ACLA: A Framework for Access Control List (ACL) Analysis and Optimization

Jiang Qian, Susan Hinrichs, and Klara Nahrstedt
University of Illinois at Urbana-Champaign[1,3]: Cisco System, Inc.[1,2]

Key words: network security, security mechanisms, security evaluation, security policy, intranet security, vulnerability test, policy based management, global policy, access control list, ACL optimisation, ACL analysis, packet classification, firewall, packet filtering

Abstract: It is a challenging task for network administrators to correctly implement corporate security policies in a large network environment. Much of the security policy enforcement at the network level involves configuring the packet classification strategies using Access Control List (ACL). A gateway device performing traffic filtering can deploy ACLs with thousands of rules. Due to the difficulties of ACL configuration language, large ACLs can easily become redundant, inconsistent, and difficult to optimise or even understand. This problem is augmented by extrinsic factors such as administrator turnovers, unstructured and ill-planned topology changes. With multiple routers in the topology, all of the ACLs need to be configured in a consistent manner to enforce the corporate security policy. In such an environment, manual examination of ACLs to ensure security policy is implemented correctly is a nearly impossible task.

In this paper, we propose a novel framework to automate ACL analysis, thus greatly simplifying the network administrator's task of implementing and verifying corporate security policies. A set of algorithms is introduced to detect and remove redundant rules, discover and repair inconsistent rules, merge overlapping or adjacent rules, map an ACL with complex interleaving permit/deny rules to a more readable form consisting of all permits or denies, and finally compute a meta-ACL profile based on all ACLs along a network path. When applied to traffic filtering ACLs, the meta-profile provides insights to the administrator as to what traffic will flow successfully from source to destination. Based on the ideas presented in this paper, we've developed a generic library called ACLA (ACL Analyser).

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35413-2_36

R. Steinmetz et al. (eds.), *Communications and Multimedia Security Issues of the New Century*

1. INTRODUCTION

One of the fundamental tasks entrusted to a network administrator is to correctly implement the corporate security policies. It is especially a challenging task in a large corporation or Internet Service Providers (ISP) environment. When size of the network increases, so does the number of firewalls and routers, typically resulting in complex topology and exponential growth in network management efforts.

Much of the security policy enforcement at the network level involves packet classification using Access Control List (ACL). ACL in a router consists of an ordered list of rules, or Access Control Entries (ACE) that collectively define a packet classification scheme (we will use the term ACE and rule interchangeably).

The process of differentiating traffic using ACL is commonly referred to ACL matching, packet classification or packet filtering. In this paper, we use these terms interchangeably. A packet flowing through the router or firewall is examined at various stages of processing to determine if it matches any particular ACLs. If a match is discovered, a pre-configured set of actions takes place. For example, the ACL is used for traffic filtering on a network gateway. Packets matching a particular ACL are either permitted to pass through the gateway or denied. , Traffic filtering offers a powerful packet flow-control tool for administrators. ACL can be combined with security protocols such as IPSec to form corporate Virtual Private Network (VPN). Taking advantage of readily available Internet infrastructure, VPN is a compelling alternative to dedicated leased lines. In a VPN, the gateway devices are configured with ACLs to classify traffic that flows into secure tunnels. Even though ACL plays an important role in corporate network security, evaluating and even understanding an ACL configuration remains a tedious, time consuming and error prone process. The following sequence of Cisco router commands (figure 1) define a simple ACL configuration for traffic filtering:

```
ip access-list extended list1                          // start ACL named list1
        permit tcp 192.168.10.0  0.0.0.255  192.168.20.0  0.0.0.255 eq 80     //1st ACE
        permit tcp 192.168.11.0  0.0.0.255  192.168.20.0  0.0.0.255 eq 80     //2nd ACE
        permit tcp 192.168.15.0  0.0.0.255  192.168.20.0  0.0.0.255 eq 80     // 3rd ACE
        deny ip any any                                                       // 4th ACE
        exit                                           // end ACL list1 configuration
interface Ethernet0                                    // configure int. Ethernet0
        ip access-group list1 in                       // use list1 to filter inbound traffic
        exit                                           // end interface configuration
```

Figure 1. Simple ACL Configuration

Without going in to details of the command syntax, the goal of the ACL named *list1* is to permit HyperText Transport Protocol (HTTP) traffic from networks *192.168.10.0, 192.168.11.0*, and *192.168.15.0* to network *192.168.20.0*, and deny all other types of traffic. This ACL is then associated with interface *Ethernet0* to perform traffic filtering on inbound packets. An incoming packet is compared with each ACE starting from the 1st rule in an ordered fashion. A match happens if and only if all the attributes of the packet match the ACE rule. In this example, a packet with source address = 192.168.10.5, destination address = 192.168.20.10, protocol = TCP, and port = 80 is a match with the 1st ACE. This packet will be permitted. Figure 2 illustrates the ACL matching sequence common to all ACL applications [12]:

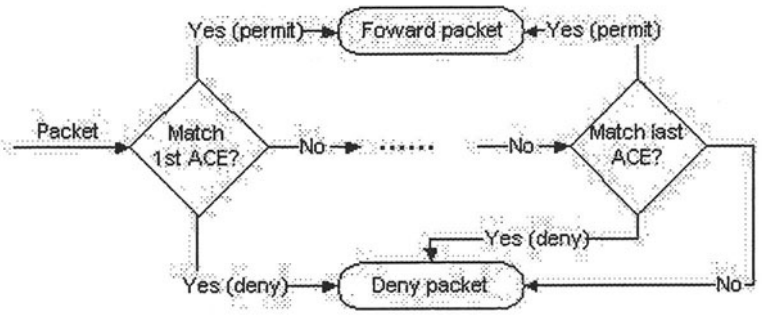

Figure 2. ACL Matching Algorithm

A gateway device performing traffic filtering can easily deploy ACLs with thousands of rules. As illustrated in the previous example, ACL configuration language is low-level and order-specific. As the network evolves with changing corporate needs, keeping these ACL configurations up to date is a challenging task. An ACL with large number of entries can easily become redundant, inconsistent, and difficult to optimise or even understand. This problem is augmented by extrinsic factors such as administrator turnovers, unstructured and ill-planned topology changes. With multiple routers in the topology, all of the ACLs need to be configured in a consistent manner to enforce the corporate security policy. In such an environment, manual examination of ACLs to ensure the security policy is implemented correctly is a nearly impossible task.

1.1 Contributions

In this paper, we propose a novel framework to automate ACL analysis, thus greatly simplifying the network administrator's task of implementing

and verifying corporate security policy. More specifically, we introduce efficient algorithms to:

- detect and remove redundant rules
- discover and repair inconsistent rules
- merge overlapping or adjacent rules to speed up packet classification and improve clarity
- map or filter an ACL with complex interleaving permit/deny rules to a more readable form consisting of all permits or denies
- compute a meta-ACL profile along a network path based on all ACLs encountered. This is typically applied to traffic filtering ACLs. The rule list generated as the result of this computation can be queried to answers questions such as:

  1. What are all the permitted traffic from src = X to dest = Y?
  2. What are all the permitted traffic from src = any to dest = $Y_1..Y_n$?
  3. Will traffic flow with src = X, dest = Y, protocol = TCP, port = 80 be permitted?

  These high level queries greatly simplifies the network manager's task of analysing ACLs and verifying correct implementation of corporate security policy.

In addition, we present a set of formal *rule relation* definitions. The meaning of *intersect*, *contain*, *overlap*, *disjoin*, *adjacent*, *inconsistent* and *redundant* are defined in precise mathematically terms. Internally, we use a dynamic multidimensional interval tree data structure [7][8] to store the rule list. Using a tree-based structure allows for efficient query, insert, and delete operations. To the best of our knowledge, no literature currently exists that performs such formal and complete analysis of ACLs.

Based on the ideas presented in this paper, we've developed a generic library called *ACLA (ACL Analyser)* that implements the above set of queries and operations. The algorithms and data structures proposed in this paper can easily be incorporated into other network management tool that would benefit from similar type of manipulation and optimisation of ACLs.

Section 2 discusses related work in this area. Section 3 defines mapping between an ACE and a multidimensional interval tree node, then describes the dynamic multidimensional interval tree data structure. Based on the data structure outlined in Section 3, Section 4 presents each ACL analysis algorithm in detail. A set of relations between rules is defined to formalise the analysis process. Finally, we conclude with future directions.

2. RELATED WORK

There are numerous vulnerability testing and intrusion detection tools on the market that analyse network security by active probes or monitors. Some of the leading products in this area are the Internet Security Systems Internet Scanner [13], Network Associates CyberCop Scanner [17], Cisco Secure Scanner [6], NESSUS [16], and SATAN [9][19]. The goal is to check for well-known security vulnerabilities, and report the result back to administrator. Vulnerability tests are an important part of security analysis process. However, it still operates at a low-level and does not give administrator a high-level view of the security policy. In addition, it can only detect security vulnerabilities after the fact. A separate mechanism is needed to prevent these vulnerabilities. The analysis proposed in this paper is a passive approach, and operates at a higher level. Potentially security vulnerabilities can be analysed before ACLs are deployed. Our work serves as a nice complement to the active vulnerability testing and intrusion detection tools.

Another security management approach related to our work is policy-based management. In [10], Guttman described a language for global policies and algorithms to generate local filtering rules. Similarly, Bartal, Mayer, Nissam, and Wool introduced the Firmato firewall toolkit [1]. Firmato derives per-device configuration from global policy with emphasis on firewall filtering rules. The leading commercial product in this area is CSPM (CiscoSecure Policy Manager) [5][11]. CSPM is a sophisticated tool that produces extensive device configurations based on the global policy. The tools presented in this work can be incorporated into the policy compilation process or used during post-processing to optimise the ACLs generated. More recently, Mayer, Wool and Ziskind introduced Fang [15]. Fang allows user to perform queries based on source range, destination range and service range. The result of query returns all traffic types that are permitted between source range and destination range. This is in spirit similar to our query on the meta-ACL profile. However, Fang does not offer ability to perform consistency checks, optimisation or filtering.

There has been much effort in recent years in developing efficient range searching data structures. Other data structures such as k-d tree [2], range tree [4], segment tree [3], and their variants are all possible alternatives to the interval tree data structure. K-d tree has a slower query time for this class of data structure. Range tree requires complex fractional cascading to reduce the query time. Segment tree is designed to handle non-axis parallel lines, which is not the case in our application. External-memory multidimensional search data structures such as grid file [18] and hB-tree [14] are common in database applications. We assume our ACL size is

small enough to perform query in memory. The interval tree data structure is chosen due to the simple mapping from rule to intervals, intuitive generalisation of higher dimensions, efficient insert, delete, and query time.

3. THE DYNAMIC MULTIDIMENSIONAL INTERVAL TREE DATA STRUCTURE

This section describes the fundamental data structure used in ACL analysis algorithms. First, a mapping between ACE attribute type and interval tree dimension is derived. The exact attribute types available in an ACE are vendor dependent. We will use Cisco's extended access list as an example. An ACE in an extended access list provides the following attribute types: source address, destination address, protocol, and port number. All values specified in each attribute form an interval. A special key work *any* can be used to specify all possible values for an attribute. Let x^i_{min} and x^i_{max} denote the minimum and maximum values of an interval in dimension i, and D^i_{min} and D^i_{max} denote the minimum and maximum possible values of dimension *i*. The mapping can be performed as follows:

Table 1. Sample ACE to Interval Tree Node Mapping

ACE Attribute	Interval Tree Node
Source Address	x^1_{min} = min address in range
	x^1_{max} = max address in range
Destination Address	x^2_{min} = min address in range
	x^2_{max} = max address in range
Protocol	$x^3_{min} = x^3_{max}$ = protocol number
Port	$x^4_{min} = x^4_{max}$ = port number
Any	$x^i_{min} = D^i_{min}$
	$x^i_{max} = D^i_{max}$

A Cisco extended access list maps to an interval tree node with 4 dimensions, with D^1 corresponding to source address, D^2 to destination address, D^3 to protocol, and D^4 to port number.

More formally, let $V := \{[v_1: v_{1'}], \ldots, [v_n: v_{n'}]\}$ be the set of *n* attribute values in an ACE. A corresponding multidimensional interval tree node is constructed as follows:

$$X := \{[x^1_{min}=v_1 : x^1_{max}=v_{1'}], \ldots, [x^n_{min} = v_n : x^n_{max} = v_{n'}]\}$$

The special attribute value *any* is always mapped to the interval $[D^i_{min} : D^i_{max}]$.

Using this mapping, we can easily convert any ACE into a multidimensional interval tree node. An interval tree is binary tree

constructed based on the end points of intervals. For example, let X be a set of intervals in one dimension, and let *m* be the median of the interval end points. The set of intervals containing *m* is stored at the root. The set of intervals to the left or right of *m* forms the left and right subtree. These intervals are recursively partitioned based on their median. Search, insert, and delete operations in a one-dimensional interval tree takes *O(logn)* time [7][8].

A multidimensional interval tree is a straightforward generalisation of the one-dimensional interval tree. An interval tree can be constructed first based on the intervals in the first dimension only. This first level interval tree (sometimes referred to as component tree) contains a set of elements in each tree node. Each node in turn stores its element using an interval tree based on the second dimension. Any query is reduced to a sequence of binary searches in each dimension. The multidimensional interval tree data structure implemented in our work is a dynamic interval tree in the sense that it can support inserts and deletes. Query, insert and delete operations take $O(nlog^d n)$ time [7][8], where d is the dimension.

4. ACL ANALYSIS ALGORITHMS

Before examining the ACL analysis algorithms in detail. We present a set of formal definitions on the *relation* between two rules. Precise definitions allow us to describe the algorithms in a concise and unambiguous manner.

Definition 1.
Given interval M = [m:m'], N= [n:n']
1. M *intersect* N if m <=n <= m' or m<=n'<=m' or n<=m<=n' or n<=m'<=n'
2. M *contain* N if m <= n and n'<= m'
3. M, N is *adjacent* if m'+1 = n or n'+1 = m

Definition 2.
Given rule X, Y, where X = [x_1, x_2, ... x_n] and Y= [y_1, y_2, ... y_n]
1. X ,Y *intersect* if for each x_i and y_i, x_i and y_i *intersect*
2. X ,Y is *disjoint* if ! *intersect*
3. X *contain* Y if for each x_i and y_i, x_i contains y_i
4. X ,Y *overlap* if there exist an x_i and y_i where x_i *intersect* y_i, for all other dimensions, $x_j == y_j$
5. X ,Y is *adjacent* if there exist an x_i and y_i that is *adjacent*, for all other dimensions, $x_i == y_I$

Contain and overlap are more specialised intersect relationship. Adjacency is a more specialised disjoint relationship. Figure 3 illustrates these relations.

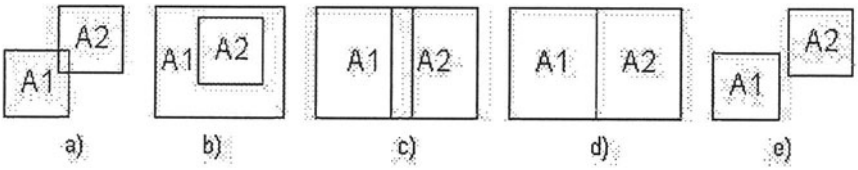

Figure 3. Rule relations: a) intersect b) contain c) overlap d) adjacent e) disjoint

The important observation to make here is that the geometric shape projected by an ACE is always rectilinear. Consequently, the relation tests only take a few simple comparisons in each dimension, which takes *O(d)* time, where *d* is the number of dimensions. Performing intersection test between two arbitrary shapes is a much harder task.

Based on the primitive relationships defined, we can formalise the notion of consistency and redundancy.

Given ACE X, Y, where X precedes Y in the ACL (note: the subscript p is used if the rule is a permit rule, d is used if it's a deny rule):

Definition 3.

X or Y is *redundant* if any of the following sufficient conditions apply

1. X_p contain Y_p
2. X_d contain Y_d
3. Y_p contain X_p and there does not exist rule Z_d between X_p and Y_p such that Z_d intersect X_p and X_p !contain Z_d
4. Y_d contains X_d and there does not exist rule Z_p between X_d and Y_d such that Z_p intersect X_d and X_d !contain Z_p

The first two conditions determine if Y is redundant. X contains Y implies packets that match Y are matched by X. Since X precedes Y, Y will never be used. Conditions 3 and 4 determine if X is redundant. The extra check for rule Z is needed because if such rule Z exists, and if rule X is removed, any packet with values that falls in the intersection region of X and Z could be matched by Z. The action specified in Z will be taken, not X. The packet classification property would be altered. If X contains Z, then Z is a *inconsistent* rule (see Definition 4), and would be removed. Figure 4 illustrates both cases.

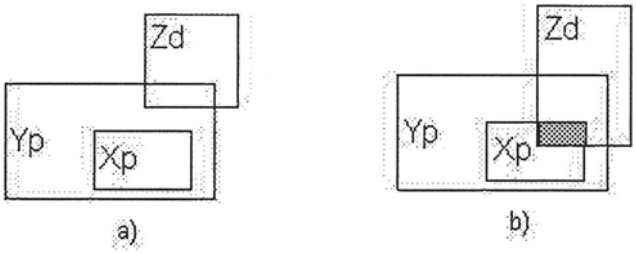

Figure 4. Redundancy Check. a) Xp is redundant, any packet that is permitted by X will be permitted by Y. b) Xp is not redundant, if Xp is removed, any packet that falls in the shaded region could be denied by Z

Definition 4.
X and Y are *inconsistent* if any of the following sufficient conditions apply
1. X_p contain Y_d
2. X_d contain Y_p

In case 1 and 2, rule Y is in some sense redundant. However, since the action Y triggers is opposite of X, different filtering behaviour will result if Y is used. This could indicate an inconsistency in terms of the packet classification goal of the ACL. Most likely, the network manager configured Y without realising it is contained in X. This could lead to a serious security compromise. (Note: It is a common and recommended practice to specify X and Y such that Y contains X. Figure 1 is an example such configuration. The idea is the administrator would first configure the rules corresponding to exceptional cases, and then one broad rule to cover the default case).

Definition 5.
1. X can be *merged* with Y (rule X is removed, rule Y is expanded) if any of the following sufficient conditions apply
2. X_p and Y_p *overlap* or are *adjacent* and there does not exist rule Z_d between X_p and Y_p such that Z_d *intersect* X_p and X_p !*contain* Z_d
3. X_d and Y_d *overlap* or are *adjacent* and there does not exist rule Z_p between X_d and Y_d such that Z_p *intersect* X_d and X_d !*contain* Z_p

The rational for the extra check for Z is similar to the redundancy case. If such rule Z exists, and if rule X is merged, any packet with values that falls in the intersection region of X and Z could be matched by Z. The action specified in Z will be taken, not X. The packet classification property would be altered. In this case, X can not be merged. Figure 5 illustrates both cases.

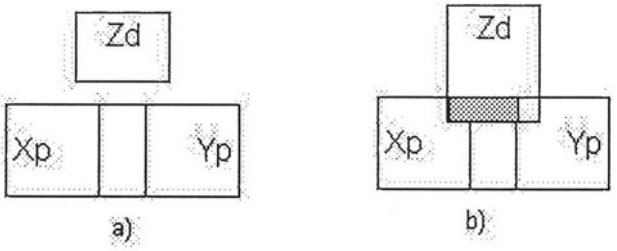

Figure 5. Merge Check. a) Xp can be merged with Yp. b) Xp may not be merged, if Xp is merged, any packet that falls in the shaded region could be denied by Z

4.1 ACL analysis, optimisation and query

The first algorithm, *optimise,* merges adjacent or overlapping ACEs, detects, reports and resolves inconsistencies, and removes redundancies. After *optimise*, an ACL is mapped to a simpler form without any modification to the packet classification properties. This simplified representation is easier for administrators to understand, and can replace the original ACL on the router. During *optimise*, redundancies and inconsistencies are identified. *optimise* can be executed offline or incorporated directly in to the router's command line interface. In this case, an administrator would receive interactive feedback as he is configuring the ACLs, thus detecting security risk at the earliest possible time. The following notations are used: I_p and I_d are two interval trees that store permit rules and deny rules respectively. S_p and S_d are sets that stores the query returned from each tree. The variable *a* defined in line 4 serves as an index to the appropriate interval tree or query set. For example, if *a* is permit, then S_a refers to S_p, $S_{!a}$ refers to S_d.

Algorithm *optimise* (L)
Input. An ACL
Output. An optimised ACL
1. for each rule *y* in L
2. $S_p = I_p$.query (*y*); // return permit rules that *intersect* or adja*cent y*
3. $S_d = I_d$.query (*y*); // returns deny rules that *intersect* or *adjacent y*
4. let *a* = action type of *y* // permit or deny
5. if any rule in S_a contains *y*
6. continue // redundancy rule 1,2
7. if any rule in $S_{!a}$ contains *y*
8. continue // inconsistency rule 1,2
9. for each rule *x* in S_a that is contained by *y*
10. if (there does not exits a rule *z* in $S_{!a}$, where *z* is between *x* and *y*

11. and x intersects z and x does not contain z)
12. I_a.remove (x) // redundancy rule 3,4
13. for each rule x in S_a that overlaps or is adjacent to y
14. if (there does not exits a rule z in $S_{!a}$, where z is between x and y
15. and x intersects z and x does not contain z)
16. I_a.remove (x)
17. y = merge(x, y) // merge rule 1,2
18. I_a.insert (y);
19. build L_{result} using I_p and I_b
20. return L_{result}

This algorithm is organised to clearly illustrate each consistency, redundancy and merge condition. The actual implementation can be less verbose. I_p and I_d are empty initially. Line 5 to 8 checks the simple cases first. Line 9 to 12 checks for redundancy of a previously defined rule. Line 13 to 17 merges a previously defined rule with the current one (Note: in order to find the adjacent rules, y needs to be expanded by 1 in all dimensions before performing the query in line 2 and 3). Finally, if line 18 is reached, this new rule is inserted into the appropriate interval tree. A warning or informational message can be issued to the administrator each time a redundancy and inconsistency is detected, or when a merge happens. Each rule stores an *order number* that corresponds to the original order in the ACL. The *order number* is maintained in throughout the optimisation process. The merge operation on line 17 does not change y's order number. The new optimised rule list can be generated by a simple traversal of the I_p and I_d tree and return the result according to the original rule order. If each query returns results proportional to n, where n is the size of original ACL, this algorithm requires $O(n^2)$ time. However, such an ACL configuration is impractical to say the least. We can safely assume that the query returns a constant size set. Therefore, the two for loops in line 9..12 and 13..17 runs in constant time. The running time of this algorithm is thus dominated by the query time, or $O(n\log^d n)$.

It is often insightful to view an ACL as a sequence of permits followed by an implicit deny-all rule, or vice versa. Arbitrary interleaving of deny and permit rules can be difficult to decipher. This is especially true if the ACL is modified or expanded over time by different administrators. The following algorithm *Filter* maps ACL with interleaving permit and deny rules to an ACL with one type of rule only, followed by a broad rule to cover the opposite action at the end. We assume I_p and I_d has been built already using *optimise*.

Algorithm *filter* (I_p, I_d, a)

Input. Two rule trees of an optimised ACL and a filter based on rule action (permit or deny)
Output. Optimised ACL containing only rules with action a (assume final broad implicit rule)

```
1. for each rule y in I_a
2.    S_!a = I_!a.query (y)          // return all rules intersect y
3.    for each rule x in S_!a
4.       remove x if it is defined after y
5.    if any rule in S_!a contains y
6.       continue
7.    queue.push_back (y)
8.    while queue is not empty
9.       t = queue.pop()
10.      if (t is disjoint from all rules in S_!a)
11.         L_result.append (t);
12.      else
13.         if (there exist a rule x in S_!a that intersects t)
14.            queue.push (rules generated by splitting t around x)
15. return Optimise (L_result)
```

If the filter *a* is permit. *filter* algorithm iterates through all the permit rules, for each permit rule, the set of intersecting deny rules that precedes it is found (line 2..4). Next, if this permit rule is contained by any of the deny rules that precedes it, we do not add it to L_{result}. If the ACL has been optimised correctly already, this should not happen. Line 7 to 14 splits the permit rule around the deny rules. A queue is used to store the temporary split permit rules. The result is a set of new rule that is pushed onto the queue. This new permit rule is not added to L_{result} unless it does not intersect with any preceding deny rules. We rely on the final optimisation step in line 15 to merge any rules that becomes adjacent or overlap as the result of the splits. The running time of this algorithm is dominated by the query, or $O(nlog^d n)$, where n is the size of ACL.

Next, we present an algorithm *joint* that finds common set of rules within an ordered set of ACLs. *joint* can be used to examine the set of ACL along a path from a source to destination. The result would inform the administrator what type of traffic will be permitted or denied. The user can pass in a filter to indicate the desired output format. For each ACL we assume its corresponding I_p and I_d are computed already.

Algorithm *joint* ($L_1..Ln$, *a*)
Input. A set of optimised ACL and a filter based on rule action (e.g., permit or deny)

Output. An optimised ACL with rules of type *a*
1. for each ACL *l* in $L_1..Ln$
2. $l = filter\ (l, a)$
3. $I_{result} = L_1.I_a$
4. for each *l* in $L_2..L_n$
5. if (a is permit)
6. I_{temp}.clear()
7. for each rule *y* in *l*
8. $S_p = I_{result}$.query (y) // return all rules *intersect* y
9. for each rule x in S_p
10. I_{temp}.insert (intersection of x and y)
11. $I_{result} = I_{temp}$;
12. else
13. simply insert all rules in *l* to I_{result}
14. return *optimise* (L_{result})

Filtering is performed first to simplify the actual joint operation. If the filter action is permit, all rules will be mapped to permit rules. A packet that traverses from source to destination has to be permitted by all the ACLs along the way. Therefore, to joint ACLs in this sense is to find the intersection of permit rules between ACLs. The common set that remains represent the meta-ACL profile, it specifies which traffic type will successfully flow from source to destination along the path. Line 5 to 11 joints two ACLs in the permit case. The result is used for the next ACL. If the filter action is deny, we can simply insert all deny rules. We again rely on the final optimisation step in line 14 to reduce the number of rules. Each joint operation takes $O(mlog^d m)$ time, where m is the size of ACL. The joint algorithm takes $O(n*mlog^d m)$, where n is number of ACLs.

A *point query* operation to match a packet condition can be implemented by performing a query on the internal I_p and I_d tree. The rule with the lowest order number is the matching rule. A *range query* operation can intersect multiple permit/deny rules, so a list of all intersecting rules should be returned.

4.2 ACLA (ACL Analyser)

We developed the *ACLA* library that implements all of the above operations. It contains a set of generic algorithms such as *optimise*, *filter*, *joint* and *query*, and containers such as the multidimensional interval tree. The actual implementation makes heavy use of the Standard Template Library (STL) and follows its generic programming concepts. The client simply needs to implement the relations specified in Definition 1, which is

typically vendor specific. The relation operators are passed to the library as a template argument. The generic algorithms will work exactly the same. Clients can experiment with their own container classes by overriding the default interval tree structure. For example, the client can choose an efficient external-memory data structure if the ACL size is large.

5. CONCLUSION

In this paper we presented a novel approach to ACL analysis. We modelled ACEs in an ACL using well-defined constructs, which leads to precise mathematical definition of their relationships. Based on these sets of primitive relation and operations, we proposed a set of new algorithms, *optimise*, *filter*, *joint,* and *query*. A multidimensional interval tree is used to facilitate efficient point and range queries. We developed the *ACLA* library based on the algorithms and data structures presented in this paper. The framework proposed in our work should greatly simplify the network administrator's task of implementing and verifying corporate security policy.

It would be interesting to incorporate ACL analysis in to the router's command line interface. The administrator would get immediate feedback on any redundancies or inconsistencies as he is configuring the ACL. The router can also optimise the configured ACLs to increase packet classification speed.

A significant extension currently under development is to automatically derive global security policies based on the existing network topology and router configurations, and test for conformance to the corporate security policies. Our current framework requires the user to define the path a packet traverses. If dynamic routing is used with complex network topology, this information can be difficult to determine. We are currently investigating alternative approaches.

References

[1] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A Novel Firewall Management Toolkit. *IEEE Symp. on Security and Privacy*, Oakland, CA 1999.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509-517, 1975.

[3] J. L. Bentley. Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon Univ., Pittsburgh, PA, 1977.

[4] J. L. Bentley. Decomposable searching problems. *Inform. Process. Lett.*, 8:244-251, 1979.

[5] Cisco Secure Policy Manager 2.2, 2000. http://www.cisco.com/warp/public/cc/pd/sqsw/sqppmn/.

[6] Cisco Secure Scanner 2.0, May, 1999. http://www.cisco.com/univercd/cc/td/doc/pcat/nssq.htm.

[7] H. Edelsbrunner. A new approach to rectangle intersections, part I. *Int. J. Computer Mathematics*, 13:209-219, 1983.

[8] H. Edelsbrunner. A new approach to rectangle intersections, part II. *Int. J. Computer Mathematics*, 13:221-229, 1983.

[9] M. Freiss. *Protecting Networks with SATAN*. OReilly & Associates, Inc., 1998.

[10] J. D. Guttman. Filtering postures: Local enforcement for global policies. *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA 1997.

[11] S. Hinrichs, "Policy-Based Management: Bridging the Gap". *Annual Computer Security Applications Conference*. Scottsdale, AZ, 1999.

[12] Interconnecting Cisco Network Devices: Student Guide, 1999.

[13] Internet Security Systems Internet Scanner, 2000. http://documents.iss.net/literature/InternetScanner/is_ps.pdf.

[14] D. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625-658, 1990.

[15] A. Mayer, A. Wool, and E. Ziskind. Fang: A Firewall Analysis Engine. *IEEE Symp. on Security and Privacy*, Oakland, CA 2000.

[16] Nessus 1.0.6, Nov. 2000. http://www.nessus.org/.

[17] Network Associates Cybercop Scanner, 2000. http://www.pgp.com/products/cybercop-scanner/default.asp.

[18] J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):257-276, 1984.

[19] SATAN, Apr. 1995. http://www.cs.ruu.nl/cert-uu/satan.html