1-1-2010

A Simplified Method for Optimising Sequentially Processed Access Control Lists

Vic Grout
Glyndwr University, v.grout@glyndwr.ac.uk

John N. Davies
Glyndwr University, j.n.davies@glyndwr.ac.uk

Recommended Citation

Grout, V. & Davies, J.N., (2010) 'A Simplified Method for Optimising Sequentially Processed Access Control Lists'. Proceedings of the Sixth Advanced International Conference on Telecommunications (AICT), 9-15 May, held in Barcelona, Spain.

A Simplified Method for Optimising Sequentially Processed Access Control Lists

Abstract

Among the various options for implementing Internet packet filters in the form of Access Control Lists (ACLs), is the intuitive – but potentially crude – method of processing the ACL rules in sequential order. Although such an approach leads to variable processing times for each packet matched against the ACL, it also offers the opportunity to reduce this time by reordering its rules in response to changing traffic characteristics. A number of heuristics exist for optimising rule order in sequentially processed ACLs and the most efficient of these can be shown to have a beneficial effect in a majority of cases and for ACLs with relatively small numbers of rules. This paper presents an enhancement to this algorithm by reducing part of its complexity. Although the simplification involved leads to an instantaneous lack of accuracy, the long-term trade-off between processing speed and performance can be seen, through experimentation, to be positive. This improvement, though small, is consistent and worthwhile and can be observed in the majority of cases.

Keywords

internet traffic, access control, lists, packet, classification, ACL, optimisatino, δ-opt, ε-opt

Disciplines

Computer and Systems Architecture | Digital Communications and Networking | Hardware Systems | Systems and Communications

A Simplified Method for Optimising Sequentially Processed Access Control Lists

An efficient process for reordering rules in traffic packet filers

Vic Grout and John N. Davies
Centre for Applied Internet Research (CAIR)
Glyndŵr University, Wales
Wrexham, UK
{v.grout|j.n.davies}@glyndwr.ac.uk

***Abstract*-Among the various options for implementing Internet packet filters in the form of Access Control Lists (ACLs), is the intuitive – but potentially crude – method of processing the ACL rules in sequential order. Although such an approach leads to variable processing times for each packet matched against the ACL, it also offers the opportunity to reduce this time by reordering its rules in response to changing traffic characteristics. A number of heuristics exist for optimising rule order in sequentially processed ACLs and the most efficient of these can be shown to have a beneficial effect in a majority of cases and for ACLs with relatively small numbers of rules. This paper presents an enhancement to this algorithm by reducing part of its complexity. Although the simplification involved leads to an instantaneous lack of accuracy, the long-term trade-off between processing speed and performance can be seen, through experimentation, to be positive. This improvement, though small, is consistent and worthwhile and can be observed in the majority of cases.**

Keywords-Internet traffic; Access control lists; Packet classification; ACL optimisatio; δ-opt; ε-opt

I. INTRODUCTION: ACCESS CONTROL LISTS – INTERPRETATION AND IMPLEMENTATION

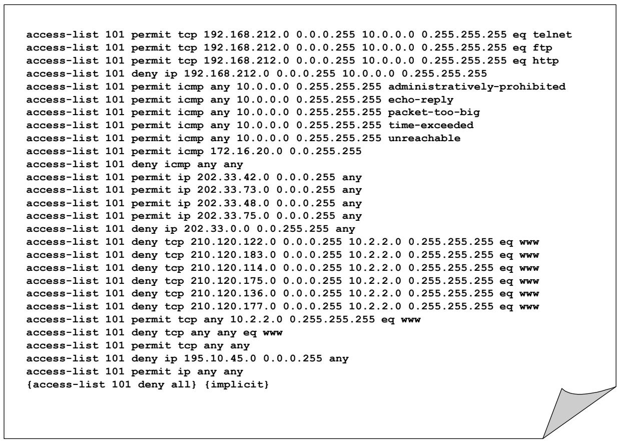
Internet devices such as *routers* switch *traffic*, usually in the form of discrete *packets*, between *networks*. The primary function of a router is to forward each packet to the most suitable device, typically another router, at each step of the journey. However, a vital secondary role is to consider whether a given packet should be passed at all, according to a set of tests, or *rules*, against which it is matched. An equally important third role is to select packets to which certain traffic *policies* apply – also achieved through the application of these same rules.

A typical rule, in the syntax of the Cisco *Internetwork Operating System* (*IOS*) [1], is

```
access-list 101 deny icmp any 10.0.0.0
                0.255.255.255 echo-reply,
```

which states that ICMP echo-reply packets from any source to the network `10.0.0.0` are to be blocked at this point. The first part of the rule simply assigns it to access list `101` (and may be ignored when discussing single lists in isolation).

A *packet filter*, or *Access Control List* (*ACL*), is then a sequence of such rules designed to implement a given objective or set of objectives. ACLs can be used for security purposes – simply to pass or block packets, or as filters for more sophisticated policies such as *traffic shaping*, *address translation*, *queuing* or *encryption* [2]. A packet may be matched against several ACLs on a single router and many more on its complete journey from source to destination. Inefficiently implemented ACLs can add significantly to packet delay and even small ACLs will contribute to this latency simply by their aggregation across several routers.



```
access-list 101 permit tcp 192.168.212.0 0.0.0.255 10.0.0.0 0.255.255.255 eq telnet
access-list 101 permit tcp 192.168.212.0 0.0.0.255 10.0.0.0 0.255.255.255 eq ftp
access-list 101 permit tcp 192.168.212.0 0.0.0.255 10.0.0.0 0.255.255.255 eq http
access-list 101 deny ip 192.168.212.0 0.0.0.255 10.0.0.0 0.255.255.255
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 administratively-prohibited
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 echo-reply
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 packet-too-big
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 time-exceeded
access-list 101 permit icmp any 10.0.0.0 0.255.255.255 unreachable
access-list 101 permit icmp 172.16.20.0 0.0.255.255
access-list 101 deny icmp any any
access-list 101 permit ip 202.33.42.0 0.0.0.255 any
access-list 101 permit ip 202.33.73.0 0.0.0.255 any
access-list 101 permit ip 202.33.48.0 0.0.0.255 any
access-list 101 permit ip 202.33.75.0 0.0.0.255 any
access-list 101 deny ip 202.33.0.0 0.0.255.255 any
access-list 101 deny tcp 210.120.122.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.183.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.114.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.175.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.136.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp 210.120.177.0 0.0.0.255 10.2.2.0 0.255.255.255 eq www
access-list 101 permit tcp any 10.2.2.0 0.255.255.255 eq www
access-list 101 deny tcp any any eq www
access-list 101 permit tcp any any
access-list 101 deny ip 195.10.45.0 0.0.0.255 any
access-list 101 permit ip any any
{access-list 101 deny all} {implicit}
```

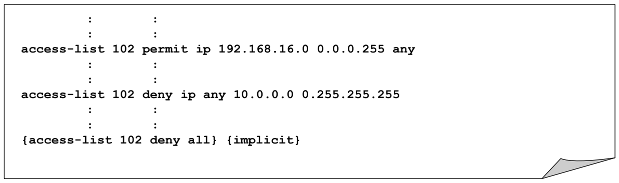
Figure 1. A typical Access Control List (ACL).

A. ACL Interpretation

An example of a complete ACL is given in Figure 1. Other than the ACL assignment, a rule may consist of up to five parts: the `permit` or `deny` type, the *protocol*, a *source address*, *destination address* and a *flag* function (as in the `echo-reply` parameter above) for fine-tuning. Each parameter may be a single value or a range of allowable matches. For example, the `any` parameter above matches all source addresses whilst the `0.255.255.255` parameter matches destination addresses in the `10.0.0.0` network. The absence of any term, such as an address, protocol or flag, indicates the rule will match a packet with any such values – provided those fields that are present *are* matched.

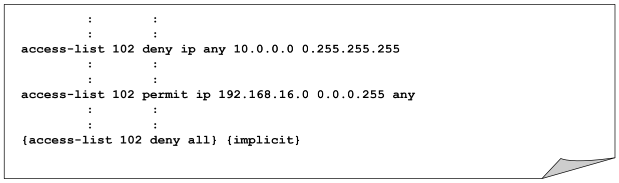
The interpretation of an ACL is that its rules are *considered* as being processed in sequential order from the top. That is, each incoming packet is tested against the first rule; if it matches, it is passed or blocked accordingly and no further rules are considered; otherwise it is tested against the second rule, and so on. There is an implicit `deny all` rule at the end of each ACL to block all packets not otherwise matched.

Access list 2(a)



```
        :       :
        :       :
access-list 102 permit ip 192.168.16.0 0.0.0.255 any
        :       :
        :       :
access-list 102 deny ip any 10.0.0.0 0.255.255.255
        :       :
        :       :
{access-list 102 deny all} {implicit}
```

Access list 2(b)



```
        :       :
        :       :
access-list 102 deny ip any 10.0.0.0 0.255.255.255
        :       :
        :       :
access-list 102 permit ip 192.168.16.0 0.0.0.255 any
        :       :
        :       :
{access-list 102 deny all} {implicit}
```

Figure 2. Dependent rules

There are three significant observations to make at this point:

1. This model of an ACL as a sequence of rules, considered in order, is only a question of *interpretation*: it should not be assumed that the ACL is actually *processed* sequentially within the device hardware or software.
2. For an ACL to be interpreted correctly. the order of the rules is crucial: an inherent *dependency* between rules prohibits arbitrary reordering. For example, in Figure 2, an IP packet from the network `192.168.16.0` to the network `10.0.0.0` will match both rules shown. The packet will be passed in 2(a) but blocked in 2(b). Clearly then, rules may not be reordered if this changes the underlying intention of the policy.
3. Not all rules are equally likely to match packets: rules with larger parameter ranges (or indeed absent parameters) may match more packets and rule *hit-rate* will vary among them. Also, different rules will become more or less significant as traffic (packet) characteristics change so these same hit-rates will be dynamic.

B. *ACL Implementation*

Space here permits only a brief overview of ACL implementation and optimisation. See [3][4][5][6] & [7] for a fuller treatment. There are essentially three basic approaches to implementation (TCAMs, trees/tries and linear lists) – although hybrids are also possible.

Implementation in TCAMs: A *Content Addressable Memory* (*CAM*) is effectively *Random Access Memory* (*RAM*) in reverse. Rather than accepting an address and returning the data at that location, a CAM can take an item of data and return the address at which it is to be found. In principle, the operation constitutes a single operation. A *Ternary CAM* (*TCAM*) permits wildcard bit matches along with binary ones and zeroes and is consequently ideal for allowing matches within *ranges* of addresses, protocols, etc. of the form to be found within ACL rules. CAMs and TCAMs can be used for various forms of *packet look-up* including routing tables as well as ACLs. In a routing table, the *longest* matching entry is returned; in an ACL, the *first*. This is the fastest but most expensive form of implementation. Not only is the immensely complex circuitry potentially restrictive; even cooling requirements can be an issue on large platforms [8].

Implementation as trees or tries: The concept of arranging ACL rules as a searchable tree structure (binary or otherwise) is a fairly obvious one. However, in practice, rules are better organised as *tries*. A trie (from 're*trie*val') is essentially a tree with an array of pointers at each node, indicating *subtries*. There is a pointer at each node for each possible value. The bits of each rule are thus stored on the braches of the trie, not the nodes. Rule look-up can be performed much faster on tries than trees. In each case the time taken to search for the first matching rule will be a (different) constant. However, there are considerable memory requirements for both approaches in addition to the processing complexity [5].

Implementation as linear lists: The simplest, but generally regarded as least efficient, approach to ACL implementation, is to process the rules sequentially as a linear list, precisely the original interpretation of rule order. In this case, the time taken to find the first matching rule will vary depending on the rule's position in the ACL. However, the value of this approach is that rules searched in this manner may be *reordered* to lower the average time for processing a *sequence* of packets, provided such a rearrangement does not violate any rule dependencies. There is a well-defined optimisation problem concerned with attempting to find such a minimising order, subject to dependency constraints. Unfortunately, it is shown in [6] that the problem is *NP-comlete* and only heuristics, not exact methods are viable. However, even for this effort to be worthwhile, the potential reduction in latency must be large enough to warrant running any optimising algorithm. These concepts are formulated and discussed in the next section.

II. PROCESSING ACLS SEQUENTIALLY – MODELLING, OPTIMISATION AND SIMULATION

A full formulation of the model of a sequentially processed ACL is given in [6]. Algorithms and performance are also discussed in [7][9] & [10].

A. *ACL Modelling*

Suppose there are *n* rules, r_1, r_2, .., r_n, in an ACL implemented as a linear list. Define a *dependency matrix*, *D*

$= (d_{ij})$ to be such that $d_{ij} = 1$ if rules r_i and r_j are dependent and 0 otherwise. If $d_{ij} = 1$ then the order of rules r_i and r_j must be preserved if the intended behaviour of the list is to be maintained. On this basis, the *dependency index*, a normalised measure of rule interdependency for the ACL, can be defined as

$$DI = \frac{2}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij}. \tag{1}$$

$DI = 0$ means no dependent rules; $DI = 1$ means all rules dependent upon all others. Higher values of DI constrain rule order more tightly.

Define the *hit-rate*, $h(r_i)$, of rule r_i to be the probability that a packet will match r_i. Hit-rates can be calculated dynamically (they will vary as traffic characteristics change) using counters within the IOS or hardware [11][12]. Define the *latency*, $\lambda(r_i)$, of a rule r_i to the time taken to (independently) process r_i. This may be calculated from the length of a rule, the nature of the protocols involved or taken from stored tables. In the implementation of some systems, latencies may be constant for all rules but this is not assumed generally in this paper.

The *cumulative latency*, $\kappa(r_i)$, for rule r_i can now be defined as the time taken to process r_i and all rules preceding it. So

$$\kappa(r_i) = \sum_{j=1}^{i} \lambda(r_j). \tag{2}$$

The *expected latency*, E, of the ACL, is then given by

$$E = \sum_{i=1}^{n} h(r_i)\kappa(r_i) = \sum_{i=1}^{n} h(r_i) \sum_{j=1}^{i} \lambda(r_j). \tag{3}$$

The purpose of any optimisation procedure, applied in this context, would be to find, or approximate, the ordering of the rules of the ACL that minimises E, subject to the dependency constraints, D.

B. ACL Optimisation

As already mentioned, this problem is *NP-complete*. The authors of [6] investigate a number of solving algorithms of varying levels of sophistication although it is unclear whether any are capable of providing a worthwhile return implemented on a production router. The main problem with most conventional sorting or swapping heuristics, in this context, is the need to re-evaluate the expected latency, E, for every revised rule order proposed, which greatly increases the processing complexity. However, consider the following.

Define the *trade-off coefficient*, T_i, to be the (possible) decrease in expected latency from swapping rules r_i and r_{i-1}. Then

$$\begin{aligned} T_i = {} & \sum_{k=1}^{i-2} h(r_k)\kappa_k + h(r_{i-1})\kappa_{i-1} + h(r_i)\kappa_i + \sum_{k=i+1}^{n} h(r_k)\kappa_k \\ & - \sum_{k=1}^{i-2} h(r_k)\kappa_k - h(r_{i-1})\kappa_i - h(r_i)\kappa_{i-1} - \sum_{k=i+1}^{n} h(r_k)\kappa_k \\ = {} & h(r_{i-1})\lambda(r_{i-1}) + h(r_i)[\lambda(r_{i-1}) + \lambda(r_i)] \\ & - h(r_i)\lambda(r_i) - h(r_{i-1})[\lambda(r_{i-1}) + \lambda(r_i)] \\ = {} & h(r_i)\lambda(r_{i-1}) - h(r_{i-1})\lambda(r_i), \end{aligned} \tag{4}$$

which, for *consecutive* rules, is a simple calculation, not a re-evaluation of the complete expected latency, E.

On this basis, [9] offers the following algorithm, named *δ-opt*, and refined in [7]:

```
δ-opt:

Step 1: Initialisation (on
            configuration/reconfiguration)
        for $i$ := 1 to $n$ do
                $h(r_i)$ := 1

Step 2: Promotion (on a match of rule $r_i$)
        $h(r_i) := \theta h(r_i)$;
        if ($d_{i-1\ i} = 0$) and
            $h(r_i)\lambda(r_{i-1}) > h(r_{i-1})\lambda(r_i)$ then
                Swap($r_{i-1}$, $r_i$)

Step 3: Reduction (every DSIZE packets)
        for $i$ := 1 to $n$ do
                $h(r_i) := h(r_i) / \max_j h(r_j)$.
```

The process works by increasing the hit-rate of the currently matched rule (by a factor, θ) and promoting it one place in the list if the trade-off in expected latency is positive. All hit-rates are assumed equal when the list is originally defined (or redefined) by the network administrator. Extensive simulation [7] suggests the ideal value of θ to be approximately 2. (This also makes the implementation more efficient: multiplying by 2 is a simple register shift.)

This is certainly a very simple and efficient algorithm. The linear ($O(n)$) `Step 1` is executed only once, as the list is defined or redefined – an infrequent event. `Step 3` (also $O(n)$) executes at intervals to prevent buffer overflow (`DSIZE` is the size, in bytes, of the registers holding hit-rates). Only the constant `Step 2` executes for each packet. Even so, it is not immediately clear that the latency savings from running such an algorithm will justify its execution time. That this is actually so can be demonstrated through simulation.

C. ACL Simulation

In the original paper [7], a comprehensive simulation process is described, based on an in-house numerical model, capable of generating ACLs and traffic flows according to a given parameter set. (Only an abridged version is given here – the original paper also extensively justifies the use of simulation rather than 'real-world' testing.) For tested ACLs, the number of rules (n) ranged from 10 to $10\,000$. Values of the dependency index, DI, in the range 0 (no dependencies) to 1 (complete dependency) were used. For each rule pair, (i,j), dependencies are randomised as $d_{ij} = 1$ with probability DI and $d_{ij} = 0$ with probability $1 - DI$. Rule latencies were uniformly randomised from $0.5\mu s$ to $1.0\mu s$. Actual values depend on the router hardware of course [5] but it is only *relative* values that are significant. (Routers that process packets faster will also optimise faster.) For traffic, the simulation is slightly more sophisticated. The traffic simulator generates packets with given probabilities of matching each rule in the list. At intervals, these probabilities may change to reflect shifting traffic patterns. Within a single traffic pattern, however, there is a certain probability that a packet is identical (other than the payload) to the previous one – or part of a similar stream - and thus matches the same rule. So, at the start of the simulation, a value of a *similarity index*, *SI*, is set. Then a *match probability*, ρ_i is randomised for each rule r_i and normalised so that $\sum_{i=1}^{n}\rho_i = 1$. The first packet is generated, matching each rule r_i with probability ρ_i. Subsequent packets match the same rule with probability *SI*, and otherwise match any rule according to the match probabilities, ρ_i. Every q packets, the match probabilities, ρ_i, are re-randomised.

n and *DI* can be set to produce different types of ACL while q and *SI* vary to reflect different types of traffic. As an example, Figure 3 shows simulated output from a test with $\theta = 1.5$ (from δ-*opt*), $n = 1\,000$, $DI = 0.25$, $q = 1\,000\,000$ and $SI = 0.75$. *4 000 000* packets were generated in total, in four stages with varying profiles. Results were reported every 100 000 packets.

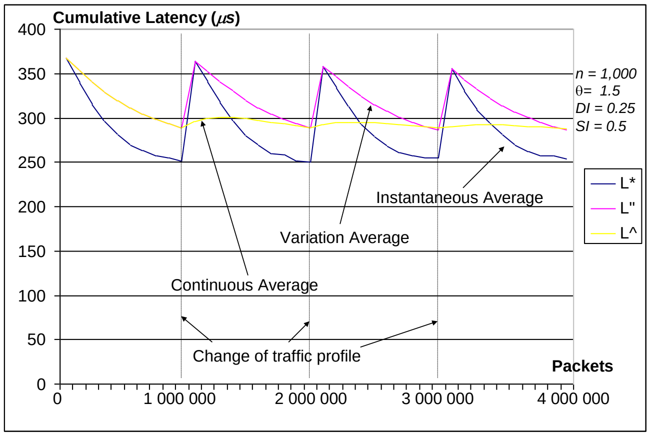


Figure 3. Simulated results: cumulative latencies

A number of results can be recorded; in particular, the mean position (*rank*) of the matched rule in the ACL and the mean cumulative latency of this rule. In both cases, three values are considered: the mean since the last set of figures (R^* & L^*) – the *instantaneous average*, the mean since the last traffic variation (*R” & L”*) – the *variation average*, and the mean of the entire simulation (*R^ & L^*) – the *continuous average*. The three latency averages, *L**, *L”* and *L^*, are plotted in Figure 3. The mean rank, *R*, for a *1 000* rule list with no optimisation will be *(1 + 1 000) / 2 = 500.5* and the mean cumulative latency, *L*, for a latency range of *0.5* to *1.0*, $500.5 \times (0.5 + 1.0) / 2 = 375.375$. In simulation, optimised averages start at these values and are then progressively lowered as rules with high hit rates are promoted. When traffic profiles change, instantaneous and variation averages become poor again but are gradually improved once more as the ACL adapts to the new characteristics. The continuous average becomes steadier over time. In this example, *L^* approaches a figure of approximately *287*, an improvement of *23%* (rounded) on the non-optimised figure.

This improvement figure, notated as φ, can be determined, through this simulation, for any combination of parameters. High values of *DI* work against the optimisation process, prohibiting desirable swaps. In the extreme cases, $DI = 1$ prevents *any* optimisation whereas $DI = 0$ allows rules to move freely. High values of q and *SI* imply greater traffic stability, which improves the optimised values. The effect of θ is more subtle. High values make rule promotion faster, which works well for similar, stable traffic but can lead to repetitive, unnecessary swaps for continuously changing, or oscillating, traffic patterns. Simulation suggests a compromise, with a value around $\theta = 2$ appearing to maximise the improvement in expected latency in most cases.

For any given ACL, `Step 1` of the δ-*opt* algorithm is executed once and can be taken as part of the configuration (or reconfiguration), `Step 2`, is executed every processed packet, and `Step 3`, every *DSIZE* packets. `Step 2` consists of an assignment, two calculations, two comparisons and a conjunction (possibly) followed by a swap of six assignments – three for the rules and three for their hit-rates - twelve operations in all. `Step 3` has two loops of size n, one to establish the maximum value and the other to reduce each value. The mean complexity (of `Step 3`) each packet is then $2n / DSIZE$ and, in total, $12 + 2n / DSIZE$ for `Steps 2 & 3` combined. Matching a packet against a rule consists of at least one operation (permit or deny) followed by between *1* and *5* comparisons (Figure 1). Taking a mean of $1 + 3 = 4$ operations per rule and a percentage saving for an optimised list of φ gives an optimisation *trade-off* of

$$T_\delta = \frac{4\varphi n}{100} - 12 - \frac{2n}{DSIZE}, \tag{5}$$

which will be positive (i.e. worthwhile) when

$$\varphi > \frac{300}{n} + \frac{50}{DSIZE}. \tag{6}$$

For example, taking $\theta = 2$, $n = 1\,000$, $DSIZE = 16$ and $DI = SI = 0.5$ gives an improvement (from simulation) of $\varphi = 15\%$ and a trade-off of $T_\delta = (15 \times 1\,000) / 25 - 12 - 2\,000 / 16 = 463$, a positive benefit. This calculation can be extended across a range of values of n and *DSIZE* and, for each *DSIZE*, the key value of n^*, the size of ACL for which optimisation is profitable, can be determined. Table I, for example, fixes *DSIZE* at *16* and calculates n^* for various values of *DI* and *SI*.

So, for example, with $SI = 0.75$ and $DI = 0.25$, n^* is calculated as *12.57*. On this basis, δ-*opt* will be worthwhile for an ACL of fifteen rules but not one of ten rules. It is then trivial to separate those lists to which optimisation is to be applied from those to which it is not [10]. Of course, it is precisely for longer ACLs that optimisation will yield the best results.

TABLE I. δ-OPT OPTIMISATION TRADE-OFF: MINIMUM ACL LENGTH

	DI =	***0.0***	***0.25***	***0.5***	***0.75***	***1.0***
SI =	***0.00***	*25.26*	*27.59*	*30.37*	*43.64*	∞
	0.25	*21.62*	*25.26*	*27.59*	*38.09*	∞
	0.50	*16.78*	*20.17*	*25.26*	*33.80*	∞
	0.75	*12.06*	*12.57*	*17.78*	*27.59*	∞
	1.00	*11.16*	*11.59*	*15.89*	*23.30*	∞

$\theta = 2$, *DSIZE = 16*. Table shows the value of n^*, the minimum length of list for $T_\delta = \varphi n/25 - 12 - n/8$, to be positive (i.e. for optimisation to be worthwhile) for different values of *DI* and *SI*

TABLE II. δ-OPT REAL-WORLD EXAMPLES

ACL	***n***	***DI***	***SI =***	***0.00***	***0.25***	***0.50***	***0.75***	***1.00***
A	*16*	*0.47*		×	×	×	×	✓
B	*20*	*0.47*		×	×	✓	✓	✓
C	*55*	*0.30*		✓	✓	✓	✓	✓
D	*144*	*0.30*		✓	✓	✓	✓	✓
E	*19*	*0.47*		×	×	✓	✓	✓
F	*93*	*0.36*		✓	✓	✓	✓	✓
G	*111*	*0.39*		✓	✓	✓	✓	✓
H	*62*	*0.12*		✓	✓	✓	✓	✓
I	*172*	*0.43*		✓	✓	✓	✓	✓
J	*68*	*0.40*		✓	✓	✓	✓	✓
K	*24*	*0.45*		×	×	×	✓	✓

$\theta = 2$, DSIZE = 16. For 11 real-world ACLs, the table shows the cases where δ-opt is worthwhile (✓) or not (×) for different levels of traffic similarity (SI)

Table II summarises the characteristics of several ACLs taken from a variety of production applications. (ACLs B, C and D are taken from college/university LANs, F, G and H from company networks and A and E from Small Office/Home Office environments connecting to the Internet via an ISP. ACLs I, J and K are derived from templates for various standard security configurations.) *δ-opt* is seen to be effective in the majority of real-world cases.

III. A SIMPLIFIED ACL OPTIMISATION HEURISTIC: *ε-OPT*

Other than the calculated proof of its effectiveness, the operation of *δ-opt* is easily justified. The essential role of `Step 2` is to promote a recently matched rule, on the basis that there is an increased probability (dependent upon *SI*) that the next packet will match the same rule. This will lower the *cumulative latency* of processing this next packet. However, this promotion only takes place if the *expected latency* of the list is reduced by the move. (All promotions are constrained by rule dependencies.) It can be seen that the operations in `Step 2`, concerned with raising the rule hit rate and testing for the trade-off in expected latency, are major contributors to the overall complexity of the *δ-opt* algorithm.

A. *ε-opt Optimisation*

A simplified alternative can be considered in which the matched rule is promoted irrespective of the trade-off calculation (but still subject to dependency constraints), seeking to lower the cumulative latency at the possible expense of expected latency. If the $h(r_i)\lambda(r_{i-1}) > h(r_{i-1})\lambda(r_i)$ condition is removed, there is no need to increase the hit-rate of the matched rule ($h(r_i) := \theta h(r_i)$) and `Step 2` reduces to

```
Step 2: Promotion (on a match of rule $r_i$)
            if $d_{i-1\ i} = 0$ then
                        Swap($r_{i-1}$, $r_i$).
```

However, removing the hit-rate update and test makes initialisation (`Step 1`) and renormalisation (`Step 3`) redundant as well so the complete algorithm becomes just

```
ε-opt:(on a match of rule $r_i$)
            if $d_{i-1\ i} = 0$ then
                        Swap($r_{i-1}$, $r_i$).
```

ε-opt, by simply promoting the currently matched rule, if valid, may lead to unnecessary or inappropriate swaps and is likely to produce smaller improvements in expected latency compared with *δ-opt* and, in turn, poorer reductive performance. However, balanced against this is the considerably reduced complexity of *ε-opt* over *δ-opt*. Overall performance will depend on this balance.

B. *ε-opt Simulation*

The simulation experiments for *δ-opt*, from section II.C, can be repeated for *ε-opt*. Unsurprisingly, by forcing (permitted) swaps that may not lower expected latency, *ε-opt*, for various values of θ, n, DI, q and SI, provides percentage improvements, ψ, that are smaller than the equivalent φ for the same values. For example, for $\theta = 1.5$, $n = 1\ 000$, $DI = 0.25$, $q = 1\ 000\ 000$ and $SI = 0.75$, ψ is determined, through the same simulation process, to be *18%*, compared with the equivalent $\varphi = 23\%$ from section II.C.

However, in contrast, the complexity of *ε-opt* is less than that of *δ-opt* and the cost of its implementation smaller. In total, for each packet, *ε-opt* consists of a comparison and (definitely this time) a swap of six assignments – three for the rules and three for their hit-rates - seven operations in all. The equivalent optimisation trade-off for *ε-opt* is therefore

$$T_\varepsilon = \frac{4\psi n}{100} - 7, \tag{7}$$

which will be positive (i.e. worthwhile) when

$$\psi > \frac{175}{n} \tag{8}$$

Taking the example, from section II.C, of $\theta = 2$, $n = 1\ 000$ and $DI = SI = 0.5$ (*DSIZE* is no longer relevant as there is no potential register flow from increased hit-rates and therefore no need for reduction) gives an improvement of ψ

$= 12\%$ and a trade-off of $T_\varepsilon = (12 \times 1\,000) / 25 - 7 = 487$, compared with 463 from *δ-opt* with $\varphi = 15\%$. Once again, extending this calculation across a range of values of *n*, *DI* and *SI*, allows the key value of *n**, the size of ACL for which optimisation is profitable, to be determined. Table III, for example, gives the *ε-opt* equivalent of *δ-opt* for Table I.

TABLE III. ε-OPT OPTIMISATION TRADE-OFF: MINIMUM ACL LENGTH

	DI =	***0.0***	***0.25***	***0.5***	***0.75***	***1.0***
SI =	***0.00***	*24.23*	*27.56*	*30.67*	*44.11*	*∞*
	0.25	*19.88*	*24.44*	*27.50*	*38.29*	*∞*
	0.50	*14.72*	*18.67*	*24.31*	*33.78*	*∞*
	0.75	*9.91*	*10.09*	*15.20*	*26.29*	*∞*
	1.00	*9.32*	*9.84*	*13.36*	*20.89*	*∞*

$\theta = 2$, *DSIZE = 16*. Table shows the value of *n**, the minimum length of list for $T_\varepsilon = \varphi n/25 - 12 - n/8$, to be positive (i.e. for optimisation to be worthwhile) for different values of *DI* and *SI*

These figures can be seen to be lower in all cases except those with particularly disadvantageous values of *DI* and *SI*. The value of *n** has been reduced in nearly all cases – more so for lower values of *DI* and higher values of *SI*. The implication is that *ε-opt* will potentially be beneficial for a larger number of ACLs than *δ-opt*.

Returning to the real-world examples, the *ε-opt* equivalent of the *δ-opt* Table II is given in Table IV.

TABLE IV. ε-OPT REAL-WORLD EXAMPLES

ACL	***n***	***DI***	***SI =***	***0.00***	***0.25***	***0.50***	***0.75***	***1.00***
A	*16*	*0.47*		×	×	×	✓	✓
B	*20*	*0.47*		×	×	✓	✓	✓
C	*55*	*0.30*		✓	✓	✓	✓	✓
D	*144*	*0.30*		✓	✓	✓	✓	✓
E	*19*	*0.47*		×	×	✓	✓	✓
F	*93*	*0.36*		✓	✓	✓	✓	✓
G	*111*	*0.39*		✓	✓	✓	✓	✓
H	*62*	*0.12*		✓	✓	✓	✓	✓
I	*172*	*0.43*		✓	✓	✓	✓	✓
J	*68*	*0.40*		✓	✓	✓	✓	✓
K	*24*	*0.45*		×	×	✓	✓	✓

$\theta = 2$, DSIZE = 16. For 11 real-world ACLs, the table shows the cases where ε-opt is worthwhile (✓) or not (×) for different levels of traffic similarity (SI)

The difference between Tables II and IV is that *ε-opt* can be seen to be effective for two further ACL scenarios above *δ-opt* (namely ACL A with *SI = 0.75* and ACL K with *SI = 0.50*). The saving in complexity of *ε-opt* over *δ-opt* has more than compensated for its lack of precision in these cases.

IV. CONCLUSIONS AND FURTHER WORK

The improvement of *ε-opt* over *δ-opt* is small but significant and comes at no cost. In general, *ε-opt* will prove worthwhile for a greater variety of real-world ACLs than *δ-opt*. Obviously there are some instances where this is not the case; however, these can be clearly identified from Table III and the appropriate algorithm implemented accordingly. The following meta-heuristic offers a crude but effective approach:

```
Ω-Opt (applied to an ACL with
   dependency index $DI$ acting on
   traffic with similarity index $SI$)
if $SI + 1 - DI < 0.75$ then
      Apply δ-opt
else
      Apply ε-opt
```

This selection may even be made dynamically in response to changing traffic characteristics. Overall, the result is a demonstrably more effective algorithm.

A final note is that *ε-opt*, being smaller in terms of code than *δ-opt*, will itself take up less space when implemented on a production router.

REFERENCES

[1] A. Colton, Cisco IOS for IP Routing, Rocket Science Press Inc., 2002.

[2] Syngress, Building Cisco Remote Access Networks, Syngress Media, 2002.

[3] J. Qian, S. Hinrichs & K. Nahrstedt, ACLA: A Framework for Access Control List (ACL) Analysis and Optimization, Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security, May 21-22, 2001, Darmstadt, Germany.

[4] E. Al-Shaer & H. Hamed, Modeling and Management of Firewall Policies, IEEE Transactions on Network and Service Management, Vol. 1-1, April 2004.

[5] G. Varghese, Networking Algorithmics: An Interdisciplinary Approach to Designing Fast Networking Devices, Morgan Kaufmann, 2005.

[6] V. Grout, J. McGinn & J. Davies, Real-Time Optimisation of Access Control Lists for Efficient Internet Packet Filtering, Journal of Heuristics, Vol. 13, No. 5, October 2007, pp435-454

[7] V. Grout, J. Davies & J. McGinn, An Argument for Simple Embedded ACL Optimisation, Computer Communications, Vol. 30, No. 2, January 2007, pp280-287.

[8] N. McKeown, Internet Routers: Past, Present and Future, British Computer Society (BCS) 2006 Lovelace Medal Lecture, http://www.bcs.org/server.php?show=nav.7935 (accessed 10 December 2009).

[9] V. Grout, J. McGinn & J. Davies, Reducing Processing Latency in Network Traffic Filters, Proceedings of the 5th International Network Conference (INC 2005) Samos Island, Greece, 5th-7th July 2005, pp3-10.

[10] V. Grout, J. McGinn, J. Davies, R. Picking & S. Cunningham, Rule Dependencies in Access Control Lists, Proceedings of the IADIS International Conference WWW/Internet 2006 (ICWI 2006), Murcia, Spain, 5-8 October 2006, pp537-544.

[11] Cisco, ACL Optimizer and Hits Optimizer, Cisco Systems, www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/cw2000/fam_prod/acl_mgr/aclm_1_x/1_5/u_guide/ac1js.pdf (accessed 10 January 2009).

[12] Cisco, ACL Manager, Cisco Systems, http://www.cisco.com/en/US/partner/products/sw/cscowork/ps402/products_user_guide_book09186a00801f42b9.html (accessed 10 January 2009).