On Optimizing Firewall Performance in Dynamic Networks by Invoking a Novel *Swapping Window*-based Paradigm

Ratish Mohan[a], Anis Yazidi[a], Boning Feng[a], John Oommen[b,*]

[a]*Department of Information Technology, University College of Oslo and Akershus, Oslo, Norway.*
[b]*School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6.*

Abstract

Designing and implementing efficient firewall strategies in the age of the Internet of Things (IoT) is far from trivial. This is because, as time proceeds, an increasing number of devices will be connected, accessed and controlled on the Internet. Additionally, an ever-increasingly amount of sensitive information will be stored on various networks. A good and efficient firewall strategy will attempt to secure this information, and to also manage the large amount of inevitable network traffic that these devices create. The goal of this paper is to propose a framework for designing optimized firewalls for the IoT.

This paper deals with two fundamental challenges/problems encountered in such firewalls. The first problem is associated with the so-called "Rule Matching" (RM) time problem. In this regard, we propose a simple condition for performing the swapping of the firewall's rules, and by satisfying this con-

*Corresponding author
Email address: oommen@scs.carleton.ca (John Oommen)

The fourth author is grateful for the partial support provided by NSERC, the Natural Sciences and Engineering Research Council of Canada. A preliminary version of this paper, with a brief sketch of the results, was presented at *ACM ICCNS '16, the 2016 ACM International Conference on Communication and Network Security*, in Singapore, in November 2016.

Fourth author's status: *Chancellor's Professor*, *Fellow of the IEEE*, and *Fellow of the IAPR*. This author can be contacted at: School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6. The author is also an Adjunct Professor with Agder University College in Grimstad, Norway. E-mail: oommen@scs.carleton.ca.

dition, we can guarantee that apart from preserving the firewall's consistency *and* integrity, we can also ensure a greedy reduction in the matching time. It turns out that though our proposed novel solution is relatively simple, it can be perceived to be a generalization of the algorithm proposed by Fulp [1]. However, as opposed to Fulp's solution, our swapping condition considers rules that are not necessarily consecutive. It rather invokes a novel concept that we refer to as the "swapping window".

The second contribution of our paper is a novel "batch"-based traffic estimator that provides network statistics to the firewall placement optimizer. The traffic estimator is a subtle but modified batch-based embodiment of the Stochastic Learning Weak Estimator (SLWE) proposed by Oommen and Rueda [2].

The paper contains the formal properties of this estimator. Further, by performing a rigorous suite of experiments, we demonstrate that both algorithms are capable of optimizing the constraints imposed for obtaining an efficient firewall.

Keywords: Firewall Optimization, Matching time, Weak Estimators, Learning Automata, Non-Stationary Environments, Batch Update.

---

1. Introduction

The inter-connectivity, convenience and the all-prevalent digital services offered by the Internet, come with a steep price. As our society becomes more dependent on the Internet, the requirement to secure the information stored on these devices and services is more stringent and demanding. To secure the information, the users and systems' administrators have to be even more security-conscious.

The filed of computer security is extensive. It encompasses the security of the physical machines as well as the information stored on them. However, in the context of the Internet, one has to be additionally concerned with *network*-related aspects of security. Such a "specialized" form of security is mandatory

especially because an increasing number of devices are connected to various networks, and primarily to the Internet [3]. Network security deals with the security aspects of data and communication within a/multiple network(s), and it spans many different concepts such as authentication, access policies, intrusion detection, intrusion prevention and honeypots/honeynets.

A first line of defence in network security is to use a firewall in order to enforce access policies. A firewall, in essence, is a system architecture program whose objective is to filter the incoming and outgoing packet traffic on a host or in a network. The task of accepting or denying access to the network is enforced by matching the header information of each data packet against a predefined set of rules, referred to as the "firewall policy". Each rule has an action associated with it, for example, to either deny or accept access, and this action is what decides whether a packet is dropped or not.

A study of many Internet and private traces shows that the major portion of any network's traffic matches only a small subset of firewall rules. This, in turn, implies that the frequency distribution for some of the traffic properties appears to be highly skewed [4]. Furthermore, when performing packet filtering, each rule in a firewall policy will usually be checked in a sequential order. Consequently, as the firewall policy increases in size, as any rule is often combined with a matching rule of a higher order, the overhead associated with the task of filtering the firewall, will become increasingly costly.

The reader will easily see that this rule matching phase can easily become a bottleneck in a high speed network when it is under attack or when it encounters a heavy network load [1, 5]. Furthermore, it is well known that the computing power of hosts, the transmission speeds of packets and the complexity of networks, continue to increase. To keep abreast with these increasingly-demanding environments, firewalls must be able to "proportionately" adapt to changes by processing packets at increasingly higher speeds [6, 7]. Thus, it is desirable that a firewall monitoring system processes a lesser number of packet matches in order to reduce the potentially exorbitant filtering overhead as well as the overall packet matching time [4].

A natural inference of the above assertions is the following: In order reduce the number of packet matches that have to be processed, and to ensure that a firewall is able to process packets at an adequate speed, it is crucial for the firewall's architecture to have an optimized *ordering* for the appropriate rules. This can be achieved by ensuring that the rule ordering is such that the rules that are matched most often, appear at the top (front) of the list of rules. This will reduce the amount of time used to process a packet by reducing the number of required packet matches, and consequently reducing the packet filtering overhead. Additionally, it will also have the effect of improving the network's throughput because a packet will spend less time being processed.

Although the problem is easily stated, the task of finding the optimal rule order is NP-hard because of inter-rule dependencies. Our goal is thus to find a heuristic algorithm to find a near-optimal rule order.

The complexity of the problem is accentuated by the fact that traffic patterns in networks are not static. This implies that since the patterns are dynamic and possibly time varying, one cannot learn the statistics of the traffic patterns using traditional estimation methods. Rather, one has to devise estimation (or learning) strategies that are rather effective for non-stationary environments. This is the task we undertake!

1.1. Problem Statement and Contributions

Put in a nutshell, this paper deals with *dynamic networks*, i.e., those that are characterized by being "under constant change and activity". Essentially, a dynamic network is one in which the state of packet traffic is time-varying and non-stationary. This implies that the packet traffic fluctuates in such a way that no single type of traffic is dominant for an extended period of time. Our goal is to find a solution to the problem of optimizing the *performance* of a firewall in such dynamic networks. In order to achieve this, we attempt to answer the following questions:

- *How can we optimize the order of the firewall rules in order to minimize the Rule Matching (RM) operations invoked?*

- *How can we learn and use the dynamic network traffic statistics to further optimize the firewall?*

Of course, to achieve the above, we shall examine the traffic patterns statistically, combine the inferences with the workings of a RM algorithm. Thus, the major contributions of this paper are:

- We present an efficient and yet simple mechanism for optimizing the order of the rules in the firewall by using a novel concept that is referred to as the **Swapping Window**. The **Swapping Window** is a straightforward strategy by which one can infer whether it is beneficial to swap the *order* of two rules in a RM algorithm by considering their matching probabilities, and simultaneously guaranteeing that no inconsistencies are introduced in the firewall. We submit that, without loss of generality, our solution is a mapped efficient solution to the ***Single Machine Job Scheduling*** (SMJS) Problem [8] – since our problem can be shown to be a specific instantiation of the latter.

- We present a novel adaptive algorithm for estimating the statistics of multinomial observations appearing in a batch mode[4]. The algorithm is able to deal with non-stationary environments and is an extension of the Stochastic Learning Weak Estimation (SLWE) work by Oommen and Rueda [2], which is, in and of itself, suitably adapted for high speed networks. The observations that the estimation scheme receives are, in our case, the different matched rules within a time interval when they are examined as a "batched" data stream and not as sequential entities.

- We combine both the above-mentioned contributions (the rule ordering algorithm augmented with the estimation scheme) into a single algorithm so as to achieve a holistic approach for optimizing the firewall's performance.

[4]The batched-mode version of Oommen-Rueda's SLWE is a contribution in its own right to the field of estimation in non-stationary environments.

1.2. Organization of the Paper

After having introduced and motivated the problem in Section 1, we proceed to review the related state-of-the-art in Section 2. In this section, we introduce the fundamental concepts and notations required for this paper to be a self-contained document. As well, in this section, as we will review the relevant related work. In Section 3, we present our solution composed of two main components: Rule Re-ordering (RR) and traffic estimation.

In Section 4, we present some theoretical results that demonstrate the validity of the algorithms proposed in Section 3 for both rule ordering and estimation. Section 5 contains simulation results demonstrating the power of the scheme in stationary and dynamic environments. The experiments done for dynamic environments were based on a realistic test-bed, while those done for stationary environments were done using a simulated set-up (without requiring a test-bed). The paper also includes a thorough discussion of the results. Section 6 concludes the paper.

2. State-of-the-Art

This section outlines the current state-of-the-art when it concerns firewall optimization. It also introduces and explains several key concepts, technologies and applications that we will use in this paper.

2.1. Firewalls

Based on the article *Benchmarking Terminology for Firewall Performance (RFC 2647)* in [9], a firewall is defined as "a device or group of devices that enforces an access control policy between networks". Firewalls are thus, devices or programs that control the flow of network traffic between networks or hosts [10].

2.1.1. Rules and Packets

To understand how a firewall operates, it is necessary to understand the relationship between the access control rules and the packets that they govern. We now present a formal explanation of the relevant terms.

A firewall rule, r, is defined as an n-tuple of ordered fields:

$$r = (r[1], \ldots, r[n]), \text{ for } n > 1.$$

Although the upper bound of n is network specific, for an Internet firewall, it is usually set to five and comprises of the following fields: **Protocol**, **Source Address**, **Source Port**, **Destination Address**, and **Destination Port** [4, 11, 12]. Each rule has an action field associated with it, and the value of the field decides the actions that the firewall will take when a match is found. Table 1 outlines the values of the action fields.

Action	Effect
Accept	Forward the packet
Deny	Drop the packet
Log, Accept	Log and forward the packet
Log, Deny	Log and drop the packet

Table 1: The action field values of a firewall and its effects.

The **Protocol** field specifies a protocol as documented in the IP packet header's protocol field, as stated in *Internet Protocol (RFC 791)*. For an Internet firewall, this would be either TCP, UDP or ICMP. However, it could also contain a wild card value (*), in which case it will match any protocol [10, 11, 13, 14]. The **Address** and **Port** fields specify the source and destination IP addresses, and the source and destination port numbers of incoming and outgoing packets. Both of these fields can be configured to represent a range of values or a set of values, rather than only a single value. Tables 2 and 3 outline these types of configurations for the port and address fields respectively.

Notation	Example	Explanation
Wild Card	* or *any*	Port range 0 - 65535
Range	90-94	The given range of port numbers
Single	90	A single given port number

Table 2: Notation for the port field in a firewall rule.

Notation	Example	Explanation
CIDR	192.0.2.0/24	Address range 192.0.2.0 - 192.0.2.255
Wild Card	192.0.*	Address range 192.0.0.0 - 192.0.255.255
Range	192.0.2.2 - 192.0.2.150	The given range of addresses
Single	192.0.2.2	A single given IP address

Table 3: Notation for the address field in a firewall rule.

A data packet, p, is defined as an n-tuple of ordered parameters:

$$p = (p[1], \ldots, p[n]), \text{ for } n > 1.$$

The upper bound of n is limited by the fields defined in the *Internet Protocol (RFC 791)* [13]. That being said, not all fields in the *IP* header are of importance for a firewall. Thus, a data packet, as seen by a firewall, is comprised mainly of the following fields [4, 11, 12]:

1. Protocol
2. Source Address IP
3. Source PORT
4. Destination Address IP
5. Destination PORT

These fields correspond to the fields that a firewall rule is comprised of and that it processes.

2.1.2. Firewall Policies

A **firewall security policy** defines how an organization's firewall handles inbound and outbound network traffic based on its security policies. Prior to establishing these security policies, generally speaking [10], an organization should conduct a rigorous risk analysis in order to discover what types of traffic passes through its networks at all times. Based on such an analysis, the administrators should determine how they can secure it. Indeed, a firewall security policy is the result of implementing such an analysis .

Examples of policy requirements include accepting only necessary IP protocols to pass [13], authorized source and destination IP addresses, authorized TCP and UDP ports to be used, and certain ICMP types and codes to be used [10]. Generally speaking, all inbound and outbound traffic that is not expressly permitted by the firewall policy should be blocked because such traffic is not needed by the organization. This practice can also have the additional benefit of reducing the risk of attacks and decreasing the volume of traffic carried into/through the organization's networks [10].

To specify a formal definition for a firewall policy, let:

$$R = \{r_1, r_2, r_3, \ldots, r_N\}, \text{ for } N > 1$$

be the set of ordered firewall rules comprising a policy.

Such a firewall policy is considered to be comprehensive if any packet, p, has a match in R. In practice, this is achieved by implementing a ***Default Rule*** [4, 10] which serves the purpose of being a *catch-all* rule. It is usually added at the end of a policy and is designed such that it will simply discard any packet that has not matched any of the above rules. Table 4 shows an implementation of one such comprehensive firewall security policy.

2.1.3. Packet Matching

In many implementations of firewalls, the rules are stored internally as linked lists [12]. A firewall will, thus, generally speaking, sequentially compare a packet with a rule. In order for a rule, r_i, to match a packet, p, the parameters of the packet header must be a subset of all the permitted corresponding fields in the rule. Thus, if

$$r_i[l], \text{ for } l = 1 \ldots n, \text{ and}$$

$$p[l], \text{ for } l = 1 \ldots n$$

represent the ordered fields of the rule r_i and the ordered parameters of the packet header for packet p respectively, the match between rule r_i and the

		Source		Destination			
No.	Proto.	IP	PORT	IP	PORT	Action	Prob.
1	UDP	190.1.*	*	*	90	accept	0.05555555555
2	UDP	190.1.1.*	*	*	90-94	deny	0.05555555555
3	UDP	190.1.2.*	*	*	*	deny	0.05555555555
4	UDP	190.1.1.2	*	*	94	accept	0.05555555555
5	TCP	190.1.*	*	*	90	accept	0.05555555555
6	TCP	190.1.1.*	*	*	88	deny	0.05555555555
7	TCP	190.1.1.2	*	*	88-94	deny	0.05555555555
8	TCP	190.1.2.*	*	*	*	accept	0.05555555555
9	TCP	*	*	161.120.33.41	25	accept	0.05555555555
10	TCP	140.192.37.30	*	*	21	deny	0.05555555555
11	TCP	*	*	161.120.33.*	21	deny	0.05555555555
12	TCP	140.192.37.*	*	*	21	accept	0.05555555555
13	TCP	*	*	161.120.33.*	22	accept	0.05555555555
14	TCP	140.192.37.*	*	*	80	deny	0.05555555555
15	TCP	*	*	161.120.33.40	80	accept	0.05555555555
16	TCP	*	*	161.120.33.43	53	accept	0.05555555555
17	UDP	*	*	161.120.33.43	53	accept	0.05555555555
18	*	*	*	*	*	deny	0.05555555555

Table 4: An example of a firewall security policy configuration with equal initial probabilities.

packet p can be denoted as:

$$p \Rightarrow r_i \iff \forall\, l,\ p[l] \subset r[l],\ \text{ for } l = 1 \ldots n.$$

Informally speaking, p matches r_i **if** and **only if** all the parameters of p are in a subset of the respective fields of r_i. Because each parameter $p[l_i]$ must match the corresponding field $r_i[l_j]$, the order of fields in a rule is important to the rule matching process. Thus, a packet p can match multiple rules in a firewall, R. The matching policy of the firewall decides which rule render the packet to be considered to be "matching".

There are, generally, three common matching policies used, namely, the *Best Match*, *Last Match* and *First Match* policies [15] listed below:

- **Best Match**: A packet is compared against all $r_i \in R$. The rule that matches the closest with the packet is selected and its action is consequently executed.
- **Last Match**: A packet is sequentially compared to each rule $r_i \in R$. The last rule that matches, $p \Rightarrow r_i \in R$, is selected and its action is consequently executed.
- **First Match**: A packet is sequentially compared to each rule $r_i \in R$. The First rule that matches, $p \Rightarrow r_i \in R$, is selected and its action is consequently executed.

The **Best Match** and potentially, the **Last Match** schemes increase the packet matching time. Consequently, in this paper, we assume a **First Match** matching policy.

2.2. Firewall Modelling and Policy Anomalies

This section outlines how a firewall is modeled and what policy anomalies are.

2.2.1. Rule Intersection

As stated in Section 2.1.1, any parameter of a rule can contain a range of values. A consequence of this is that multiple rules can intersect. Two rules, r_i and r_j, intersect if a comparison of their ordered parameters yields a nonempty set. More formally, this is represented as below:

$$r_i \cap r_j \neq \emptyset \iff \exists l,\ \ r_i[l] \cap r_j[l] \neq \emptyset,\ \ \text{for } l = 1 \ldots n.$$

Consider Table 5 which displays some examples of this.

		Source		Destination		
No.	Proto.	IP	PORT	IP	PORT	Action
1	UDP	190.1.*	*	*	90	accept
2	UDP	190.1.1.*	*	*	90-94	deny
3	TCP	140.192.37.30	*	161.120.33.40	80	deny

Table 5: Intersecting and non-intersecting rules.

Rules 1 and 2 intersect because Rule 2 describes a subnet in Rule 1. Further, the port in Rule 1 is a subset of the ports in Rule 2. In other words, the intersection of Rules 1 and 2 yields the following non-empty set:

$$\{190.1.1.0 - 190.1.1.255,\ 0 - 65535,\ 0.0.0.0 - 255.255.255.255,\ 90\}.$$

On the other hand, Rule 3 is completely separate from the other two rules and does not intersect with them. The existence of rule intersections in a firewall policy can limit the size of the set of valid rule orderings and be the cause of anomalies in the policy.

2.2.2. Precedence Relationships

As described in Section 2.1.2, a firewall policy is defined as an ordered set of firewall rules, R, and each packet, p, will be *sequentially* compared to a rule, r, in a list-like manner. Furthermore, a packet can match multiple rules, and

this is evident by the different types of matching policies that a firewall has. This means that the *order* in which the rules are maintained and processed is important, and should be preserved. If the order is not preserved when the rules are re-ordered (for example, if they are, instead, reversed), a packet might match the wrong rule and violate the integrity of the policy.

The integrity of a policy is defined as the original intent of the policy. To formalize this, let R be the original firewall security policy, and let R' be a re-ordering of all the rules in R. In that case, in order for the system to maintain the integrity of the firewall policy R in R', a packet, p, must match the same rule and have the same action executed in R' as it would have done in R.

It is important for the reader to understand that a firewall is not merely comprised of disjoint rules. More often than not, there will be ***Precedence Relationships*** between many of the rules. A precedence relationship is a connection between two or more rules where a rule must appear before another in order for the integrity of the policy to be kept intact.

In order to accurately model a firewall policy with relationships, one uses a Directed Acyclic Graph, *DAG* $G = (R, E)$, rather than a list. In such a model, R represents the set of firewall rules in a policy and E, the set of directed edges, represents the set of precedence relationships between the rules. Representing a firewall policy using a *DAG* has distinct advantages over a list representation. They are:

- The foremost advantage of a *DAG* representation is that it renders the task of modeling precedence relationships in a firewall much easier. This is because each node in the graph will represent a rule and each directed edge between two nodes will represent a precedence relationship. Indeed, an edge between rules is determined by finding the intersection between the rules in the firewall, R.

- Secondly, the problem of optimizing the rule order of a firewall has been shown to be comparable to that of the *single machine job scheduling problem* subject to certain precedence constraints. Further, since a *DAG* model can

be used to represent the scheduling problem, it would be appropriate to use a similar model in order to a model a firewall policy.

For clarification purposes, the precedence relationships specified in Table 4 are shown in Figure 1. As one can see, the figure displays the *DAG*, created (or rather, implied) by the relationships. In the interest of completeness, we now explain, in greater detail, two of the best-reported solutions for this problem.

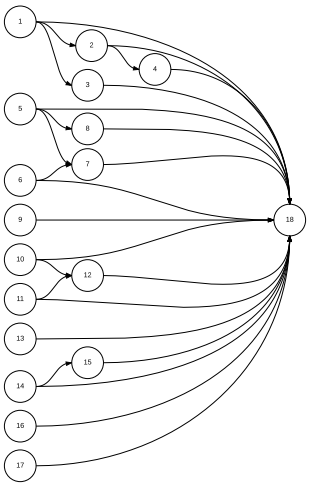

Figure 1: A Direct Acyclic Graph (*DAG*) representation of a firewall security policy.

2.3. Firewall Matching Optimization: Legacy Approaches

In this section, we describe the relevant rule order optimization algorithms that are based on matching optimization, and outline the general problem of firewall Rule Re-ordering (RR).

The task of optimizing a firewall is comparable to that of solving the ***Traveling Salesman Problem*** (TSP) [16] with precedence constraints [8]. The standard TSP is defined as the task of finding the shortest route while traversing each city exactly once, given N cities and their intermediate distances. However, as observed by the author of [17], when constraints are included in the problem, it becomes more complex. The authors of [8] examined a variant of the TSP with

precedence constraints. This variant, known as the Time-Dependent Traveling Salesman Problem (TDTSP), considers the case when transition costs between two cities depends on the time of the visit[5]. This implies that certain cities can only be visited at a given time, and thus, trying to find an optimal path with such a constraint means that some cities must be visited before others due to the dependency relationships between the cites. This is precisely, a mapping of the problem of finding the optimal rule ordering in a firewall policy with dependency relationships, because finding the optimal rule ordering in a firewall entails creating a rule ordering such that some rules must be "visited" or compared against before other rules, until a match is found.

2.3.1. A Bubble Sort-like Algorithm

Since the problem is NP-hard, the author of [1], designed a simple heuristic algorithm, given in Algorithm 1, for optimizing firewalls rule ordering.

Algorithm 1: A simple Bubble Sort-like rule ordering algorithm.

```
   Data: A list of firewall rules
   Result: A new and improved ordering of firewall rules
1  done = False
2  while !done do
3      done = True
4      for (i = 1; i < n; i++) do
5          if (p_i < p_{i+1} AND r_i ∩ r_{i+1} = ∅) then
6              interchange rules and probabilities
7              done = False
```

By studying the algorithm, one observes that it is similar to the Bubble Sort algorithm. It compares neighbours and, if possible, swaps them. Further [1], in order to preserve the rule precedence relationships, the algorithm uses rule probabilities and rule intersection as the swapping criteria. For example, con-

[5]The authors of [8] confirm that the TDTSP can be mapped onto ***single machine job scheduling problem*** [8] which is known to be NP-hard [18]. Thus the optimization problem for firewall rules is also NP-hard. The only option to find the optimal solution requires an exhaustive search of the solution space — which is not a scalable solution. Rather, one must be content to find a sub-optimal solution using a heuristic algorithm.

sider the scenario when there are two rules, i.e., Rule1 and Rule2 where Rule1 has a lower probability than Rule 2, and where the rules don't intersect. This means that the rules are not dependant on each other and are thus "swappable". The algorithm will process the rules, in a pair-wise manner, until there are no more "swappable" rules.

The problem with this algorithm, however, is that one rule can prevent another from being re-ordered [1], rendering the algorithm to be unable to reorder groups of rules. The following is an example of this problem; suppose there are three rules, namely Rule1, Rule2, and Rule3. Rule 1 and Rule 3 have a dependency relationship, and the rules have the associated probabilities given in Table 6:

Rule	Prob.
Rule1	0.1
Rule2	0.5
Rule3	0.4

Table 6: An example with a small number of rules with their probabilities.

Ideally, the rule with the highest matching probability would appear at the beginning of the list of rules in order to reduce the number of packet matches. Thus, in order to preserve the dependency relationships, the optimal rule order is: Rule1, Rule3, Rule2. However, the algorithm by [1] is not able to achieve this rule ordering, as explained below.

The algorithm will first swap Rule1 with Rule2. It will then check if Rule1 can be swapped with Rule3, but because they intersect, they will not be swaped. In the second iteration of the While loop the problem encountered becomes evident. Indeed, because Rule2 is better than Rule1 they will not be swapped, and further, because Rule1 and Rule3 intersect they will not be swapped either. Thus, when the algorithm terminates, the final order will be, clearly, suboptimal[6].

[6]This is a very simplistic example. Indeed, the possibility of terminating on suboptimal solutions is accentuated when the number of rules is larger.

However, despite its problems, this algorithm will still create a rule ordering that is better than the original, if possible.

In the same vein (and inspired by the classical ascending-order sorting algorithms), Groutl *et al.* proposed a method [19] to optimize the performance of the firewall using rule re-ordering, and more particularly, swapping operations. The method aspires to push the most accessed rules to the front of the firewall. However, the method suffers from a fundamental impediment when the rules are dependent. Unfortunately, the swapping algorithm proposed in [19] does not accommodate for the precedence relationships that might occur between the filtering rules which renders the problem to be NP-hard. In this paper, we, on the other hand, propose a simple condition that can be perceived as a *Swapping Window* mechanism, and that ensures that no precedence constraint is violated.

2.3.2. A DAG-based Algorithm

The authors of [14] presented a heuristic algorithm for optimized policy RR that is able to re-order a policy containing precedence relationships (or a subgraph in the **DAG**) in such a way that the policy integrity is maintained. A short synopsis of the most important aspects of this algorithm is given below.

The algorithm functions by operating on certain data structures. It needs a set, $G(r_i)$, of rules containing the sub-graph rules of r_i, i.e. the dependency relationships for r_i. It also uses a FIFO Queue, S, to represent the optimal policy rule sorting, where S is initially empty. Additionally, it requires a list, Q, containing the rules to be sorted, and this list is initially equal to the original firewall policy, R.

For each pass, the algorithm selects the rule with the highest average subgraph probability from the graph of rules available during *that particular* pass. The selected rule is then inserted into the list of sorted rules, S, if it has no rules dependent on it. Otherwise, the algorithm iteratively sorts the subgraph of its dependents until it finds a rule that has no dependent rules and inserts that rule into the list of sorted rules. The algorithm then updates the respective

data structures and repeats the process until all the rules have been placed in S.

3. Proposed Solution

3.1. Overview of the Solution

The goal of this paper, as expressed in the problem statement, is not merely to create a rule ordering algorithm. Rather, our aim is to explore the problem of optimizing a firewall's **performance** in a **dynamic network**. This means that for the firewall to have an optimised performance at all times, there needs to be an explicit understanding of when the rules have to be re-ordered as the network traffic dynamically begins to favour other rules in the policy.

This implies the need for two algorithms: The first algorithm must be useful to achieve the necessary RR, and the second one must be capable of updating the rule probabilities as the network traffic fluctuates. From an overall perspective, we also need a single scheme that connects both these algorithms into a single, optimized adaptive firewall. We first consider the requirements for both these algorithms.

- **The RR algorithm** should to be able to sort a firewall's rule order based on each rule's matching probability, dependency relationships, and firewall position. This will ensure that the average packet matching time is reduced. In order to satisfy these criteria, the algorithm will need to have access to the current firewall security policy, a knowledge of the dependency relationships, and the matching probabilities of every rule. The details of this algorithm are presented in Section 3.2.

- **The traffic aware algorithm** should be able to update a rule's matching probability dynamically as the network's traffic state changes. This means that this algorithm will need to have access to the currently-applied firewall security policy and the current number of packet matches for each rule. In order to enable dynamic estimation of the rule matching

probabilities, we present a novel weak estimator, which is a central component of our approach, in Section 4.1.

- Finally, the above two algorithms must be combined in such a way that they can communicate with each other. The traffic aware algorithm needs to be able to update the probability associated with a rule, and this update must be reflected in the rules used by the RR algorithm. If this is not achieved, the RR will never be able to find the optimal rule ordering of the firewall when the traffic state of the network changes. Thus, we will, briefly, describe two mechanisms for triggering the RR, namely, periodically and "performance triggered". These are described briefly in Section 4.2, and in more detail in the section that reports the experimental results that we have obtained, Section 5.

The primary reason why the problem is complex is because the RR and traffic-aware criteria themselves may be conflicting. Indeed, rule r_i may have to precede r_j with regard to the network's security policy requirements, and yet the probability of r_j being applied may be greater than that of r_i being applied. However, we will consider the RR issue first.

3.2. The Rule Re-Ordering Algorithm

The algorithm that we propose for RR, uses as its foundation the simple RR algorithm described earlier and presented in [1], namely Algorithm 1. Our new strategy is shown in Algorithm 2. However, before we formally present the algorithm, we shall explain its rationale.

3.2.1. Rationale for the Algorithm

Quite naturally, the algorithm itself takes as its input, a list of rules. It also has a list of rules that each given rule must *precede*, and which each rule must *succeed*. If these lists collectively formed a *DAG* that represented a *single string* of connected edges with a single source and a single sink, the problem of re-ordering the rules would have been trivial. The problem is necessarily complex

because the set of lists of *preceding* and *succeeding* nodes could be potentially conflicting. Our solution represents a heuristic scheme by which these conflicting requirements are resolved in the best possible manner.

To be more specific, the algorithm itself takes as its input, a list of rules. It also maintains two data structures.

- **The preceding list** of a rule, r_i, contains all the rules that are dependent on r_i. Essentially, this means that r_i must appear **before** the rules in the preceding list in order to maintain the integrity of the policy.
- **The succeeding list** of a rule, r_i, contains all the rules that r_i is dependent on. Analogous to the above, this means that r_i must appear **after** the rules in the succeeding list in order to maintain the policy's integrity.

Our algorithm contains two main loops that it iterates through. For every iteration of the outer loop, the inner loop will traverse the whole list. The reason for this is that the algorithm will compare the current element in the outer loop, r_x, with the current element in the inner loop, r_y.

The algorithm will then try to find a **swapping window** between r_x and r_y. A swapping window is defined as an interval of positions in a firewall in which the two comparing rules can be swapped, without breaking the integrity of the firewall policy. The window is found by analyzing the two comparing rule's succeeding and preceding lists.

By finding the rule with the *highest* position in the firewall in the preceding list for r_x and the rule with the *lowest* position in the firewall in the succeeding list of r_y, an interval of positions can be found. Once such an interval has been determined, the algorithm will check if the window is a valid swapping window for the current rules being compared.

In order to check the validity of the swapping window, the algorithm will check if the current position of r_x is less than the lowest position in the succeeding list of r_y and if the position of r_y is grater than the highest position in the preceding list of r_x. If the latter expression is evaluated to be *True*, the swapping window is deemed to be valid.

However, the above is only valid if r_x has a higher position in the firewall than r_y. In the case where r_y has a higher position in the firewall than r_x (as seen in lines 12 and 13 in Algorithm 2) there is a slight difference in the swapping criteria. In this case, the r_x and r_y values in the "*If expression*" switch places. The swapping mechanism is illustrated in Figure 2.

Once the algorithm has found a valid swapping window and thus knows that r_x and r_y can be swapped without violating the integrity of the policy, it will do a simple comparison of the rules' matching probabilities in order to decide whether they should be swapped or not. Even if the algorithm determines that they should be swapped, the algorithm will not properly swap them yet. Rather, the algorithm will do this based on a criterion value, Δ_{new}, explained below.

The value Δ_{new} is created using the matching probability and position number of the rules being compared against and simply yields the estimated average matching time before and after swapping r_x and r_y. This can be said to represent the swapping rank of r_y. The higher the swapping rank, the more optimal the swap is considered to be. Consequently, the algorithm will then perform a test to check whether this Δ_{new} value is greater than the current maximum value of Δ, i.e., Δ_{max}. If it is greater, then Δ_{max} is re-set to assume this rule's Δ_{new} value, and this rule is now the optimal swapping option.

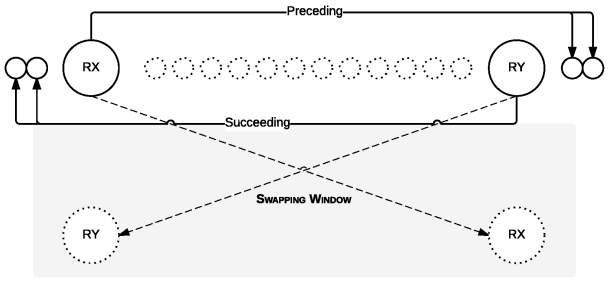

Figure 2: How Algorithm 2 re-orders rules.

When the inner loop has finished its traversal, a check is performed in order to find if r_x should be swapped with a rule or not. If it should be swapped, the rule with highest delta value, Δ_{max}, is chosen to be the optimal rule for it to be swapped with. Finally, the outer loop will complete the iteration and move on to the next rule at which point the process above is repeated for that rule.

In essence, what this heuristic algorithm tries to achieve, is to get as many rules as possible, with a high matching probability, as close to the top of the firewall as possible.

The formal algorithm follows.

4. Theoretical Results: Estimation and Rule Ordering

4.1. Designing a Weak Estimator for Batch Updates

Having described our RR algorithm, we now proceed to the issue of traffic estimation, and design a Weak Estimator scheme that is relevant for batch updates. The algorithm that we propose is a modified version of the *weak estimator* algorithm initially proposed by Oommen *et al* [20]. It is modified in such a way that it is able to use a batch of packet matches (as opposed to a single packet match as the SLWE scheme from [20] would do) in order to calculate

Algorithm 2: Our newly-proposed Rule Re-ordering algorithm.

```
   Data: A list of firewall rules
   Result: A new and improved ordering of firewall rules
1  for rx in rules do
2      Δ_max = 0
3      for ry in rules do
4          Δ_new = 0
5          if r_x ≠ ry then
6              if r_x.pos < r_y.pos then
7                  if r_x.pos < succeeding_max(r_y) AND
                     r_y.pos > preceding_min(rx) then
8                      if r_x.prob < r_y.prob then
9                          Δ_new = (r_y.prob − r_x.prob) ∗ (r_y.pos − r_x.pos)
10                         if Δ_max < Δ_new then
11                             Δ_max = Δ_new
12             else
13                 if r_y.pos < succeeding_max(rx) AND
                     r_x.pos > preceding_min(r_y) then
14                     if r_y.prob < r_x.prob then
15                         Δ_new = (r_x.prob − r_y.prob) ∗ (r_x.pos − r_y.pos)
16                         if Δ_max < Δ_new then
17                             Δ_max = Δ_new
18     if Δ_max > 0 then
19         swap(r_x, r_y)
```

the packet matching probabilities for a given rule. This ensures that the algorithm does not have to constantly perform estimate updates for each incoming packet.

The algorithm takes as its input a list of rules and a value for its parameter, λ. It then iterates through the list of rules and updates the probability associated with each rule by using the modified weak estimator algorithm given below. Quite simply put, in order to update the probability associated with each rule, the algorithm calculates it using the previous probability of the given rule, $\hat{p}_i$, the total number of packet matches, M, and the number of packet matches for any single rule, m_i. The pseudocode is given in Algorithm 3.

Algorithm 3: The Weak Estimator algorithm.

Data: A list of firewall rules, and a lambda value
Result: Updated probabilities for each rule in the list of rules
for *rule i in rules* **do**
    $\hat{p}_i = \frac{m_i}{M}\hat{p}_i + \lambda(\hat{p}_i - \frac{m_i}{M})$

4.1.1. Theoretical Results: The Batch-oriented Weak Estimator

In this section, we present some theoretical results related to our algorithms. The first result concerned the optimality of the devised Batch-oriented Weak Estimator (Algorithm 3) described above. The algorithm is a generalisation of the Stochastic Learning Weak Estimator (SLWE) proposed by Oommen and Rueda [20]. The main difference is that the Stochastic Learning Weak Estimator operates in an incremental manner, i.e., updates the estimates of the probabilities upon receiving every single observation. As opposed to this, the Batch-oriented Weak Estimator proposed here is able to handle a batch of M observations.

Specifically, let X be a multinomially distributed random variable, which takes on the values from the set $\{'1', \ldots, 'r'\}$. We assume that X is governed by the distribution $S = [s_1, \ldots, s_r]^T$ as follows:

$X =$ 'i' with the probability s_i, where $\sum_{i=1}^{r} s_i = 1$.

We assume that between two discrete time instants n and $n+1$, we obtain a batch of M concrete realisations of X. Let $\{x(n,1), x(n,2), x(n,3), ..., x(n,M)\}$ denote the batch of M observations obtained between the time instants n and $n+1$. The intention of the exercise is to estimate S, i.e., s_i for $i = 1, \ldots, r$ based on the batch of observations. We achieve this by maintaining a running estimate $P(n) = [p_1(n), \ldots, p_r(n)]^T$ of S, where $p_i(n)$ is the estimate of s_i at time 'n', for $i = 1, \ldots, r$. We omit the reference to time 'n' in $P(n)$ whenever there is no confusion.

Let $m_i(n)$ be the number of elements in the batch $\{x(n,1), x(n,2), x(n,3), ..., x(n,M)\}$ for which $X =$ 'i'. Formally, $m_i(n) = \sum_{k=1}^{M} I(x(n,k) = 1)$ where $I(.)$ is the indicator function. Then, the values of $p_i(n), 1 \leq i \leq r$, are updated in the following way:

$$p_i(n+1) \quad \leftarrow \quad \frac{m_i(n)}{M} p_i(n) + \lambda\,(p_i(n) - \frac{m_i(n)}{M}). \tag{1}$$

The reader should note that the above algorithm is a generalization of Oommen and Rueda's original SLWE algorithm [20]. In fact, when $M = 1$, the above updated equation coincides with the original algorithm devised in [20].

The properties of the estimator are catalogued and proven below.

Theorem 1. *Let the parameter S of the multinomial distribution be estimated by $P(n)$ at time 'n' as per equation (1). Then, $E[P(\infty)] = S$.*

Proof. The expected value of $p_i(n+1)$ given the estimated probabilities at time 'n', P, is:

$$\begin{aligned}
\mathrm{E}[p_i(n+1)|P] &= [\frac{k}{M} + \lambda\,(p_i(n) - \frac{k}{M})] \sum_{k=0}^{M} Prob(m_i(n) = k) && (2)\\
&= [\frac{k}{M}(1-\lambda) + \lambda\, p_i(n)] \sum_{k=0}^{M} Prob(m_i(n) = k) && (3)\\
&= (1-\lambda) p_i + (1-\lambda) \sum_{k=0}^{M} \frac{k}{M} \binom{M}{k} s_i^k (1-s_i)^{M-k} && (4)\\
&= (1-\lambda) p_i + (1-\lambda) \sum_{k=0}^{M} \frac{k}{M} \frac{M!}{k!(M-k)!} s_i^k (1-s_i)^{M-k} && (5)\\
&= (1-\lambda) p_i + (1-\lambda) \sum_{k=1}^{M} \frac{(M-1)!}{(k-1)!(M-k)!} s_i^k (1-s_i)^{M-k} && (6)\\
&= (1-\lambda) p_i + (1-\lambda) s_i \sum_{k=1}^{M} \binom{M-1}{k-1} s_i^{k-1} (1-s_i)^{M-k} && (7)\\
&= (1-\lambda) p_i + (1-\lambda) s_i \sum_{l=0}^{M} \binom{M}{l} s_i^{l} (1-s_i)^{M-l} && (8)\\
&= (1-\lambda) p_i + (1-\lambda) s_i. && (9)
\end{aligned}$$

With regard to the algebraic manipulations, in Eq. (4) we have applied the

mulinomial distribution theorem in order to obtain $Prob(m_i(n) = k)$. Further, in Eq. (8), we have invoked a change of the variable k, where $k-1 = l$. Finally, in Eq. (9), we have applied the binomial theorem.

Taking expectations a second time, we have:

$$\mathrm{E}[p_i(n+1)] = (1-\lambda)s_i + (1-\lambda)\mathrm{E}[p_i(n)]\,. \tag{10}$$

As $n \to \infty$, both equations $\mathrm{E}[p_i(n+1)]$ and $\mathrm{E}[p_i(n)]$ converge to $\mathrm{E}[p_i(\infty)]$, and can be written:

$$\begin{aligned} \mathrm{E}[p_i(\infty)](1-\lambda) &= (1-\lambda)s_i & (11)\\ \Rightarrow \mathrm{E}[p_i(\infty)] &= s_i\,. & (12) \end{aligned}$$

The result follows because (12) is valid for every component p_i of P. □ □

The next result deals with the rate of convergence of the mean of the estimator.

Theorem 2. *The rate of convergence of P is* fully *determined by* λ.

Proof. The proof follows directly from the corresponding proof in [20]. It is omitted to avoid repetition. □

4.2. Theoretical Results: Triggering Rule Re-ordering

For triggering the decision to attempt RR in a dynamic environment, we will use two types of approaches: Schedule-based rule ordering and Performance-triggered rule ordering. In simple terms, the Schedule-based RR will re-order the rule after a fixed number of packets have been received. On the other hand, the Performance-triggered RR will re-order the rules whenever the performance of the current policy degrades. Obviously, the problem with Schedule-based approaches is that of determining the periodicity of change. While changing the rule ordering too frequently results in unnecessary computation, if it is changed with too low a frequency, it results in a system that is unable to

track the environments. As opposed to this, Performance-triggered ordering can avoid both these trends if a degradation can be detected. However the efficiency of such a scheme is dependent on how fast the degradation can be detected, which is actually quite related to resolving the change detection problem.

These two forms of mechanisms for triggering the RR, i.e., either periodically or Performance-triggered, are described in detail in the experimental results, Section 5.

With regard to Algorithm 2 we now prove a central result related to the condition that we use for swapping two rules, namely Δ_{new}. We will show that Δ_{new} is simply the difference between the average matching time before and after swapping. Indeed, we will prove two important properties of the RR algorithm which are the following:

- Whenever a swapping is performed, the average matching time of the firewall is decreased;
- The swapping condition based on the concept of the swapping window will preserve the integrity of the firewall.

In what follows, we shall use the notation that for any rule r_i, located at position $r_i.pos$, the associated probability of it being invoked is $r_y.prob$.

4.2.1. The Swapping Condition

Theorem 3. *The difference of the average matching time before and after swapping two rules r_x and r_y is given by:* $\Delta_{new} = (r_y.prob - r_x.prob).(r_y.pos - r_x.pos)$

Let $r_k.pos$ be the position of rule k before swapping r_x and r_y, and let $r_k.pos'$ be the position of rule k after swapping r_x and r_y. It is easy to note that:

- $r_k.pos = r_k.pos'$ if $k \neq x$ and $k \neq y$, and that
- $r_x.pos' = r_y.pos$
- $r_y.pos' = r_x.pos$.

$$
\begin{aligned}
\Delta_{new} &= \text{The Average time before Swapping} - \text{The Average time after Swapping} \\
&= \sum_{k=1}^{N} r_k.prob \cdot r_k.pos - \sum_{k=1}^{N} r_k.prob' \cdot r_k.pos \\
&= (r_x.pos\ r_x.prob + r_y.pos \cdot r_y.prob) - (r_x.pos' \cdot r_x.prob + r_y.pos' \cdot r_y.prob) \\
&= (r_x.pos\ r_x.prob + r_y.pos \cdot r_y.prob) - (r_y.pos \cdot r_x.prob + r_x.pos \cdot r_y.prob) \\
&= r_x.pos(r_x.prob - r_y.prob) + r_y.pos(r_y.prob - r_x.prob) \\
&= (r_y.prob - r_x.prob) \cdot (r_y.pos - r_x.pos).
\end{aligned}
$$

Note that $\Delta_{new} = (r_y.prob - r_x.prob) \cdot (r_y.pos - r_x.pos) = (r_x.prob - r_y.prob) \cdot (r_x.pos - r_y.pos)$.

The theorem is thus proven. □ □

4.2.2. Preserving Policy Integrity: Consistent Rule Re-ordering

Theorem 4. *A rule r_k does not introduce inconsistency (i.e., it obeys all precedences relationships) if:*

$$preceding\_min(r_k) < r_k.pos < suceeding\_max(r_k).$$

Proof. The reader will observe that obeying all the precedence relationships in which rule r_k is involved in, reduces to two conditions:

- $r_k.pos$ is less than any element in succeeding list of r_k
- $r_k.pos$ is bigger than any element in preceding list of r_k.

The above two conditions can be written as: $preceding\_min(r_k) < r_k.pos < suceeding\_max(r_k)$, proving the result. □

Theorem 5. *If $r_y.pos < succeeding\_max(rx)$* **AND** *$r_x.pos > preceding\_min(r_y)$*, then swapping r_x and r_y will not introduce inconsistency.

Proof. It is easy to prove that a rule r_k does not introduce inconsistency if $preceding\_min(r_k) < r_k.pos < suceeding\_max(r_k)$, implying that all the precedences relationships are not violated.

Let $r_k.pos$ be the position of rule k before swapping r_x and r_y, and let $r_k.pos'$ be the position of rule k after swapping r_x and r_y. Further, observe:

- $r_k.pos = r_k.pos'$ if $k \neq x$ and $k \neq y$, and that
- $r_x.pos' = r_y.pos$
- $r_y.pos' = r_x.pos$.

An inconsistency occurs *only* due to either a violation due to the new position of r_x or due to the new position of r_y. We will prove that the new position of r_x, i.e., $r_x.pos'$, does not yield inconsistency. In other words:

$preceding\_min(r_x) < r_x.pos' < suceeding\_max(r_x)$.

By our hypothesis, we have ensured that $r_y.pos < succeeding\_max(r_x)$ which is the same as $r_x.pos' < succeeding\_max(r_x)$. Since we have invoked the max operator, $r_x.pos'$, the new position is less than any element in succeeding list of r_x. Thus, $r_x.pos' < suceeding\_max(r_k)$.

We shall now prove that $preceding\_min(r_x) < r_x.pos'$. We know that:

- $r_x.pos < r_y.pos$ is equivalent to $r_y.pos' < r_x.pos'$ since $r_x.pos' = r_y.pos$ and $r_y.pos' = r_x.pos$;
- $preceding\_min(r_x) < r_x.pos$ implies that $preceding\_min(r_x) < r_y.pos'$.

By combining the above two observations we obtain: $preceding\_min(r_x) < r_y.pos' < r_x.pos'$. We have thus proved that $preceding\_min(r_x) < r_x.pos'$.

Similarly, we can prove that the new position of r_y will not introduce inconsistency if $r_y.pos'$ is less than any element in succeeding list of r_x. Hence the result. □ □

Theorem 6. *Suppose that:* $r_x.pos < succeeding\_max(r_y)$ **AND** $r_y.pos > preceding\_min(r_x)$ then swapping r_x and r_y will not introduce inconsistency. □

5. Experimental Results

In this section we will describe the experimental results obtained by testing our algorithm on a rigorous suite of environments. The experiments were divided into two categories, those involving **Static** and **Dynamic** environments respectively. While the static experiments were designed in such a manner that they were capable of only verifying the RR algorithm, the dynamic experiments verified the overall firewall optimizer. All together, we conducted six experiments, namely three static and three dynamic experiments.

5.1. Performance Metric: The Average Matching Time

The authors of [21] defined a metric describing the average matching time of an Access Control List (ACL). This metric can be applied to a firewall precisely because a firewall policy is comprised of ACL rules with dependency relationships. The following describes how the metric is calculated.

Let θ_i represent the matching probability of a rule r_i in R. Then the average matching time of the rule is:

$$r_i * \theta_i$$

In other words to find the average matching time, we have to simply multiply the rule r_i's probability with its current position in the firewall. Extending the above to the firewall, R, the average matching time of the firewall R can be denoted as,

$$\sum_{i=1}^{N} r_i * \theta_i, \text{ for } N > 1.$$

Thus, the average matching time is defined as the average number of rules that a packet must be compared against before a match is found. For example, if a policy R has an average matching time of 2.6, it means that on average, 2.6 packets will be compared against the rules, $\{r_i\}$, in R before a match is found. From this, it is apparent that to optimize a firewall, the average matching time of the firewall must be low. By a simple analysis one sees that this can be

achieved by ensuring that the rules with high probabilities are at the top of the firewall.

5.2. Experimental Environment

The experiments were conducted on virtual machine instances created on the *Alto Cloud* cloud service at the *Oslo and Akershus University College of Applied Sciences*. All the instances were obtained using an *ubuntu 14.14* server image provided in the cloud.

In order to test the algorithms and the resulting firewalls, we needed two machines. Machine1 (M1) would run the firewall and the optimization algorithms. Machine2 (M2) would generate network traffic using a traffic generating script. However, because the firewalls being tested contained rules with random source and destination IP addresses, the traffic generating script could not send the traffic through the internet because it would have been lost and never reached the firewall at M1. This was because there were no hosts in the environment that possessed those IP addresses. Consequently, in order to solve the problem, we needed a direct connection between M1 and M2. This connection was created by changing M2's default gateway to the IP of M1 so that all traffic from M2 was routed through M1. This ensured that the spoofed IPs in the network traffic generated by the traffic generating script running on M2, would reach the firewall at M1. Figure 3 illustrates this.

5.3. Static Experiment 1: Intra-rule Re-ordering

The intention of this experiment was to provide proof that the algorithm for optimizing the RR was able to re-order the rules in such a way that the rules with the highest probability were at the top of the firewall while still maintaining the integrity of the policy. This experiment specifically tested RR within an intra-dependant group of firewall rules.

The experiment used a small policy of eight rules as described in Table 7. The rules {A - D} and {E - H} are intra-dependent but not inter-dependent.

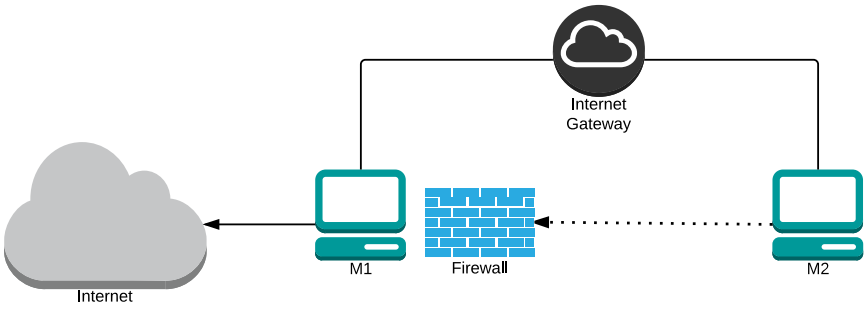

Figure 3: Proposed firewall testing environment.

This means that there are only dependency relationships between the respective rule groups {A - D} and {E - H}, but no relationships between the groups themselves. Figure 4 illustrates the relationships in Table 7 using a *DAG*.

			Source		Destination			
No.	Unique Name	Proto.	IP	PORT	IP	PORT	Action	Prob.
1	A	UDP	190.1.*	*	*	90	accept	0.1147
2	B	UDP	190.1.1.*	*	*	90-94	deny	0.0812
3	C	UDP	190.1.2.*	*	*	*	deny	0.4286
4	D	UDP	190.1.1.2	*	*	94	accept	0.1866
5	E	TCP	190.1.*	*	*	90	accept	0.0621
6	F	TCP	190.1.1.*	*	*	88	deny	0.0499
7	G	TCP	190.1.1.2	*	*	88-94	deny	0.0415
8	H	TCP	190.1.2.*	*	*	*	accept	0.0353

Table 7: The small firewall policy used for experiments in "Static Experiment 1".

From Table 7 one observes that this is no optimal rule ordering. Rules C and D have a higher probability than the rules A and B, and thus, C and D should be placed higher up in the firewall as long as the integrity is maintained. Furthermore this configuration yields the firewall an average matching time of 3.4921 units.

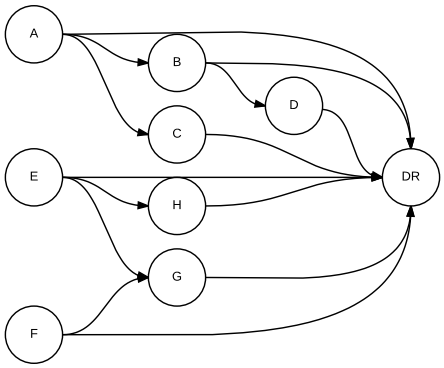

Figure 4: The DAG of the small firewall policy used for experiments in "Static Experiment 1". Here DR represents the *Default Rule*.

5.3.1. Expected results: Static Experiment 1

The expected results from this experiment are the following:

1. Rule C should be placed higher up than rule B, but below rule A.
2. In spite of having a higher probability than rule B, Rule D should not be placed higher up in the firewall because of the dependency relationship between rule B and D.
3. The average matching time will decrease when the policy is re-ordered for optimality. It should be 2.6535.

5.3.2. Results Obtained: Static Experiment 1

As mentioned earlier, the goal of this experiment was to show that the rule ordering algorithm was able to re-order rules while maintaining the integrity of the firewall policy. Figure 5 shows the initial conditions of the firewall.

Consider Figure 6. From this figure, it is apparent that rule C has been moved above rule B but below rule A. This is expected as there is no dependency relationship between rule B and C, but there is one between rules A and C which is why rule C must be placed below it in order that the integrity

```
Chain FORWARD (policy DROP 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               destination
       0        0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
       0        0 ACCEPT     tcp  --  *      *       0.0.0.0/0            192.168.128.206     tcp dpt:22 state NEW
      18      504 ACCEPT     udp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           udp dpt:90 state NEW /* A */
      19      532 DROP       udp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           udp dpts:90:94 state NEW /* B */
      87     2436 DROP       udp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* C */
      26      728 ACCEPT     udp  --  eth0   *       190.1.1.2            0.0.0.0/0           udp dpt:99 state NEW /* D */
       6      240 ACCEPT     tcp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           tcp dpt:90 state NEW /* E */
      10      400 DROP       tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpt:88 state NEW /* F */
      10      400 ACCEPT     tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpts:88:94 state NEW /* G */
       5      200 ACCEPT     tcp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* H */
```

Figure 5: The FORWARD chain of iptables containing the firewall rules for Experiment 1.

of the policy is maintained. The average matching time reduced to 2.6535, as expected.

```
Chain FORWARD (policy DROP 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               destination
       0        0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
       0        0 ACCEPT     tcp  --  *      *       0.0.0.0/0            192.168.128.206     tcp dpt:22 state NEW
       0        0 ACCEPT     udp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           udp dpt:90 state NEW /* A */
       0        0 DROP       udp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* C */
       0        0 DROP       udp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           udp dpts:90:94 state NEW /* B */
       0        0 ACCEPT     udp  --  eth0   *       190.1.1.2            0.0.0.0/0           udp dpt:99 state NEW /* D */
       0        0 DROP       tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpt:88 state NEW /* F */
       0        0 ACCEPT     tcp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           tcp dpt:90 state NEW /* E */
       0        0 ACCEPT     tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpts:88:94 state NEW /* G */
       0        0 ACCEPT     tcp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* H */
```

Figure 6: The FORWARD chain for Experiment 1 after Rule Re-ordering was achieved.

5.4. Static Experiment 2: Inter-rule Re-ordering

The intention of this experiment was to show how the rule ordering algorithm was able to re-order rules with no dependency while still maintaining the integrity of the policy. The experiment used a modified version of Table 7 where the intra-dependent rules, {E - H} had a higher probability than those of rules {A - D}. Table 8 illustrates the new table.

As can be observed from Table 8, the rules {E - H} should appear at the top of the policy, while rules {A - D} should be at the bottom. Because the groups of rules are independent from each other, the policy integrity should be maintained. The average matching time before optimization for this firewall

configuration is 5.1427.

			Source		Destination			
No.	Unique Name	Proto.	IP	PORT	IP	PORT	Action	Prob.
1	A	UDP	190.1.*	*	*	90	accept	0.0621
2	B	UDP	190.1.1.*	*	*	90-94	deny	0.0499
3	C	UDP	190.1.2.*	*	*	*	deny	0.0415
4	D	UDP	190.1.1.2	*	*	94	accept	0.0353
5	E	TCP	190.1.*	*	*	90	accept	0.4286
6	F	TCP	190.1.1.*	*	*	88	deny	0.1866
7	G	TCP	190.1.1.2	*	*	88-94	deny	0.1147
8	H	TCP	190.1.2.*	*	*	*	accept	0.0812

Table 8: The small firewall policy used for experiments in "Static Experiment 2".

5.4.1. Expected results: Static Experiment 2

The expected results from this experiment are the following:

1. The rules {E - H} should appear at the top of the policy in the same order, while the rules {A - D} should be at the bottom, and in the same order.
2. The average matching time should decrease when the policy is re-ordered for optimality. It should be 2.6535.

5.4.2. Results Obtained: Static Experiment 2

The results obtained confirmed that the RR algorithm was capable of re-ordering non-dependent rules while maintaining the policy integrity of the firewall. Figure 5 shows the initial conditions of the firewall.

The results shown in Figure 8 were essentially as expected. Rules {E - H} were at the top, as expected, while the rules {A - D} were at the bottom. The one difference from the expected results was that rule C was above rule B rather than the expected order of A, B, C and D. However, this is still a very positive result because our intent was to observe the re-ordering of *non-dependent* rules,

```
Chain FORWARD (policy DROP 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               destination
       0        0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
       0        0 ACCEPT     tcp  --  *      *       0.0.0.0/0            192.168.128.206     tcp dpt:22 state NEW
      11      308 ACCEPT     udp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           udp dpt:90 state NEW /* A */
       9      252 DROP       udp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           udp dpts:90:94 state NEW /* B */
      11      308 DROP       udp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* C */
       9      252 ACCEPT     udp  --  eth0   *       190.1.1.2            0.0.0.0/0           udp dpt:99 state NEW /* D */
      77     3080 ACCEPT     tcp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           tcp dpt:90 state NEW /* E */
      38     1520 DROP       tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpt:88 state NEW /* F */
      16      640 ACCEPT     tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpts:88:94 state NEW /* G */
      12      480 ACCEPT     tcp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* H */
```

Figure 7: The FORWARD chain of iptables containing the firewall rules for Experiment 2.

and thus, the intra-RR should have had no bearing on the outcome of the experiment. The average matching time was 2.6619, which is slightly worse than the expected value of 2.6535. The reason for this is that there were more packet matches for rule C than there were for rule B, in spite of the fact that rule B was characterized by a superior probability.

```
Chain FORWARD (policy DROP 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               destination
       0        0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
       0        0 ACCEPT     tcp  --  *      *       0.0.0.0/0            192.168.128.206     tcp dpt:22 state NEW
       0        0 ACCEPT     tcp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           tcp dpt:90 state NEW /* E */
       0        0 DROP       tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpt:88 state NEW /* F */
       0        0 ACCEPT     tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpts:88:94 state NEW /* G */
       0        0 ACCEPT     tcp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* H */
       0        0 ACCEPT     udp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           udp dpt:90 state NEW /* A */
       0        0 DROP       udp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* C */
       0        0 DROP       udp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           udp dpts:90:94 state NEW /* B */
       0        0 ACCEPT     udp  --  eth0   *       190.1.1.2            0.0.0.0/0           udp dpt:99 state NEW /* D */
```

Figure 8: The FORWARD chain for Experiment 2 after Rule Re-ordering was achieved.

5.5. Static Experiment 3: Comparing against Fulp's Bubble Sort-like Algorithm

The next experiment was done to obtain a comparison between our RR algorithm and Fulp's Bubble Sort-like rule ordering algorithm [1] given in Section 2.3.1 and to understand the differences in the criteria when it concerns the resulting policy re-orderings. Here, the actual firewall IPtables scripts was not so crucial, since we were only interested in testing the rule ordering algorithms. The only information that both the algorithms needed were the dependency

relationships between the rules and the matching probabilities of the various rules.

The experiment used a program to generate *DAG*s in order to generate generic dependency relationships and probabilities. The experiment used *DAG*s consisting of a 100 nodes (rules). The optimality was measured by calculating the average matching time of the resulting optimized firewall policies after each algorithm had applied its rule ordering on the policy.

5.5.1. Expected results: Static Experiment 3

The algorithm presented in [1] (and given in Section 2.3.1) does not take into account dependency relationships between multiple non-neighbouring rules, or the position of each rule within the policy when deciding to perform a swap. We can thus infer that it should generate policies with significantly worse average matching times than the algorithm designed by us.

5.5.2. Results Obtained: Static Experiment 3

With regard to the data generation strategy in the the program that created the *DAG*s, there was a variable that decided on the probability (given in terms of a percentage) that a pair of rules would have a dependency relationship between them. In this experiment, we set the value of this percentage to 1% and 5%. The results of the experiments for these two values are shown in Tables 9 and 10 respectively. Also, the metric that was used to compare the algorithm from [1] and our RR algorithm was the average matching time.

Table 9 shows the result of five tests done using a 1% chance of edges on a graph with 100 nodes (or rules). First of all, we notice that the algorithm from [1] is not able to noticeably improve the average matching time. As opposed to this, our algorithm is able to significantly improve the average matching time. By way of example, the algorithm from [1] reduced the average matching time from 34.8254 to 31.7256 (by 8.9%). As opposed to this, our algorithm had an average matching time of only 11.788 – a significant improvement of 66.2%. Indeed, it was 62.8% better than the algorithm from [1].

Percentage	Initial	Our Algorithm	Fulp	Number of Rules
1 %	26.42	12.42	24.157	100
1 %	41.18	12.16	38.81	100
1 %	31.657	12.12	24.45	100
1 %	37.31	10.87	36.2	100
1 %	37.56	11.37	35.011	100

Table 9: A comparison of our algorithm with the one presented in [1] with the *DAG* characterized by a 1% chance of an edge between two nodes.

Regarding Table 10, the results obtained essentially state the same conclusion. The algorithm from [1] reduced the average matching time to 41.477 (from 42.2902, i.e., by 1.9%). Our algorithm, on the other hand yielded an average of 25.5324, which was an improvement of 39.6%. Comparatively, our algorithm was 38.4% better than the one reported in [1].

Percentage	Initial	Our Algorithm	Fulp	Number Rules
5 %	41.98	22.87	41.33	100
5 %	30.28	21.321	29.06	100
5 %	46.371	27.701	45.52	100
5 %	33.7	21.71	33.115	100
5 %	59.12	28.06	58.36	100

Table 10: A comparison of our algorithm with the one presented in [1] with the *DAG* characterized by a 5% chance of an edge between two nodes.

Overall, we can conclusively state that our algorithm was significantly better than the one from [1]. However, understandably, we also noticed that the complexity of finding the optimal ordering increased with the density of the *DAG*. This is evident by observing that the average matching time increased significantly when the probability of having a directed edge between two nodes was increased (i.e., the graph was denser).

5.6. Dynamic Experiment 1: Schedule-based Rule Re-ordering with Dynamic Traffic

The intention of this experiment was to test both the RR algorithm and the Batch-oriented Weak Estimator algorithm in a dynamic network using a

Schedule-based re-ordering policy. The schedule policy was based on the quantity of packet matches in the firewall.

The experiment used two Zipf distributions based on the firewall in Table 7. The first distribution, **Zipf_dist_X**, gave a higher probability to the rules in the group {E - H}, while the second distribution, **Zipf_dist_Y**, gave a higher probability to the rules in the group {A - D}. The firewall optimizer script ran the RR algorithm for every 100 packet matches generated by the traffic generating script using the Zipf_dist_X distribution. After 1,000 packets had been matched, the traffic generator would switch the distribution to Zipf_dist_Y, while the optimizer script would continue to attempt RR every 100 packet matches.

With regard to the metric of comparison, for every iteration of the firewall optimizer, we calculated the average matching time of the current firewall policy configuration, using both the current Zipf distribution probabilities and the probability values estimated by using the Batch-oriented Weak Estimator algorithm. Such a process was able to produce the true average matching time and the estimated matching time per packet matched. A base line average matching time was also calculated, which was simply the the average matching time of the firewall without any RR. Storing these values as tuples, where each was stored with the current number of packet matches at the time of calculation, enabled us to create a graph displaying the improvement rate of the average matching time for the performance of the firewall optimizer script.

The X-axis of the graph represents the total number of packet matches and the Y-axis represents the average matching time. On this graph, we plotted the progression of the three different matching time metrics.

5.6.1. Expected results: Dynamic Experiment 1

The expected results from this experiment were the following:

1. The average matching times should be very high at the beginning, before steadily decreasing. Once the distribution switch occurs, the matching times should once again sharply increase before decreasing.

2. The exception should be the base line time, which should have a very high matching time until the traffic changes, at which point the average matching time should decrease sharply.
3. The true average matching time should increase and decrease at a sharper rate than the estimated average matching time.

5.6.2. Results Obtained: Dynamic Experiment 1

As mentioned earlier, the intention of this experiment was to test both the rule order and traffic aware algorithms using a schedule based re-ordering policy in a dynamic network. The resulting graphs for this experiment are given in Figures 9 and 10 respectively.

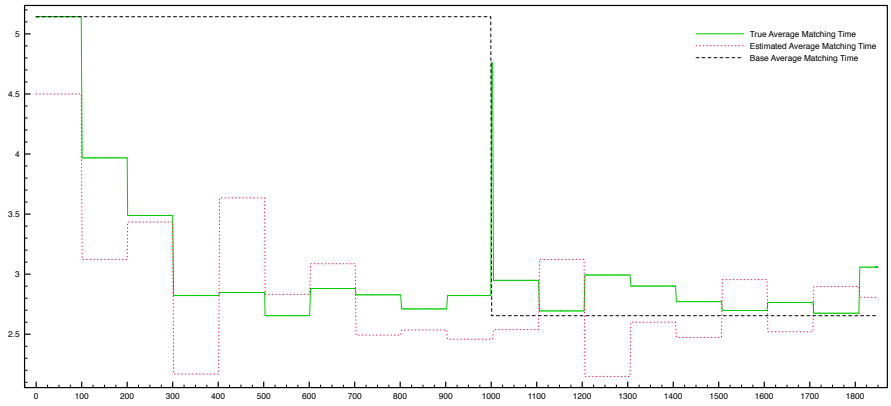

Figure 9: The results obtained from the first data set comparing our algorithm with a traditional schedule-based re-ordering policy.

The results of the first experiment demonstrate that the algorithms behaved as expected. We observe that both the **True** and **Estimated** average matching times start with high values, representative of a poorly-optimized rule ordering. They both, thereafter, start to gradually improve their times. However, there are some fluctuations in the results that leads to a spiking behavior. These spikes might be because of the nature of the traffic generator, because it does not consistently guarantee that packets with high probabilities are always chosen. Rather, the generator uses a "roulette wheel" function in order to decide

which rule is to be tested. Thus, we might end up with rules with relatively low associated probability being chosen at random and being sent to the firewall, causing the observed fluctuations.

When comparing the **True** and **Estimated** average matching times, we observe that they both match, relatively closely, with the **Estimated** values being consistently slightly below the true average matching times. Besides these observations, the base line time behaves as expected: it starts with a high average packet matching time until the switch, at which point it decreases rather sharply, and attains a matching time that is relatively close to the optimal.

The results of the second experiment are given in Figure 10.

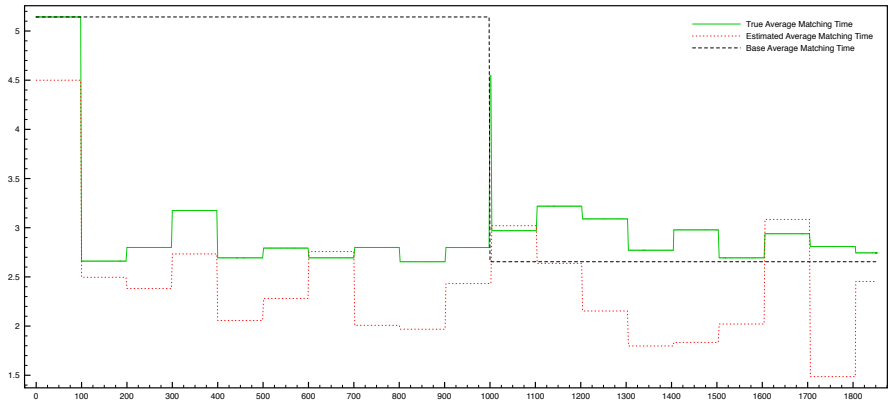

Figure 10: The results obtained from the second data set comparing our algorithm with a traditional schedule-based re-ordering policy.

These graphs display more unexpected results as there seems to have been more fluctuations. While the base line times are as expected, the **True** and **Estimated** times seem to be too flat. Again this could simply be due to the random phenomena due to the traffic generator. More importantly, we also observe that the **True** and **Estimated** times are not relatively aligned anymore, which might be because of the generally low updates to each rule as it matches a packet given by the weak estimator function.

5.7. Dynamic Experiment 2: Extended Schedule-based Re-ordering with Dynamic Traffic

The intention of this experiment was to see the long term effects of the algorithm in a dynamic network using a Schedule based re-ordering policy. Thus, this experiment was similar, in spirit, to the previous Schedule-based experiment. However, while the general setup was the same, the scale used here was different. The experiment used a larger firewall policy, consisting of 17 rules as defined in Table 4 (disregarding the default rule). Consequently, the Zipf distributions were also larger in order to match the firewall policy. Further, in this experiment, the traffic generator started to send data using an initially optimized Zipf distribution, and after 10,000 packets, it switched to a different less-optimised distribution.

The RR algorithm ran every 1,000 packet matches. The average matching times were calculated in the same manner as before, and the values were also stored in the same manner. The resulting graphs were also similar to those recorded for the previous experiment, but with higher maximum values on both axes.

5.7.1. Expected results: Dynamic Experiment 2

For this experiment:

1. We expected that the average matching times would first be as low as they can be and would thereafter continue to be relatively low close to their optimized values until the switch occurs. Thereafter, we expected them to increase at a fast rate, until the RR caused a steady decrease again.
2. Further, we expected that the base line time would initially have a low and generally optimized average matching time. It would then sharply increase once the switch occurs.
3. Finally, we expected the true average matching time to increase and decrease at sharper rates than the estimated average matching time.

5.7.2. Results Obtained: Dynamic Experiment 2

The results we obtained concurred with our expectations. They are given in the graph in Figure 11. As can be seen, the base line is within the expectations. It has a low average matching time in the beginning and once the switch occurs at 10,000 packet its average matching time increases sharply and becomes very poor.

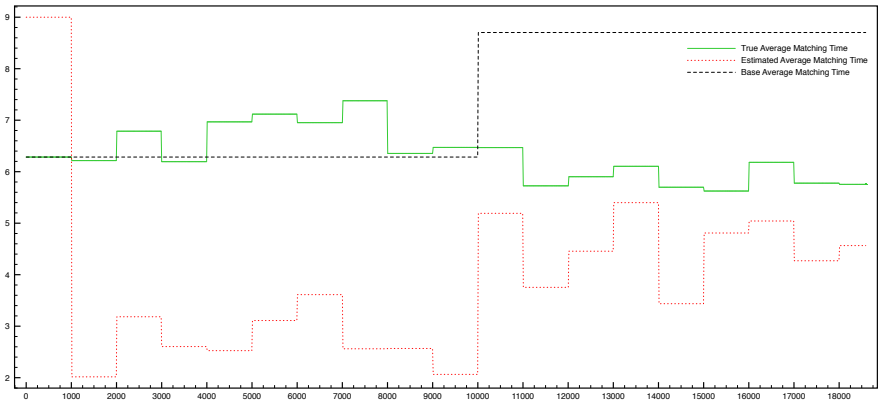

Figure 11: The results obtained for the second experiment, comparing our algorithm with an extended schedule-based re-ordering policy.

In this regard:

- We observe that the **True** average matching time behaved unexpectedly as one would have expected to see a sharp increase in the average matching time once the switch occured. However, comparing the **True** and **Estimated** times before the distribution switch, we observe that they behaved very similarly. The only difference between the two was that the **True** matching time had a much higher average matching time than the **Estimator**. This can be explained by the fact that our version of the weak estimator only increases each rule's estimate by a small value.

- We further noticed that the **Estimated** time behaved as expected. Indeed, this was with the exception that it had a consistently lower average matching time. This, in turn, can be explained in a similar manner,

because our version of the weak estimator increased/decreased the rule matching probabilities by only a small amount, each time.

5.8. Dynamic Experiment 3: Performance-Triggered Rule Re-ordering using a Sliding Window

The intention of this experiment was to observe the behavior of the algorithms when using a Performance-triggered criterion. Such a Performance-triggered criterion was based on a sliding window comprising of the most recent values of the estimated average matching time of the firewall. The experiment consisted of two parts both of which used the same Zipf distribution throughout the experiment. However, the first part shuffled the Zipf distribution at the traffic generator after every 500 packets sent. The second part shuffled the distribution after every 1,000 packets sent.

The firewall optimizer script ran the RR algorithm according to a Performance-triggered condition. The condition consisted of a list of the latest average matching times of the firewall. With each new calculation of the average matching time, the value was added to the list and if the list was full, the oldest element would be removed in order to make space for the latest value. This is, essentially, a sliding window. In order to decide whether to run RR or not, the optimizer script determined the trend of the sliding window. If the trend demonstrated that the average matching time was increasing, the RR procedure would be invoked. Otherwise, the RR procedure would not be called. The trend was calculated by computing the average of all but the latest value in the sliding window, and the average was compared against the latest value. If the average was greater, RR would run; otherwise, it would not.

The results enabled us to create a graph, in which the X-axis represented the total number of packet matches, and the Y-axis represented the average matching time.

5.8.1. Expected results: Dynamic Experiment 3

With regard to this experiment:

1. We expected more RRs, but with the additional benefit that the average matching time would stay relatively low throughout the experiment.
2. We also expected that the performance would increase with the size of the sliding window. In other words, we anticipated that the average matching time would vary inversely with the size of the sliding window.

5.8.2. Results Obtained: Dynamic Experiment 3

The results obtained by running this experiments are given in Figures 12 and 13.

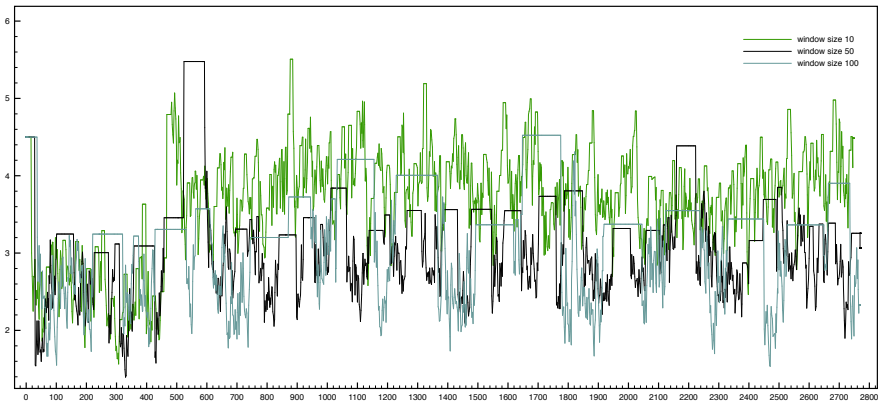

Figure 12: The graph obtained for the experiment with a dynamic environment where the distribution switched every 500 packets sent.

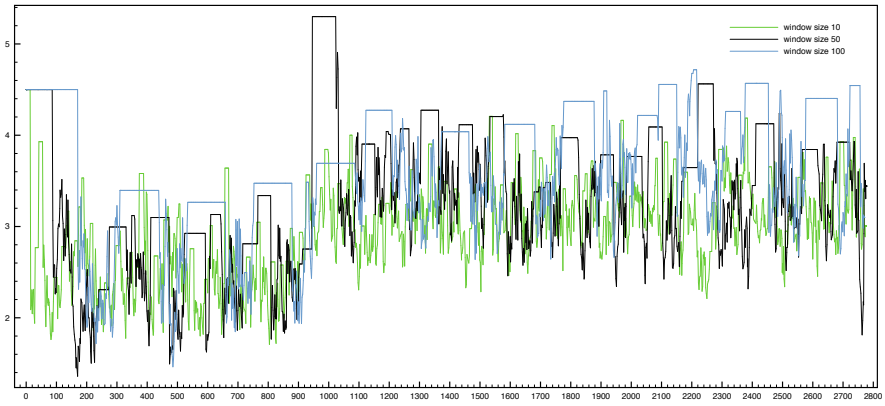

Figure 13: The graph obtained for the experiment with a dynamic environment where the distribution switched every 1,000 packets sent.

From observing the results we see that, in general, the average matching time decreased with the window size. The reason for this is because the scheme is provided with more information to determine if the trend displays an overall increase or decrease in the matching time. A small window size will, generally speaking, yield a lot of false positives resulting in a RR that occurs too early. For example, we notice that in Figure 12, when the window size is 10, the average matching time is consistently higher than for a window size equaling 50 and 100. However, in the graph where the distribution was switched after 1,000 (rather than just 500) packets, we observe that even a window size of 10 is able get comparatively good results relative to window sizes of 50 and 100. The reason for this is that the network traffic state will stay in a relatively stable state for a longer time span until the switch occurs. Of course, there are some fluctuations here, but they can be be caused by the random nature of the traffic generator.

Overall, the results match the expected results.

5.9. Discussions

In this paper, we evaluated the field of firewall optimization from two perspectives, namely that of firewall RR and traffic awareness. We designed and

implemented two algorithms, and combined these into a proof-of-concept firewall optimizer that was capable of re-ordering rules based on the statistics about the network concerning each firewall rule.

The RR algorithm was able to sort a firewall policy based on each rule's packet matching probability, its dependency relationships, and its current position in the firewall. In that sense, it was able to remedy the weaknesses of the work of [1]. Our solution is a simple but a rather deep algorithm in the sense that it considers many, different and potentially conflicting criteria for RR, by reducing the average matching time of the firewall.

The traffic aware algorithm was able to update a rule's matching probability for the various rules by reading the statistics of the IPtables. It utilized a batch-oriented novel version of the *Weak Estimator* algorithm due to Oommen and Rueda [2]. Our scheme was able to use a batch of packet matches to update a rules matching probability as opposed to the original SLWE which requires a *one-by-one* updating policy, and which had to, consequently, monitor every packet in the system. Our experience is that the single-rule-based SLWE lacks efficiency in this application domain.

5.9.1. The Experiments

In order to verify and confirm the hypothesis that the paper proposed, we conducted a total of six distinct experiments. The results of these were reported here, in all brevity. The goal and results of the experiments are as follows:

- The first two were simple proof-of-concept experiments with the goal of demonstrating that the RR algorithm was able to re-order rules in such a way that the new rule order was superior to the original one, and that the policy integrity was simultaneously kept intact.

- The third experiment was a comparative study, where the algorithm from [1] was compared against our RR algorithm. The intention was to show that our algorithm was able to, more efficiently, optimize a given firewall policy. The results we submitted demonstrated that our RR algorithm

performed significantly better than the one reported in [1] by as much as a 68% reduction in the average matching time.

- The last three experiments were more dynamic in nature. They were designed and implemented to improve the speed of our RR algorithm, the slowness of which can be attributed to the search required by a sorting algorithm which is not feasible, especially in a real-life scenario with large firewall policies. The first two of these utilized Schedule-based RR policies which simply specified when an action, i.e., to call the rule update function, had to be executed. The experiments set the execution policy to be after 100 packets and a 1,000 packets respectively. The resulting graphs showed that it could be a viable solution to the catalyze the RR algorithm.

- The last experiment was a Performance-triggered RR experiment, which similar to Schedule-based RR, used simple criteria to find when to run the rule update function. It calculated the optimal time instances to call the update function. In our experiment, we used a sliding window to contain the latest average matching times of the policy and sought to determine the trend within the window. If the trend was an overall increase in the matching time, it meant that the current rule ordering was sub-optimal and that it had to be re-ordered. The results from this experiment revealed that this type of RR worked very well. The graphs showed that the average matching time was consistently kept at a low level. However, reducing the computational intensity of Performance-triggering is still unsolved.

We conclude these discussions by mentioning that we believe that primarily, large organizations and institutes will benefit the most from the algorithms and results presented here, inasmuch as they handle large amounts of data from their workforce. Indeed, ensuring that the firewall is optimized is especially important because of the predictions made about the *Internet of Things*

[3]. The predictions are that in the foreseeable future, every person will have, on the average, 6.58 devices connected to the internet, resulting in, possibly, 50 billion devices being connected to the Internet worldwide. Having an optimized firewall will ensure that the firewall does not become a bottleneck in current and future high-speed networks. Additionally, with so many devices connected to the internet, there will inevitably be a lot of sensitive data stored on various networks. Thus, having an optimized firewall will be able to mitigate malicious attacks to the networks that use static and un-optimized firewall. Finally, our results can easily be integrated into existing firewalls such as IPtables.

5.9.2. Limitations of the Weak Estimator Algorithm

This paper also included a weak estimator when the samples appeared as a batch rather than individually. This *Batch-oriented Weak Estimator* algorithm is independent of the batch size. Further, the weak estimator algorithm has been generalized here to use the following formula to calculate an updated probability for a given rule as:

$$\hat{p}_i = \frac{m_i}{M}\hat{p}_i + \lambda(\hat{p}_i - \frac{m_i}{M}),$$

where m_i is the number of current packet matches for a given rule, and where M is the total amount of current packet matches for the entire firewall policy. It is apparent that the algorithm will not be able to tell the difference between proportionate values of m_i and M. Thus,

$$\frac{m_i = 4}{M = 5} \Leftrightarrow \frac{m_i = 40}{M = 50}$$

We are concerned that the ratio $\frac{40}{50}$ invloves more packets than $\frac{4}{5}$, and so the former should increase the associated probability more than the latter. The current version of the Batch-oriented Weak Estimator is incapable of this.

Another limitation of the *Batch-oriented Weak Estimator* algorithm is that it must be run often because each update of the estimates, changes the probabil-

ity values by only a small amount. This could potentially cause unnecessary overhead. Thus, it might be a advantageous to seek an improved estimation scheme that does not need to be run as often.

5.9.3. Future Work

The potential avenues for future work are:

- Considering existing heuristic algorithms for the Asymmetric Traveling Salesman Problem (TSP with precedence constraints) and their application for firewall rule optimization. We believe that this would be a fruitful avenue because a TSP with precedence constraints is equivalent to the problem of optimizing a firewall with rules having dependency relationships.
- Investigating new types of firewall optimization. One could attempt to create a program that is able to read the fingerprint of a network (i.e the current state of the traffic), and to generate an optimized firewall based on this fingerprint, and to thereafter store these firewall rules/orderings in a database. Then, on invocation, the traffic reading program should be able to read the current traffic pattern and apply the appropriate firewall based on the inferred fingerprint from the database.

6. Conclusion

The main goal of this paper was to investigate how we could optimize a firewall's rule ordering using the network's traffic statistics.

The problem statement was addressed by developing two algorithms to achieve the Rule Re-ordering (RR) in order to optimize the firewall's rules in a dynamic network. The first algorithm was a RR algorithm. It was distantly based on the philosophy introduced in [1]. However, our algorithm used more complex criteria for initiating RR, and we experimentally demonstrated that it was able to reduce the average matching time by as much as 68% than the

algorithm due to [1]. Our second main contribution was to devise a traffic-aware algorithm. It was based on the weak estimator algorithm proposed in [20]. However, it was modified to accommodate a batch updating of the rule probabilities rather than having to rely on keeping track of every packet in the system in order to update the rule probabilities.

Through various rigorous experiments, we have been able to show that the firewall performance optimizer worked very well, and that it was able to re-order the rules by using dynamic and time-varying information gleaned from the network.

[1] E. W. Fulp, Optimization of network firewall policies using ordered sets and directed acyclical graphs, in: Proc. of IEEE Internet Management Conference, 2005.

[2] L. Rueda, B. J. Oommen, Stochastic automata-based estimators for adaptively compressing files with nonstationary distributions, Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on 36 (5) (2006) 1196–1200.

[3] D. Evans, The internet of things: How the next evolution of the internet is changing everything, CISCO white paper 1.

[4] Q. Duan, E. Al-Shaer, Traffic-aware dynamic firewall policy management: techniques and applications, Communications Magazine, IEEE 51 (7) (2013) 73–79. doi:10.1109/MCOM.2013.6553681.

[5] S. Acharya, J. Wang, Z. Ge, T. Znati, A. Greenberg, Traffic-aware firewall optimization strategies, in: Communications, 2006. ICC '06. IEEE International Conference on, Vol. 5, 2006, pp. 2225–2230. doi:10.1109/ICC.2006.255101.

[6] C. Benecke, A parallel packet screen for high speed networks, in: Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual, IEEE, 1999, pp. 67–74.

[7] O. Paul, M. Laurent, S. Gombault, A full bandwidth atm firewall, in: Computer Security-ESORICS 2000, Springer, 2000, pp. 206–221.

[8] L.-P. Bigras, M. Gamache, G. Savard, The time-dependent traveling salesman problem and single machine scheduling problems with sequence dependent setup times, Discrete Optimization 5 (4) (2008) 685–699.

[9] D. Newman, Benchmarking terminology for firewall performance (rfc 2647), RFC 2647, RFC Editor, `http://www.rfc-editor.org/rfc/rfc2647.txt` (August 1999).
URL `http://www.rfc-editor.org/rfc/rfc2647.txt`

[10] K. Scarfone, P. Hoffman, Guidelines on firewalls and firewall policy, NIST Special Publication 800 (2009) 41.

[11] E. D. Zwicky, S. Cooper, D. B. Chapman, Building internet firewalls, " O'Reilly Media, Inc.", 2000.

[12] R. L. Ziegler, C. B. Constantine, Linux firewalls, Sams Publishing, 2002.

[13] J. Postel, Internet protocol (rfc 791), STD 5, RFC Editor, `http://www.rfc-editor.org/rfc/rfc791.txt` (September 1981).
URL `http://www.rfc-editor.org/rfc/rfc791.txt`

[14] A. Tapdiya, E. Fulp, Towards optimal firewall rule ordering utilizing directed acyclical graphs, in: Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on, 2009, pp. 1–6. doi:10.1109/ICCCN.2009.5235232.

[15] J. R. Vacca, Computer and information security handbook, Newnes, 2012.

[16] V. Grout, J. McGinn, Optimisation of policy-based internet routing using access control lists, in: Proceedings of the 9th IFIP/IEEE Symposium on Integrated Network Management, 2005.

[17] A. Schrijver, On the history of combinatorial optimization (till 1960), Handbooks in Operations Research and Management Science: Discrete Optimization 12 (2005) 1.

[18] R. L. Graham, E. L. Lawler, J. K. Lenstra, A. R. Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, Annals of discrete mathematics 5 (1979) 287–326.

[19] V. Grout, J. McGinn, J. Davies, Real-time optimisation of access control lists for efficient internet packet filtering, Journal of Heuristics 13 (5) (2007) 435–454.

[20] B. J. Oommen, L. Rueda, Stochastic learning-based weak estimation of multinomial random variables and its applications to pattern recognition in non-stationary environments, Pattern Recognition 39 (3) (2006) 328–341.

[21] V. Grout, J. Davies, J. McGinn, An argument for simple embedded acl optimisation, Computer Communications 30 (2) (2007) 280–287.