the firewall management tool that initiates the deployment happens to reside outside the firewall, the false denials can prevent the deployment from finishing, which effectively turns the deployment into a denial-of-service attack.

Definition 2 (Permission-safe)**.** *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. the initial policy and the target policy.*

We say that the running policy *falsely* permits a packet during a deployment if this packet is permitted by neither the initial nor the target policies. Permission-safe deployments do not have false permissions. A deployment that is not permission-safe can falsely permit illegal packets and introduce security holes.

Definition 3 (Safe Deployment)**.** *Policy A is safe w.r.t policies B and C iff it is both denial-safe and permission-safe w.r.t. B and C. A deployment is safe iff it is both denial-safe and permission-safe.*

A safe deployment neither falsely denies nor falsely permits a packet during the deployment. A firewall policy x can be categorized by $\mathcal{P}(x)$, the set of packets that it permits, or by $\mathcal{D}(x)$, the set of packets that it denies. Let predicate $Safe(x, I, T)$ denote that x is safe w.r.t. the initial policy I and target policy T of a deployment. We have the following tautologies, which directly lead to Proposition 1.

$$Safe(x,\, I,\, T) \Longleftrightarrow (\mathcal{P}(I) \cap \mathcal{P}(T)) \subseteq \mathcal{P}(x) \subseteq (\mathcal{P}(I) \cup \mathcal{P}(T))$$

$$Safe(x,\, I,\, T) \Longleftrightarrow (\mathcal{D}(I) \cap \mathcal{D}(T)) \subseteq \mathcal{D}(x) \subseteq (\mathcal{D}(I) \cup \mathcal{D}(T))$$

Proposition 1. *During a safe deployment, any packet that is permitted by both the initial and target policies is always permitted; any packet that is denied by both the initial and target policies is always denied.*

Definition 4 (Policy Shuffle)**.** *Let A and B be two non-empty firewall policies. A shuffle of A and B is a policy S generated by the following procedure: (1) Initially S is empty; let A' and B' be a copy of A and B respectively. (2) Let r be the first rule of A' or B'; remove r from its policy. Unless r is already in S, append r to the end of S. (3) Repeat step 2 until both A' and B' are empty.*

If a rule r of shuffle S comes from A (respectively B), then every rule before r in A (resp. B) also shows up before r in S. If A and B do not have any rules in common, the shuffling procedure is the same as shuffling two decks of cards, with each card representing a unique rule. Figure 2 shows two possible shuffles of two policies.

Let a *prefix* of policy P be a policy obtained by removing zero or more rules from P's end.

Theorem 1 (**Safe Shuffling Theorem**)**.** *Let A and B be two firewall policies. Every shuffle of A and any prefix of B is safe w.r.t. A and B.*

Proof. We first prove that every such shuffle S is permission-safe. Let p be a packet that hits a permission rule r in S. Examining the shuffling procedure that generates S, we know r comes from either A or B. If r comes from A, then we know that every rule before r in A also shows up before r in S. Since p does not match any rule before r in S, p does not match any rule before r in A either. This means p is guaranteed to hit r in A and will be permitted, i.e., S is permission-safe w.r.t. A and B.

To prove S is denial-safe, we follow the same reasoning as before, except when p matches none of the rules of S and will be denied by the default deny-all rule. In that case, p matches none of the rules of A either and will be denied by A. Thus S is also denial-safe w.r.t. A and B. □

Let I and T be a deployment's initial and target policies, respectively. A deployment algorithm $\mathcal{A}$ takes

I and T as input, and generates a sequence of editing commands to be sent to the firewall to transform I into T. From the shuffling theorem, we have that every shuffle of I and any prefix of T is safe w.r.t. I and T, and so is every shuffle of T and any prefix of I. This constitutes a theoretical basis for constructing algorithms for safe deployment.

A firewall has a new running policy every time an editing command is applied. Thus a deployment can be viewed as a sequence of running policies $I = R_0, R_1, \ldots, R_{n-1}, R_n = T$, with R_{i+1} derived by applying an editing command to R_i. For a packet p and two consecutive policies R_i and R_{i+1}, if one policy permits p and the other denies p, we say p is *flipped* during the deployment. If p is accepted (or denied) by both I and T, it can get flipped an even number of times during deployment, but cannot be flipped at all if the deployment is safe. If p is accepted (resp. denied) by I but denied (resp. accepted) by T, it needs at least one flip for the deployment to be correct, but can be flipped any odd number of times, even in a safe deployment. In that sense, a safe deployment is not necessarily *monotonic.*

Definition 5 (Monotonic Deployment)**.** *A deployment is* monotonic *iff every packet is flipped at most once during the deployment.*

Monotonic deployments are desirable in practice because they provide traffic stability: once an illegal (resp. legal) traffic is denied (resp. permitted), it will stay that way during the remainder of the deployment. We view monotonicity as providing a higher degree of safety: a monotonic deployment is always safe, but a safe deployment is not necessarily monotonic. Even a monotonic deployment is not necessarily most-efficient. For example, suppose a, b, and c are permission rules. A deployment that takes two `mov`s to move a and b up in policy [c, a, b] to form the target [a, b, c] is monotonic, because no flips happen. A most-efficient deployment, however, takes only one step by moving c down.

5 Type I Deployment

Type I editing only supports `del` and `app`, which makes policy deployment inefficient even for what seems to be a small policy change. For example, if we want to insert a new rule at the beginning of the initial policy, we need to delete all its rules so that the appended rule becomes the first, and put all deleted rules back again. In some cases, however, a subsequence of the initial policy can be preserved. We need to selectively delete and append rules in order to make the deployment most-efficient.

Algorithm 1 Scanning Deployment

```
ScanningDeployment ($I$, $T$) {
    /* An algorithm using only app and del */
    /* to transform policy $I$ into policy $T$ */

    $\mathcal{S} \leftarrow$ empty stack
    $\mathcal{H} \leftarrow$ empty hash table
    /* Phase 1: add rules */
    $i \leftarrow 1$
    for $t \leftarrow 1$ to SizeOf($T$) do
        while $i \leq$ SizeOf($I$) and $I[i] \neq T[t]$ do
            /* $I[i]$ needs to be deleted */
            $\mathcal{S}$. PUSH ($I[i]$)
            $\mathcal{H}$.ADD($I[i]$)
            $i \leftarrow i + 1$
        if $i >$ SizeOf($I$) then
            if $\mathcal{H}$.Contains($T[t]$) then
                $\mathcal{H}$.Remove($T[t]$)
                IssueCommand( del $T[t]$)
            IssueCommand( app $T[t]$)

    /* Phase 2: clean up */
    for $j \leftarrow$ SizeOf($I$) down to $i$ do
        IssueCommand( del $I[j]$)
    while not $\mathcal{S}$.IsEmpty() do
        $r \leftarrow \mathcal{S}$.POP()
        if $\mathcal{H}$.Contains($r$) then
            IssueCommand( del $r$)
}.
```

A theoretically appealing alternative is to declare type I editing obsolete and ignore it. We cannot do this, as type I firewalls are still on the market. Further, the large installed customer base for type I products will include many customers who resist upgrading to a newer product due to the additional cost, time, training, and hassle.

We start by giving a simple deployment algorithm for an initial policy I and target policy T. I and T are coded as arrays, so that $I[i]$ refers to the ith rule of I. Initially the running policy R equals I. In phase 1, the algorithm appends to the end of R every rule r in T, starting from $r = T[1]$. If r is already in I, then it removes r from R before appending it back. In phase 2, it removes from R every rule r that is in I but not T, starting from the last rule in I. Let $|X|$ be the number of rules in X, and c_1 be the number of rules that are in both T and I. The total number of commands generated in this algorithm is $|I| + |T|$, which is independent of the similarity between I and T, and suggests that the algorithm is not most-efficient. In phase 1, R is always a shuffle of I and a prefix of T, except for the intervals between deleting the c_1 shared rules and appending them back. In phase 2, R is always a shuffle of T and a prefix of I. By the shuffling theorem, R is safe during the deployment, except for c_1 intervals, assuming all intervals between subsequent editing commands are even.

We provide ScanningDeployment (Algorithm 1) as a more sophisticated approach to calculate a most-efficient deployment for I and T. In phase 1, the algorithm selectively appends the rules in T to R. It first sets t and i to 1, and then increases their values while maintaining the invariant that $T[1, \ldots, t]$ is a subsequence of $R[1, \ldots, i']$, where i' is the index of $I[i]$ in R. In each step, it increases t by 1, and then keeps increasing i until the invariant is maintained. When i goes beyond the end of I, $T[t]$ is appended to R, which still makes $T[1, \ldots, t]$ a subsequence of R. (If $T[t]$ already appears in R, it is deleted first.) In phase 2, all rules in I but not T are removed from R one by one, starting from the one that appears closest to the end of I.

To prove that Algorithm 1 is most-efficient, we show that every `del` and `app` it ever generates also needs to take place sooner or later in every correct deployment. Initially R equals I. At the beginning of phase 1, the algorithm scans through I to look for rule $T[t]$, where $t = 1$. Any rule $I[i]$ encountered on the way that is not $T[1]$ *must* be deleted in every correct deployment; otherwise $I[i]$ would occur before $T[1]$ in the final R, but the final R should have $T[1]$ as the tth rule ($t = 1$). If $T[1]$ is not in I, $T[1]$ *must* be appended in every deployment; otherwise it would be absent from the final R either because of its absence in I or a previously-described necessary `del`. In the latter case, the `del` needs to take place before this `app` can take place, as duplication is not allowed in R. If it finds $T[1]$ in I, $T[1]$ can be preserved in R. Then the algorithm is done with $T[1]$ and t becomes 2. The necessity of every `del` and `app` for $t = 2$ follows the same logic; continue on until the end of phase 1. In phase 2, every rule deleted in the `for` loop is in I but not in T, hence needs to be deleted in every deployment. Every `del` issued in the second while loop is an execution of a previously-described necessary deletion.

We have argued that every command generated by ScanningDeployment must occur in every correct deployment. We can also prove that the algorithm deletes (and appends) each rule at most once. Thus Algorithm 1 generates a most-efficient deployment. Let c_2 be the number of rules in T's longest prefix that is a subsequence of I. The number of editing commands this algorithm generates is $|I| + |T| - 2c_2$. Assuming that a hash table lookup takes constant time, the algorithm requires $O(n)$ time and space, where n is the larger of $|I|$ and $|T|$.

Algorithm 1 buffers `del`s in a stack and issues them later in reverse order, as an optimization for safety. Consequently, the deployment is safe when I and T do not share any rule: at any moment in phase 1, the running policy R consists of I at the beginning followed by a prefix of T, which is safe by the shuffling theorem. In phase 2, rules in I are deleted in reverse order, so that R consists of T at the end preceded by a prefix of I, which is also safe. This heuristic, however, does not guarantee safety when I and T have rules in common. Similar to the analysis in the previous algorithm, we find that there are $c_1 - c_2$ intervals during which the deployment is not guaranteed to be safe. Without considering deployment efficiency, is there an algorithm that always calculates a safe type I deployment?

Theorem 2. *Not all firewall policies can be deployed safely by appending and deleting only the rules that are in the initial or target policies.*

Proof. Let I and T be $[a, b]$ and $[b, a]$ respectively, where a=“`permit` A”, b=“`deny` B”, and A and B are packet filters. Packet p is in A but not B. During any deployment, a is the first rule in the running policy R until `del` a takes place, and `del` a has to take place as a is not the first rule in the final R. Then between the time a is deleted and appended, p is denied by the default “`deny-all`” rule. But p is permitted in both I and T. □

6 Type II Deployment

6.1 Most-efficient Deployment

To find a most-efficient type II deployment to transform policy I to T, we can leverage the extensively studied `diff` problem of determining the differences between two sequences of symbols. The `diff` problem calculates the minimal number of deletions and insertions, also called *edit distance*, to convert one sequence to another. This is also equivalent to the problem of finding a longest common subsequence of two sequences [29]. Treating every rule as a symbol, we can use `diff` to calculate an editing command sequence D that contains the minimal number of `del` and `ins` commands to transform I into T. Since firewall policies do not allow duplicate rules, a policy can be viewed as a permutation of a sequence of rules. Let c_3 be the number of rules in a longest common subsequence of I and T. Since we need to touch all the commands but those in one of the longest common subsequences [29], the number of commands in D is $[D] = |I| + |T| - 2c_3$. From the definitions of c_2 and c_3, we know that c_3 is at least as big as c_2; so D is more efficient than the most-efficient type I deployment, which takes $|I| + |T| - 2c_2$ steps.

Given that D has the minimal number of `del` and `ins` commands, we have the following immediate observations. If a rule r is in I but not T, there exists one and only one command in D that deletes r; if r is in T but not I, there exists one and only command in D that inserts r. If r is in both I and T, r may be absent in D, which means r stays untouched; if r shows up in D, then D has to delete r first, then insert it back sometime later, as otherwise r will be missing in T. Thus deletions and insertions of the same rule in I and T always show up in pairs in D. Such a pair can occur at most once, as otherwise we can keep the last pair of `del` and `ins` and derive a deployment sequence with fewer steps. It follows that a `del` and `ins` command pair on r can be replaced by a `mov` command, and D can be translated into a type II deployment command sequence D' that has the minimal number of `del`, `ins`, and `mov` commands. Thus an extension of a `diff` algorithm can be used to calculate (possibly unsafe) most-efficient type II deployments. Since there are $c_1 - c_3$ pairs of `del` and `ins` in D that can be replaced by `mov`, the number of commands changes from $|I| + |T| - 2c_3$ for D to $|I| + |T| - c_1 - c_3$ for D'.

There are many `diff` algorithms available. For permutation sequences, the fastest algorithms are $O(n \log n)$ and $O(d\,n)$, where d equals $|D|$ and n is the larger of $|I|$ and $|T|$. As policy deployment usually involves small changes, $O(d\,n)$ algorithms are generally faster. In the worse case, however, d can be almost as big as $2n$, in which case $O(n \log n)$ is faster.

6.2 Safe Deployment

6.2.1 Greedy Two-phase Deployment

We provide a greedy two-phase algorithm, named TwoPhaseDeployment, to calculate a safe type II deployment for policies I and T. In phase 1, the algorithm inserts the rules of T at the beginning of the running policy R. When a rule to be inserted is already in R, it gets moved up to the right position instead. In phase 2, all rules that are in I but not T are deleted, starting at the end of I. This is described in Algorithm 2.

At any moment in phase 1, the running policy R is a shuffle of I and a prefix of T; so R is safe based on the shuffling theorem. When phase 1 is done, the running policy has T at the beginning, followed by I minus the rules in both I and T. At any moment in phase 2, the running policy R is a shuffle of T and a prefix of I, so R is safe. This means the deployment generated by Algorithm 2 is safe. When phase 2 finishes, the running policy becomes T, i.e., the deployment is also correct. The number of editing commands this algorithm generates is $|I| + |T| - c_1$. Let n be the larger of $|I|$ and $|T|$. We assume that the operator $\notin$ used in lines 8 and 16 will take constant time, if the rules are put into a hash table first. The function `IndexOf`(x, X) returns the position of x in array X, and we assume it requires constant

Algorithm 2 Greedy 2-Phase Deployment

```
 1. TwoPhaseDeployment ($I$, $T$) {
 2.     /* algorithm to calculate a safe type II deployment */
 3.     /*  to transform firewall policy $I$ into $T$ */
 4.
 5.     /* Phase 1: insert and move */
 6.     $inserts \leftarrow 0$
 7.     for $t \leftarrow 1$ to SizeOf($T$) do
 8.         if $T[t] \notin I$ then
 9.             IssueCommand(ins $t$ $T[t]$)
10.             $inserts \leftarrow inserts + 1$
11.         else
12.             IssueCommand( mov IndexOf($T[t]$ , $I$) + $inserts$ $t$)
13.
14.     /*  Phase 2: backward delete */
15.     for $i \leftarrow$ SizeOf($I$) down to 1  do
16.         if $I[i] \notin T$ then
17.             IssueCommand( del $i + inserts$)
18. }.
```

time if we store all the rule-to-position mappings in a hash map, which takes $O(n)$ time to construct. Thus the worst case running time of Algorithm 2 is $O(n)$.

6.2.2 The Sanitization Algorithm

Very often, deployment safety and efficiency do not coincide. Algorithm 2 calculates a safe deployment for I and T, but rarely produces a most-efficient deployment. A `diff` algorithm calculates a most-efficient deployment, but it is not necessarily safe. For example, running the Linux implementation of `diff` on α and β from Figure 1 gives an unsafe deployment sequence `del` b, `ins` f, `del` d, `del` e, `ins` g, and `ins` h (position numbers omitted). This is not surprising, as `diff` knows nothing about firewall policies.

Nonetheless, can a type II deployment be both safe and most-efficient? Most-efficient deployments are not unique; e.g., when only the relative order of two rules is different in I and T, we have the option to move one rule up or the other down. Even for a given sequence of editing commands, the order can be rearranged: we can reposition a `del` at a different step without affecting the eventual editing outcome. So it is with `ins` as well, and even `mov`, provided that we adjust the position parameters accordingly. This suggests the possibility of refining a deployment by rearranging (possibly dropping) its steps and adjusting the position parameters, so that the resulted deployment becomes safe, yet without increasing the number of steps. We call this *sanitization*. Encouragingly, we show that every type II deployment of a firewall policy can be sanitized, and the sanitization can be done efficiently.

In Algorithm 3, we present a fast sanitization algorithm called SanitizeIt to sanitize any given deployment $\mathcal{D}$ for I and T, where $\mathcal{D}$ consists of a sequence of type II commands that transforms the running policy R from I to T. SanitizeIt outputs a safe type II de-

Algorithm 3 Sanitization Algorithm

```
SanitizeIt ($I$, $T$, $\mathcal{D}$) {
    /* $\mathcal{D}$: a deployment sequence of type II commands */

    /* initialize */
    $\Delta \leftarrow$ set of all rules in $\mathcal{D}$
    $\mathcal{U} \leftarrow$ empty set
    $DM, iDM \leftarrow$ empty array
    $idxR \leftarrow$ array of size SizeOf($\Delta$)
    for $i \leftarrow 1$ to SizeOf($I$) do
        if $I[i] \in \Delta$ then
            Append $I[i]$ to $DM$
            if $I[i] \in T$ then
                Append IndexOf($I[i]$ , $T$) to $iDM$
            else
                Append $i +$ SizeOf($T$) to $iDM$
    /* count each rule's leftside preceding rules in $DM$ */
    $lPreds \leftarrow$ BinaryCount($iDM$)
    /* count each rule's rightside preceding rules in $DM$ */
    $rPreds \leftarrow$ Reverse(BinaryCount(Reverse($iDM$)))

    /* Phase 1: insert or move up */
    $inserts \leftarrow 0$
    $i \leftarrow 1$
    for $t \leftarrow 1$ to SizeOf($T$) do
        $r = T[t]$
        if $r \notin I$ then
            /* Case 1: need to insert $r$ */
            $\mathcal{U}$.Add($r$)
            IssueCommand(ins SizeOf($\mathcal{U}$) $r$)
            $insert \leftarrow insert$+1
            $\Delta$.Remove($r$)
        else if $r \notin \Delta$ then
            /* Case 2a: $r$ stays untouched. Search for $r$ in $I$ */
            while $i \leq$ SizeOf($I$) and $I[i] \neq r$ do
                if $I[i] \notin \mathcal{U}$ then
                    $\mathcal{U}$.Add($I[i]$)
                    /* save $I[i]$'s position in the running policy */
                    $idxR$[IndexOf($I[i]$ , $DM$)] $\leftarrow$ SizeOf($\mathcal{U}$)
                $i \leftarrow i + 1$
            $i \leftarrow i + 1$
            $\mathcal{U}$.Add($r$)
        else if $r \notin \mathcal{U}$ then
            /* Case 2b: need to move $r$ up */
            $cur \leftarrow$ IndexOf($r, I$)+
              $rPreds$[IndexOf($r$, $DM$)] + $inserts$
            $\mathcal{U}$.Add($r$)
            IssueCommand( mov $cur$ SizeOf($\mathcal{U}$))
            $\Delta$.Remove($r$)
    /* process the remaining commands in $I$ */
    while $i \leq$ SizeOf($I$) do
        if $I[i] \notin \mathcal{U}$ then
            $\mathcal{U}$.Add($I[i]$)
            /* save $I[i]$'s position in the running policy */
            $idxR$[IndexOf($I[i]$ , $DM$)] $\leftarrow$ SizeOf($\mathcal{U}$)
        $i \leftarrow i + 1$

    /* Phase 2: backward delete or move down */
    for $j \leftarrow$ SizeOf($DM$) down to 1 do
        if $DM[j] \in \Delta$ then
            if $DM[j] \notin T$ then
                IssueCommand( del $idxR[j]$)
            else
                $pos \leftarrow$ IndexOf($DM[j]$ , $T$) + ($j - lPreds[j]$)
                IssueCommand( mov $idxR[j]$ $pos$)
            $\Delta$.Remove($DM[j]$)
}.
/* auxiliary counting algorithm */

BinaryCount($A$) {
    /* For every element $A[i]$, count the number of elements
    in $A$ that are to the left of and smaller than $A[i]$; e.g., if
    $A$=[2,4,1,3], there are zero elements to the left side of 2,
    one element $A[1] = 2$ to the left of and smaller than 4, zero
    elements to the left side of and smaller than 1, . . ., so it
    returns [0,1,0,2]. */

    $preds \leftarrow$ array of size SizeOf($A$)
    if $A$ is empty then
        return $preds$
    $preds[1] \leftarrow 0$
    $root \leftarrow$ CreateBinaryNode($A[1], null, null, 1$);
    for $i \leftarrow 2$ to SizeOf($A$) do
        $preds[i] \leftarrow$ InsertAndCount($A[i]$ , $root$)
        /* can optionally do AVL insertion rebalance here */
    return $preds$
}.

InsertAndCount($e$, $root$) {
    /* insert $e$ into the tree, and return the number of elements
    smaller than $e$ */

    $W_L \leftarrow 0$ /*stores the left subtree's weight */
    if $root.left \neq null$ then
        $W_L \leftarrow root.left.weight$
    if $e = root.value$ then
        return $W_L$
    $root.weight \leftarrow root.weight + 1$
    if $e < root.value$ then
        if $root.left \neq null$ then
            return InsertAndCount($v, root.left$)
        $root.left \leftarrow$ CreateBinaryNode($e, null, null, 1$)
        return 0
    if $root.right \neq null$ then
        return $W_L$ + 1+InsertAndCount($e, root.right$)
    $root.right \leftarrow$ CreateBinaryNode($e, null, null, 1$)
    return $W_L + 1$
}.
```

ployment sequence for I and T, with a length at most the number of editing commands in $\mathcal{D}$. Thus if the input $\mathcal{D}$ is already a most-efficient deployment, SanitizeIt calculates a deployment that is both safe and most-efficient.

Following the shuffling theorem, SanitizeIt also works in two phases. In phase 1, it inserts rules in T but not I into the running policy R, and selectively moves rules up (i.e., toward a position closer to the beginning) in R, which keeps R as a shuffle of I and a prefix of T. In phase 2, it deletes rules that are in I but not T, and selectively moves rules down in R, which keeps R as a shuffle of T and a prefix of I, and eventually turns R into T.

In phase 1, SanitizeIt maintains three important variables, Δ, i, and t. Δ stores the set of rules that we still need to issue a command for. It is initialized with Δ_0, the set of all rules edited in $\mathcal{D}$, and is strictly decreasing. The two indexes i and t, for I and T respectively, are both initially set to zero and only increase. There are other variables, such as iDM and $lPreds$, that are used to help calculate the position parameters of the editing commands.

SanitizeIt maintains a shuffling invariant in phase 1: R is always a shuffle of I and $T[1, \ldots, (t-1)]$. The algorithm iterates through rules in T by increasing t. While processing rule $T[t]$, it may also increase the index i. Intuitively, the set of rules in $I[1, \ldots, (i-1)]$ and $T[1, \ldots, (t-1)]$, tracked by $\mathcal{U}$, contains all the rules that have completed phase 1 processing so far. In the running policy R, each rule in $\mathcal{U}$ either is already in the correct position, or needs to be deleted or moved down in phase 2. The deletions have to take place in phase 2; otherwise, the shuffling invariant would be violated. The downward moves are postponed until phase 2 to simplify the calculation of their position parameters.

For each $r = T[t]$ in phase 1, there are two cases. In case 1, r is not in I, and r needs to be added. Since r appears in T only once, r has not been added to R yet. Clearly r must be in Δ, otherwise the deployment $\mathcal{D}$ would be incorrect. The algorithm outputs a command to insert r after all commands in $\mathcal{U}$, and removes r from Δ. In case 2, r is in I. There are three subcases, corresponding to the three situations that r does not need to be moved, needs to be moved up, and needs to be moved down. Case 2a: r is not in Δ. This means that r is not touched by any command in $\mathcal{D}$; thus r exists in I and can be left untouched. The position of r in I cannot be smaller than i, otherwise, r needs to be moved *up*. The algorithm keeps increasing i until $I\,[i] = r$, puts every other rule x encountered before r into $\mathcal{U}$, and does bookkeeping for x's position in R to prepare for the deletion or downward move of x in phase 2. Case 2b: r is in Δ, and r is not in $\mathcal{U}$. Then the algorithm moves r up to the position just after all rules in $\mathcal{U}$, and removes r from Δ. Case 2c: r is in Δ as well as $\mathcal{U}$. This means that r needs a downward move, which waits until phase 2. At the end of phase 1, the algorithm goes through the remaining rules in I and saves their positions in R for future deletions in phase 2; these rules are not in T, as otherwise they would have already been added to $\mathcal{U}$.

In phase 2, SanitizeIt maintains the invariant that R is a shuffle of T and a prefix of I. Every r in Δ is either deleted (if r does not occur in T) or moved down to its target position. These deletions and moves start from the end of DM, a subsequence of I with rules to be deleted or moved, and continue backward. In the end, R remains a shuffle of R and an empty prefix of I, which means that R has become T.

One challenge of sanitization is to efficiently calculate the correct position parameters for every `ins`, `mov`, and `del` operation. SanitizeIt does amortized bookkeeping to save the calculation time, and also introduces an AVL-tree [12] based counting algorithm, named BinaryCount, to speed up the calculation. In the appendix, we provide a more detailed explanation of position calculation, and also prove the correctness and safety of SanitizeIt.

Asymptotic analysis shows that SanitizeIt is fast. Phase 1 scans through I and T exactly once. Phase 2 is linear in the size of I. The IndexOf and $\in$ operators used in the algorithm will take constant time, provided that we spend $O(n)$ time to construct the according hash map and hash set first. BinaryCount is $O(d \log d)$ worst case. Overall, SanitizeIt is $O(n + d \log d)$ worst case, which is asymptotically much better than the upper bound for `diff`, or as good as `diff` when d gets close to n. This is important because we do not want the whole deployment to be slowed down by sanitization. Since many real-world policies are as large as 10K rules and 50K rules are not unheard of, an $O(n^2)$ time sanitization algorithm can take hours to finish, even for a small policy change (small d).

6.3 Monotonicity

We prove that any most-efficient deployment calculated by SanitizeIt is also monotonic. Before we start, we give the following observation, which is straightforward based on the definition of a shuffle: let M be a shuffle of I and a prefix of T, or a shuffle of T and a prefix of I. If p hits rule r_I in I, r_T in T, and r_M in M, then r_M must be either r_I or r_T.

Suppose a sanitized most-efficient deployment is not monotonic. Then there must exist a packet p that can get flipped at least three times in the deployment. We first consider the case where neither r_I nor r_T is the hidden default deny-all rule. The running policy is always a shuffle of I and T's prefix (or T and I's prefix) during the deployment, so p can only hit either r_I or r_T. Initially p hits r_I in I. To have three flips, p must hit r_T in some subsequent running policy (flip 1), then r_I in another running policy (flip 2), and r_T in another running policy (flip 3). Before each flip can take place, at least one action (`ins`, `mov`, or `del`) has to take place on either r_I or r_T; otherwise, p will consistently hit r_I or r_T and does not flip. Thus three flips entail at least three actions on r_I or r_T. Since the deployment is still most-efficient and takes at most one action on each individual rule, r_I and r_B can receive at most two actions in total, which is a contradiction. The above argument still holds if r_I and/or r_T are the default deny-all rule. So the sanitized deployment is most-efficient, safe, and monotonic. This gives us the following theorem.

Theorem 3. *For every firewall policy, there is a safe, most-efficient, and monotonic deployment that uses only the* `ins`*,* `del`*, and* `mov` *commands.*

7 Experimental Results

The previous sections have shown that the computational complexity of our approach to safe and efficient deployment is quite reasonable. However, even a $O(n)$ algorithm can be slow in practice if the constant factor is large. This consideration is important for our work because if sanitization and minimization adds significant time to deployment, our approach will not be practical in spite of its modest complexity class.

To address this question, we use four firewall policies, Small, Medium, Large, and Extra Large, with 2,000, 5,000, 10,000, and 25,000 lines of rules, respectively. For each policy size, we have 5 test cases. Test 1 has an edit distance of size 10, meaning that at least 10 command changes are needed to turn the initial policy into the target policy. Real world firewall policy changes are usually incremental and tend to be small, with edit distances measured in dozens of lines of code. (However, a small edit distance does not necessarily result in a small deployment in the real world, either due to lack of type II editing support or due to the fact that the deployment tool does not implement an efficient deployment algorithm.) Edit distances measured in hundreds are considered large changes. Tests 2 and 3 have edit distances of 500 and 1000, respectively. Tests 4 and 5 are even larger, with edit distances equal to 60% and 90% of the initial policy size, respectively. Edit distances as large as in tests 4 and 5 are rare in the real world; we include them to test the performance of our algorithms under extreme conditions.

We use the `diff` algorithm in [29] to calculate the most-efficient deployment, then run SanitizeIt to make it safe. All algorithms are implemented in Java, and the experiments are run on an HP XW4200 desktop with 2.8 GHZ CPU and 2 GB memory. For each test case, we record the time (in seconds) spent in `diff`, SanitizeIt and downloading. The download time is the span between starting to send the first command to the firewall and finishing sending the last command. We use a firewall simulator that is configured to match the performance of a PIX 525 firewall and connect to it over a 10Mb ethernet link. We run each test case 50 times and then record the average for `diff` and SanitizeIt.

Table 1 gives the results for tests 1-5 on the four policies. The results show that the time spent in `diff` is small for common policy changes (test cases 1-3), ranging from 5 to 203 microseconds. The `diff` time becomes more noticeable in the extreme cases (tests 4-5), going as high as 26 seconds for a 90% edit distance on the extra large policy; the corresponding download time is 306 seconds and overshadows the `diff` time. The sanitization time is negligible in almost all the test cases. Even in test 5 on the extra large policy, sanitization takes about 1 second. In summary, the sanitization time is negligible compared to the `diff` time, while the `diff` time is negligible compared to the download time. This suggests that for practical policy sizes and changes, our proposed sanitization algorithm can makes a deployment safe while adding almost no delay; and that the `diff` algorithm has a negligible cost compared to the download time and should always be used to reduce the deployment size.

8 Discussion

Besides the type I and II editing languages that we define, there are other flavors on the market. For example, the editing languages used in Nokia IPSO 3.8 [9] and EdgeForce 4.5 [10] stand between type I and II, by supporting `ins` and `del` but not `mov`. Similar to the proof for Theorem 2, we argue that not all firewall policies can be deployed safely by inserting and deleting only the rules that are in the initial or target policies. However, we can apply `diff` and SanitizeIt to calculate a most-efficient and safe type II deployment first, as if `mov` is supported. Then we replace every `mov` with the according `del` and `ins`, so that the time that the deployment is unsafe remains only within the small number of intervals between a `del` and the

Test	Small (2,000)			Medium (5,000)			Large (10,000)			Extra Large (25,000)		
	diff	sani	depl	diff	sani	depl	diff	sani	depl	diff	sani	depl
Test 1	.005	.007	2.160	.005	.018	2.10	.013	.031	2.260	.073	.169	2.250
Test 2	.010	.002	10.500	.010	.018	10.660	.013	.036	10.680	.161	.122	11.280
Test 3	.023	.015	20.680	.026	.023	21.300	.031	.039	21.400	.203	.122	21.520
Test 4	.025	.015	24.540	.179	.026	61.300	1.255	.127	122.420	11.955	.627	306.560
Test 5	.062	.008	35.600	.356	.031	89.000	4.158	.234	181.680	25.952	1.031	454.260

Table 1. Results of Experiments (in seconds)

subsequent `ins` on the same commands, which makes the deployment almost safe.

Some editing languages are more powerful than type II. For example, iptables [6] supports inserting and deleting more than one rule at a time, and replacing a rule with one or more rules. These editing languages can still be used to safely deploy any policy, and our sanitization algorithm still works. The deployment sequence calculated by the `diff` algorithm, however, is no longer most-efficient, since the editing commands can now operate on more than one rule. Considering that the editing distances in real-world deployments are small, the output from `diff` and SanitizeIt is already safe and reasonably efficient. We can further optimize it by running a greedy algorithm to combine the consecutive commands of the same type into one command.

The safety problems that we address can also be solved by letting firewalls support transactional semantics for policy updates. For example, if a firewall buffers incoming policy changes and switches to the target policy in a single atomic action, then no traffic anomalies could occur. The manual commit mode in FWSM [2] allows policies to be updated in an almost transactional fashion. Firewalls that support policy names can emulate a transaction by switching to a policy with a different name, at the cost of building the target policy from scratch and losing the original policy name. Full native transaction support, however, is still rare in today's firewalls. Firewalls started as command-line-interface low-end appliances and were not designed to be management-friendly in the first place. Further, the need for transactions is not apparent to most people, because the management tools already provide network administrators with the feel of transactions. These tools allow administrators to make multiple changes through the GUI, followed by a single mouse click to "commit" (deploy) the changes. However, as we have shown, the resulting deployment is not necessarily atomic or safe. This problem is particularly acute with type I policy editing languages, which we have proved unable to guarantee safe deployments without complex workarounds, such as introducing semantically equivalent rules that occur in neither the initial or target policies. These workarounds, however, do not change the fact that type I editing commands are too limited for effective support of firewall policies with order-sensitive semantics.

One can think of our techniques for generating a safe deployment as providing a poor-man's transaction facility, as the two approaches offer similar safety guarantees. However, we believe that the runtime costs using our techniques are lower than the overhead for a runtime transaction facility would be. In particular, critical policy patches and large deployments can take effect incrementally, rather than waiting for the commit point of a transaction. Since smart attackers may be lying in wait to exploit a vulnerability during deployment, every microsecond counts. Compared to transactions, our approach to safe deployment also has the additional advantage of minimizing the code base at the firewall, which reduces the potential for bugs and security holes and also helps throughput. On the other hand, transactional approaches can also make use of our approach to most-efficient deployments, to improve performance.

Although our discussion uses static packet inspection, we believe our conclusions also apply to stateful packet inspection (SPI) [32] firewalls. An SPI firewall inspects traffic at the network and transportation layers by monitoring the state of each TCP connection or pseudo UDP connection. It uses the initial packets of a connection to establish a session, and decides whether to accept a future packet based on which session it belongs to. To extend our results to SPI, we would start by changing the discussion of *accept* and *deny* from the granularity of *packets* to *sessions*, and adjusting the safety definitions in section 4 accordingly. For example, we would define policy A to be permission-safe w.r.t. policies B and C iff every session that A permits is also permitted by B or C. We leave a more rigorous analysis of how to apply our results to SPI for future work.

9 Related Work

While we find no research on firewall policy deployment safety, there is quite a lot of research literature on firewall/VPN policy conflict detection (e.g,, [13, 20, 15, 14, 38]). This work has formalized types of conflicts in firewall policies and provided a variety of fast and efficient algorithms to detect them. Going one step further, researchers have proposed optimization algorithms to generate more concise and efficient firewall policies, using techniques like address/port combination and performance tuning based on traffic analysis [31, 22, 11]. All this research is orthogonal to our work, and can be applied to improve the quality of firewall policies before they are deployed.

Firewall policy engineering and specification models are also getting a lot of attention. For example, Firmato allows policy specification based on global entity-relationship information [16]; Bellovin et al. [17, 25] propose a distributed firewall model that allows centralized policy specification, while reducing or eliminating topology dependencies; the STRONGMAN architecture supports scalable policy composition which nonetheless allows autonomy within the constraints of a global policy [26]. Gouda et al. propose firewall design diagrams to support consistent and compact policy generation [21]. Their results complement our work by helping create effective policy specification tools, which leads to the need for proper policy deployment.

Researchers have extensively studied the problem of determining the differences between two sequences of symbols [35, 34, 23, 24, 27, 30, 29, 33]. Many algorithms have been proposed, though with no all-time winners. The best known results for permutation sequences are $O(d\,n)$ [29], when the difference d between the two sequences is small, and $O(n \log n)$ [24], when d is close to n. Our work builds on these results, and concentrates on achieving safety without sacrificing efficiency.

10 Conclusions

In this paper, we have shown how unsophisticated approaches to firewall policy deployment can temporarily open a network to unwanted traffic and prohibit desired traffic. Up to this day, unsafe deployment approaches are still being practiced by commercial firewall management tools. We have provided the formal definition and theoretical analysis of the safe deployment problem, proposed ways to remedy it based on the shuffling theorem, and measured their effectiveness in an implemented system. The bottom line is that vendors' state-of-the-art policy deployment products can and should be modified to close this security loophole, and our techniques can be used to accomplish this goal without compromising efficiency.

We have shown that type I policy editing languages are very inefficient in deploying order-sensitive policies. To make things worse, they cannot safely deploy an arbitrary firewall policy without complex or expensive workarounds. We have shown that the situation is more hopeful with type II policy edit languages, for which one can always efficiently find a minimal set of editing commands that will safely update the policy. Our experimental results showed that our approach to finding a safe and minimal deployment adds only a user-imperceptible overhead to the policy update time for typical small policy changes, when a type II editing language is used. Even large changes added only 0.2% or less to the total policy deployment time in our experiments. Thus safety can be achived without compromising deployment efficiency, which is important for critical policy patches. Our approach can also be viewed as a light-weight, low-cost implementation of a policy commit facility that provides transactional safety guarantees.

Acknowledgements

This research was supported by NSF under grants IIS-0331707, CNS-0325951, CNS-0524695, and CNS-0524516 and by ONR under grant N00014-02-1-0715. Zhang is also associated with Cisco Systems, Inc. We thank Hari Shankar, Rui Jiang, and the anonymous reviewers for their valuable comments and suggestions.

References

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

[2] Cisco Firewall Service Module. http://www.cisco.com/en/US/products/hw/modules/ps2706/ps4452/index.html.

[3] Cisco PIX Firewall. http://www.cisco.com/en/US/products/hw/vpndevc/ps2030.

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

[5] Enterasys Matrix X Core Router . http://www.enterasys.com/products/routing/x/.

[6] Iptables Howto. http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO-7.html.

[7] Juniper Networks JUNOSe Operating System. http://www.juniper.net/techpubs/software/erx.

[8] Juniper Networks NetScreen-Security Manager. http://www.juniper.net/products/integrated/ns_sm.html.

[9] Nokia IPSO. http://europe.nokia.com/A4153092.

[10] ServGate EdgeForce. http://www.servgate.com/pdf/infosec_article.pdf.

[11] S. Acharya, J. Wang, Z. Ge, T. F. Znati, and A. Greenberg. Traffic-aware firewall optimization strategies. In *IEEE International Conference on Communications (ICC)*, Istanbul, Turkey, June 2006.

[12] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet Math. Doclady*, 3:1259–1263, 1962.

[13] H. Adiseshu, S. Suri, and G. M. Parulkar. Detecting and resolving packet filter conflicts. In *INFOCOM*, pages 1203–1212, 2000.

[14] E. S. Al-Shaer and H. H. Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *Integrated Network Management*, pages 17–30, 2003.

[15] F. Baboescu and G. Varghese. Fast and scalable conflict detection for packet classifiers. In *IEEE International Conference on Network Protocols (ICNP)*, pages 270–279, 2002.

[16] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, 1999.

[17] S. M. Bellovin. Distributed firewalls. *;login: magazine, special issue on security*, November 1999.

[18] M. Englund. Securing systems with host-based firewalls. *Sun BluePrints OnLine*, September 2001.

[19] A. O. Freier, P. Karlton, and P. C. Kocher. Secure socket layer 3.0, IETF draft, November 1996.

[20] 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 Workshop*, pages 39–56, 2001.

[21] M. G. Gouda and X.-Y. A. Liu. Firewall design: consistency, completeness and compactness. In *IEEE Int. Conf. Distributed Computing Systems (ICDCS)*, 2004.

[22] H. Hamed and E. Al-Shaer. Dynamic rule-ordering optimization for high-speed firewall filtering. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, March 2006.

[23] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

[24] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[25] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[26] A. Keromytis. Strongman: A scalable solution to trust management in networks. PhD thesis, University of Pennsylvania, Philadelphia, 2001.

[27] W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[28] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm. Technical report, CAIDA, ICSI, Silicon Defense, UC Berkeley EECS and UC San Diego CSE, 2003.

[29] E. W. Myers. An $O(nd)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[30] N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Inf.*, 18:171–179, 1982.

[31] J. Qian, S. Hinrichs, and K. Nahrstedt. ACLA: A framework for access control list (ACL) analysis and optimization. In *Communications and Multimedia Security*, 2001.

[32] C. Roeckl. Stateful inspection firewalls. *Juniper Networks White Paper*, 2004.

[33] I. Simon. Sequence comparison: some theory and some practice. In *Electronic Dictionaries and Automata in Computational Linguistics*, number 377, Saint Pierre d'Oléron, France, 1987.

[34] J. D. Ullman, A. V. Aho, and D. S. Hirschberg. Bounds on the complexity of the longest common subsequence problem. *J. ACM*, 23(1):1–12, 1976.

[35] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[36] A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004.

[37] T. Ylonen. SSH - secure login connections over the internet. In *Proceedings of the 6th USENIX Security Symposium*, Berkeley, CA, 1996.

[38] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *IEEE Symposium on Security and Privacy (S&P'06)*, Oakland, CA, 2006.

Appendix

A. Calculation of Position Parameters in SanitizeIt

$\mathcal{U}$'s size $|\mathcal{U}|$ gives the target position in R for any command to be inserted or moved at line 29 and 47. For the upward `mov` in line 47, r's current position in R is the sum of its position in I, the number of commands in DM that come after r in I but before r in T (stored in $rPreds$), and the number of inserted commands. Similarly, the target position of $DM[j]$ in line 64 is its position in T plus the number of commands in DM that come before $DM[j]$ in I but need to be either removed from R or moved below $DM[j]$. $lPreds[i]$ stores the number of rules that show up earlier than $DM[i]$ in both DM and T. $lPreds[i]$ stores the number of rules that occur later than $DM[i]$ in DM but earlier than $DM[i]$ in T. When $DM[i]$ is not in T, the value of $lPreds[i]$ and $rPreds[i]$ does not matter. Every rule $DM\,[i]$ is translated into an integer value and stored in $iDM\,[i]$. If $DM\,[i]$ is in T, $iDM\,[i]$ stores $DM\,[i]$'s index in T; otherwise, $iDM\,[i]$ stores i plus the size of T.

The helper function BINARYCOUNT calculates *lPreds* and *rPreds* based on *iDM*. It takes an array A, and counts for every $A[i]$ the number of elements $A[i]$ to the left of and smaller than $A[i]$, i.e., $j < i$ and $A[j] < A[i]$. The counting utilizes a binary search tree with each node m also storing the weight (number of nodes) of the subtree whose root is m. Considering that the worst-case search time in a unbalanced binary tree is linear, we can keep the binary tree balanced as an AVL tree by doing AVL rebalancing after each insertion to guarantee logarithmic search time.

B. Correctness and Safety of SANITIZEIT

Proof. we first argue that the following invariants hold at the beginning of every iteration of the for loop in phase 1: (a) $\mathcal{U}$ equals the set of the first $|\mathcal{U}|$ rules of the running policy R, and also equals the set of all rules in $T[1, \ldots, (t-1)]$ and $I[1, \ldots, (i-1)]$; (b) the running policy R is a shuffle of I and $T[1, \ldots, (t-1)]$; (c) the rules in the set $G = \mathcal{U} - \Delta$ have a *correct* relative order in R, meaning the relative order is the same as in T. We prove this by mathematical induction on t.

The invariants are trivially true when $t = 1$. Suppose they hold at the beginning of some iteration where $t = k$, $\mathcal{U}$'s value is $\mathcal{U}_k$, and $\mathcal{U}$'s size is u_k. Continuing on with this iteration, suppose that rule $r = T[t]$ is in U_k already, or is inserted or moved up to position $u_k + 1$ in R, or gets added to $\mathcal{U}$ along with all rules between $I[i]$ and r in I. R becomes a shuffle of I and the first t rules. So all the invariants hold at this point.

Invariants (a) and (b) are still true at the beginning of the next iteration. If $r \in \mathcal{U}_k$, then G stays unchanged, and so does the relative order of G's rules in R. If $r \notin \mathcal{U}_k$, the only possibility for (c) not to hold would be if there exists $r_v \in \mathcal{U}_k$ that is not in $\mathcal{U}_k - \Delta_k$, stays before r in R at the end of this iteration, but comes after R in T. Then r_v does not belong to Δ_0. The only way such an r_v can get into $\mathcal{U}$ is at line 36, which is followed by rule r_w (added in line 41) that is not in Δ_0 either. Since r_v comes after r in T and r_w comes before r in T, the relative order of r_v and r_w is different in I and T; but neither r_v nor r_w belongs to Δ_0, which contradicts the precondition that $\mathcal{D}$ is a correct deployment. Thus all the invariants hold at the beginning of every iteration of phase 1's for loop.

Invariant (c) also holds in the beginning of every iteration of the `while` loops at lines 50 and 34. We can use an argument similar to the above to prove that every rule added to $\mathcal{U}$ in the loop either belongs to Δ or its relative order with respect to all rules in $\mathcal{U} - \Delta$ is correct already. At the end of phase 1, $\mathcal{U}$ is the set of all rules in R, R becomes a shuffle of I and T, and the relative order of all rules in $R - \Delta$ is correct.

Second, we argue that two invariants hold at the beginning of every iteration of the for loop (line 58) of phase 2: (d) the relative order of the rules in $R - \Delta$ is correct; and (e) R is a shuffle of T and $I[1, \ldots, i_j]$, where i_j is the index of $DM[j]$ in I. We prove this by an induction on the number of iterations. In the first iteration, $j = \text{SizeOf}(DM)$. From the proof for phase 1 above, we know (d) is true and R is a shuffle of I and T. Since every r in I after $I[i_j]$ is not in Δ, all such r's are contained in $R - \Delta$, whose rules' relative order is already correct. Thus (e) also holds.

Suppose (d) and (e) hold for some iteration where $j = k$. Continuing on with this iteration, if $DM[j] \notin \Delta$, then R and Δ stay unchanged, so (d) holds. Invariant (e) also holds based on the induction hypothesis and a similar argument to that for the first iteration with I changed to the first i_j rules of I. If $DM[j]$ is deleted (line 61), it gets removed from both R and Δ, so (d) and (e) still hold. If $DM[j]$ is moved to a new position (line 64), $DM[j]$'s relative order with respect to all rules in $R - \Delta$ is correct, based on the position calculation explained earlier. Thus (d) and (e) still hold at the beginning of the next iteration.

When phase 2 ends, Δ becomes empty, R contains all rules in T, all rules in I but not T have been deleted from R, and $R - \Delta$ is in correct order. So R equals T, which means SANITIZEIT is correct. Since R is a shuffle of I and a prefix of T in phase 1, and a shuffle of T and a prefix of I in phase 2, we conclude that SANITIZEIT calculates a safe deployment, by the shuffling theorem. □