EFFICIENT DATA STRUCTURES FOR LOCAL INCONSISTENCY DETECTION IN FIREWALL ACL UPDATES

S. Pozo, R. M. Gasca, F. de la Rosa T.
Department of Computer Languages and Systems,Computer Engineering College, University of Seville
Avda. Reina Mercedes S/N, 41012 Sevilla, Spain
{sergiopozo,gasca,ffrosat}@us.es

Keywords: diagnosis, consistency, conflict, anomaly, firewall, acl, ruleset, update

Abstract: Filtering is a very important issue in next generation networks. These networks consist of a relatively high number of resource constrained devices and have special features, such as management of frequent topology changes. At each topology change, the access control policy of all nodes of the network must be automatically modified. In order to manage these access control requirements, Firewalls have been proposed by several researchers. However, many of the problems of traditional firewalls are aggravated due to these networks particularities, as is the case of ACL consistency. A firewall ACL with inconsistencies implies in general design errors, and indicates that the firewall is accepting traffic that should be denied or vice versa. This can result in severe problems such as unwanted accesses to services, denial of service, overflows, etc. Detecting inconsistencies is of extreme importance in the context of highly sensitive applications (e.g. health care). We propose a local inconsistency detection algorithm and data structures to prevent automatic rule updates that can cause inconsistencies. The proposal has very low computational complexity as both theoretical and experimental results will show, and thus can be used in real time environments.

1 INTRODUCTION

A wireless ad hoc network is a collection of autonomous nodes that communicate with each other by forming a multihop network and maintaining connectivity in a decentralized manner. The network topology is in general dynamic.

In these networks, before and after the authentication step, there are attacks that can be performed with the aim of degrading network performance. In traditional networks, layer 3 firewalls reduce the impact of these attacks However, the firewall concept must be adapted (Fantacci, 2008): filtering must be implemented at each node of the network.

An Access Control List (ACL) is an ordered list of condition/action rules. The *condition* part of the rule is a set of condition attributes or selectors. In layer 3 firewall domain, the condition set is typically composed of five elements, which correspond to five fields of a packet header (Taylor, 2003). In this paper, we are interested in consistency problems in next generation networks (Al-Shaer, 2004) (Pozo2, 2008). Due to real-time frequent ACL updates, inconsistencies must be detected and automatically managed very fast.

This paper focuses in the design of specialized data structures and an algorithm to efficiently solve this problem. The algorithm is capable of handling full ranges in rule selectors without doing rule decorrelation, range to prefix conversion, or any other pre-process. Results are returned over the original, unmodified ACL. To the best of our knowledge, there are only two algorithms that do not decompose the ACL: the trivial one (which is worst case $O(f^2)$ time complexity); and a modification over it (Pozo3, 2008), which only improves the average and best cases.

The paper is structured as follows. In section 2, we briefly analyze the internals of the consistency management problem in firewall ACLs. In section 3 we explain the methodology followed to solve the problem. In section 4, we give experimental results with real ACLs. In section 5 we review related works. Concluding remarks are given in section 6.

Table 1. Example ACL.

Priority/ID	Protocol	Source IP	Src Port	Destination IP	Dst Port	Action
R0	tcp	192.168.1.5	any	*.*.*.*	80	deny
R1	tcp	192.168.1.*	any	*.*.*.*	80	allow
R2	tcp	*.*.*.*	any	172.0.1.10	80	allow
R3	tcp	192.168.1.*	any	172.0.1.10	80	deny
R4	tcp	192.168.1.60	any	*.*.*.*	21	deny
R5	tcp	192.168.1.*	any	*.*.*.*	21	allow
R6	tcp	192.168.1.*	any	172.0.1.10	21	allow
R7	tcp	*.*.*.*	any	*.*.*.*	any	deny
R8	udp	192.168.1.*	any	172.0.1.10	53	allow
R9	udp	*.*.*.*	any	172.0.1.10	53	allow
R10	udp	192.168.2.*	any	172.0.2.*	any	allow
R11	udp	*.*.*.*	any	*.*.*.*	any	deny

2 CONSISTENCY MANAGEMENT IN FIREWALL ACL UPDATES

Let ACL_f be a layer 3 firewall ACL consisting of f rules, $ACL_f = \{R_1,...R_f\}$. Consider $R_j \in ACL_f =< H, Action >, H \subseteq Z, 1 \le j \le f$, $Z = protocol \times srcIP \times srcPrt \times dstIP \times dstPrt$ as a rule, where $Action = \{allow, deny\}$ is its action. A selector of a firewall rule R_j is defined as $R_j[k], k \in H, 1 \le j \le f$, Some of these selectors can be expressed as naturals, and others as both naturals and intervals of naturals (an analysis of the supported syntaxes for firewall selectors is also available (Pozo1, 2008)). Firewall ACLs can be trivially divided in two disjoint sets, one composed of rules with allow action (ACL_{allow} with size m), and the other with deny action rules (ACL_{deny} with size n), with $ACL_f = ACL_{allow} \cup ACL_{deny}$. In real-life firewall ACLs, $m<<n$ or vice-versa. An example ACL is presented in Table 1.

Definition 1. Inconsistency between two rules. Two rules $R_i, R_j \in ACL_f$ are *inconsistent* if and only if the intersection of *each of all* of their k selectors $k \in H, H \subseteq Z$ is not empty, and they have different actions, *independently of their priorities.* The inconsistency is considered to be a fault if an administrator identifies the behaviour of the executed ACL as being causing undesirable effects (or having errors).

There are three basic update operations: insertion, removal or modification of one or more rules. These operations need an analysis in order to know if they can cause an inconsistency. This analysis has been provided in other works (Al-Shaer, 2004) (Pozo3, 2008). It is assumed in the paper that a collection of these operations over an ACL is always executed in sequence. It is also assumed that the initial node rule set (if any) is consistent.

3 INCONSISTENCY DETECTION PROCESS

The process is based on divide and conquer algorithm. We depart from the trivial ACL_f decomposition in ACL_{allow} and ACL_{deny}. For the rest of the section and in order to simplify explanations, it is assumed that $n<<m$ and that R_d (a rule that is going to be inserted in the node ACL) has *deny* action. If R_d has *allow* action and/or $n>>m$, explanations are analogous. The proposed algorithm returns all rules in ACL_{allow} that are inconsistent with R_d, during an update operation.

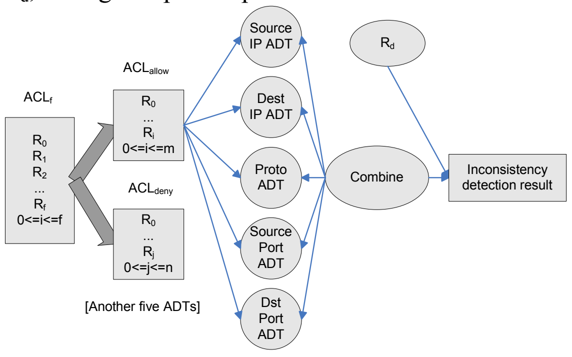

Figure 1. Proposed inconsistency detection process (considering updates).

One of the main ideas of our approach is to use a specialized abstract data type (ADT) to store the set of all selectors of the same type in ACL_{allow} (i.e. one ADT to store protocols used in all rules, two ADTs to store the source and destination IPs used in all rules, and another two ADTs to store source and destination ports). In fact, a duplicate of these ADTs is necessary in order to store ACL_{deny} selectors (if R_d has *allow* action), but as we have noted before, the process is analogous. Process is depicted in Fig. 1.

Three main operations are needed in these ADTs: search, insert, remove. The three operations must be fast enough, since all of them are used for any of the three ACL update operations. ADT population is done before deployment (off-line).

3.1 ADT for Protocol Number Selector

Attending to the exhaustive analysis of real firewall languages presented in another work (Pozo1, 2008) the protocol selector only admit 8-bit natural numbers and the wildcard, *. Although symbolic names are also possible, they can be converted to naturals using IANA protocol numbers (RFC5237). An important fact is that no ranges are allowed in the syntax of the selector, and thus search is a trivial operation for the ADT: to find a non-empty intersection with a protocol number (the one of R_d) there are only two possible valid values in the ADT: '*' (and thus R_d intersects with all rules of the ADT, that is all rules in ACL_{allow}); or exactly the same value.

To store the association <*Protocol number, Rule ID*> we propose to use a hash table with perfect and minimal hash function with protocol as the key, and the rule IDs as value. Hash tables (Cormen, 2001) have *O(1)* (constant) time complexity for insertions, removals, updates and search operations if a perfect hash function is used.

However, hash tables do not allow duplicate keys to be inserted. This issue is resolved by grouping all rules that share the same protocol number (the same key). In this case, the associated value to the key is a set containing the rule IDs of all rules that have the key value as the vale of their protocol selector. However, as removal of values could be inefficient in this way (a hash lookup plus a search in the list of rule IDs), the list is replaced by a fixed-size bit set of size *m* (the size of ACL_{allow}). Each position of the bit set represents one of the *m* rules in ACL_{allow}.

3.2 ADT for Port Number Selectors

Port selectors admit 16-bit natural numbers, double-ended closed natural intervals, and '*' (Pozo1, 2008).

There are two well-known 2D problems in computational geometry that solve similar searches: first, given a set of data points (port numbers) and a query rectangle (port interval), give all the points that are inside the rectangle (this is the orthogonal *range search problem*); second, given a set of (possibly intersecting) data rectangles (port intervals) and a query point (port number), give all rectangles that intersect the query point (this is the *stabbing problem*). These two 2D problems can be reformulated into 1D space, where rectangles are intervals and points are only represented by one coordinate. In 1D, these problems are called *1D range search problem* (Cormen, 2001) and *overlapping interval search problem* (Edelsbrunner, 1983), respectively. Fortunately, specialized data structures for 1D and 2D problems that give optimal bounds (in time and space) solutions to these two problems exist. In the particular case of 1D, the *Interval Tree* (Edelsbrunner, 1983) (Cormen, 2001), or *ITree*, is the selected ADT because it has optimal bound for the 1D problem (in time and space).

Fortunately, our port number and interval search problems can trivially be reformulated to *range search* and *overlapping interval search* problems, as port numbers can be represented as points in a 1D plane, and port intervals can be presented as lines in the same 1D plane.

Space complexity is linear with the number of rules in ACL_{allow}, m. However, in our implementation duplicate intervals or points are not allowed, and thus space complexity is reduced in a constant factor. Update time is in *O(logm)*. Query time is in *O(logm + L)*, where *L* is the number of returned results (a constant factor). Thus, instantiation is in worst case *O(m*logm)*, one

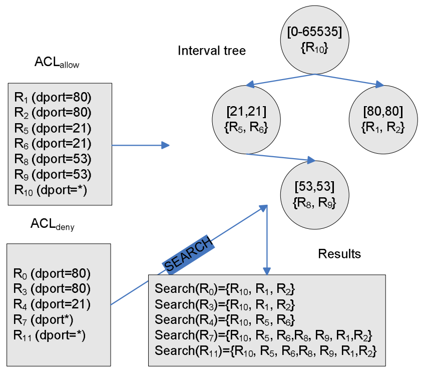

Figure 2. Interval tree for destination port selector of Table 1 example.

insertion for each rule in ACL_{allow}.More details are available in (Edelsbrunner, 1983) (Cormen, 2001).

The result of the search operation over the ITree with a port number or interval of the rule R_d, is the union of all bit sets associated to port values in the ITree which intersect the given port of R_d, or a bit set with all bits set to '1' if the given port of R_d is '*'. Fig. 2 presents the ITree associated to Table 1 example (destination port).

3.3 ADT for IP Address Selector

IP address selectors admit 32-bit host IP addresses plus CIDR format, and '*' (Pozo1, 2008). Symbolic names are converted to octets.

As with previous cases duplicates are not allowed (bit sets are used again). Thus, the result of the search operation must be a bit set with positions set to '1' for all rule IDs of ACL_{allow} which have an intersecting IP with the given in the rule R_d.

If a comparison between this selector and the previous ones is made, some similitude and differences arise. On one hand, an IP is formed by four octets, each one being an 8-bit natural; but on the other hand, a the search operation must use the netmask address of the IPs stored in the ADT (Let $IP_1/CIDR_1$ and $IP_2/CIDR_2$ be two IP addresses, if $CIDR_s$ is the shortest of the two netmasks, then the intersection of IP_1 and IP_2 is not empty if $IP_1\&CIDR_s=IP_2\&CIDR_s$.). Thus, we propose the design of a completely new and specialized ADT to store IP addresses. Note that valid network IP addresses have CIDR values between 1 and 30.

The tree is formed by four levels. For each node, 255 children are possible at most (0-254). These children values of each node (octets) are stored in a hash table (perfect and minimal hash is possible again). The association <*Node octet, Children octets*> is called a node-value.

No repetitions of node-values are allowed in an IP Tree, except for leaves. Leaf nodes must also store information regarding CIDRs and rule IDs, where the CIDR represents the CIDR of the IP whose insertion ended in that leaf, and where each CIDR value has an associated set of rule IDs (as a bit set) to associate an inserted IP/CIDR to one or more rule IDs (if there are repetitions). The <*CIDR, RuleID Bit set*> pair is stored as a hash table (perfect and minimal hash again, since there are only 30 possible CIDRs).

Insertions are done traversing the tree from top to bottom. First, the IP/CIDR address to be inserted, R_d, is decomposed in its four natural octets plus the

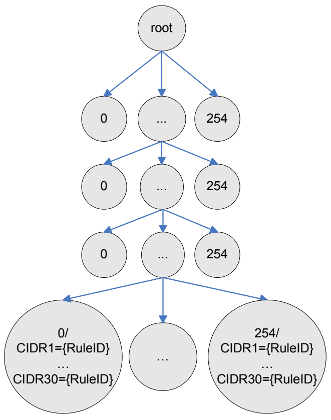

Figure 3. IP Tree general structure.

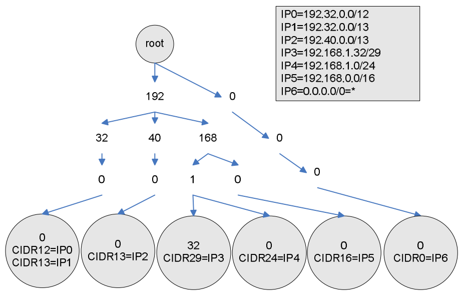

Figure 4. IP Tree example for network IPs.

CIDR value: *o1.o2.o3.o4/cidr.* Then, the root node children hash table is asked in order to know if *o1* is already in the tree. If it is, the next step is to traverse to the second level through the found node. If not, a new node with value *o1* is inserted in the root node children hash table. These same is done for *o2, o3*, and *o4*. Once at the last level, if *o4* has been found, a check is launched for the CIDR data stored in the leaf <*CIDR, Rule ID Bit set*> hash table using *cidr* value of the IP. If *cidr* value is found, the bit corresponding to the ID of the inserted IP is set to '1'. If not, a new CIDR value is created with its corresponding bit set. Thus, the insertion of a new IP consists, in the worst case, of three constant time searches in perfect hash tables, plus a *O(1)* search in a leaf perfect hash table, resulting in *O(1)* worst case time complexity.

The search operation is very similar to insertion one. Note that in order to know if two IP addresses intersect, the application of the shortest netmask of the two IP addresses is necessary, as has been pointed at the beginning of the subsection. However, the ACL_{allow} IP Tree contains the IPs of the *m* rules in ACL_{allow}. Thus, the application of all netmasks of the IPs in the IP Tree which are smaller than or equal the CIDR of the given R_d IP address is necessary (at most 30 netmasks). The result of the

Table 2. Performance evaluation.

ACL Size	%Deny Rules	ACL_{deny} Size	ACL_{allow} Size	No. Inconsist.	Trivial (ms)	Optimized Trivial [8] (ms)	Proposal remove/search =detection(ms)	Proposal insert (ms)	Proposal update (ms)	ADT build (ms)
50	28,21	11	39	37	0.23	0.19	0.05	0.1	0.15	1.41
144	30,91	34	110	108	0.66	0.58	0.07	0.14	0.21	3.94
238	66,43	95	143	231	1	0.75	0.06	0.12	0.18	6.52
450	34,73	116	334	422	2.17	1.77	0.08	0.16	0.24	14.41
900	14,8	116	784	871	5.2	4.42	0.09	0.18	0.27	31.65
2500	6,97	163	2337	3349	15.58	13.2	0.19	0.38	0.57	128.51
5000	1,98	97	4903	4903	32.6	28.28	0.34	0.68	1.02	276.75
10611	2,05	213	10398	11746	72.87	60.94	0.96	1.92	2.88	539.67

application of these netmasks is a set of (at most) 30 network IPs. Now, a search operation for each of this IPs is launched. The search operation follows the same algorithm used for insertions, but taking the list of rule IDs associated to the CIDR of the leaf which coincide with the CIDR used for the search. The result of the search operation over the IP Tree with an IP address of the rule R_d, is the union of all bit sets associated to IP addresses in the IP Tree which intersect the given IP address of R_d (e.g. the result of the –at most- 30 searches), or a bit set with all bits set to '1' if the given IP address of R_d is '*'.

The general structure of an IP Tree, as well as an example ACL and an IP Tree of network addresses are presented in Figures 3 and 4 respectively.

3.4 Combination of Search Results

Using the calculated worst case time complexities of the search operations for the five selector and, by the sum of the rule, the combined search time for five selectors is in worst case $O(1)+2O(1)+2O(logm)=O(logm)$. The first factor is the time associated to searching in a hash table, the second is the two searches in an IP Tree, and the last one is the two searches in an ITree.

The obtained results are five bit sets with positions set to '1' for intersecting rule IDs. However, from the inconsistency definitions, *all* selectors must overlap for a rule to be inconsistent with other(s). Thus, the composition of this result is somewhat trivial: the intersection of the five bit sets.

As its name indicates, a bit set is an ADT whose main purpose is to store bit elements. The intersection of the five bit sets is a linear time operation with the number of rules in ACL_{allow}, m, which is also the size of the bit sets. However, note that although the problem is linear, logical operations over bit arrays are very efficient, as they are instructions that can be executed in one machine cycle over 128 bit registers using special multi-register multimedia instructions. This yields a severe problem reduction by a big constant, $k=128,$ in time (with no space penalty).

Thus, time complexity of the full search process (which is equivalent for insertion), including the combination operation, is in worst case $O(logm+m/k),n=m=f/2 \rightarrow O(log(f/2))+O((f/2)/k), m/k>logm \rightarrow O((f/2)/k) \rightarrow O(f/2k), k=128.$

As has also been shown, the space needed in the process is linear with the number of rules in ACL_{allow} plus some bit sets (the space needed to store the bit sets is negligible).

Note that a number of optimizations have been introduced in order to stop the search (in shortcut) if a zero bit set is returned from any of the search operations, because if a selector of R_d is consistent with the same selector of all the rules of ACL_{allow}, then R_d is consistent by definition, and no more searches for the rest of selectors are needed. Thus best and average cases time complexity are achieved when there a lot of selector repetitions in ACL_{allow} (and thus ADTs are very small, reducing the time needed for search operation in the ITree to near a constant), when $n<<m$, and when R_d is consistent (there are no combination of results), resulting in $O(logn),logn \approx constant \rightarrow O(1).$

Removals of values in the ADTs have the same worst case time complexity than searches (minus the combination step, $O(logm)$), and updates are a removal and an insertion (or search).

4 EXPERIMENTAL RESULTS

The proposed process has been tested with real firewall ACLs (Table 2). Experiments were performed on a Java Sun JDK 1.6.0_10 32-bit HotSpot VM, on a machine with AMD Geode LX800 (500MHz) and 256Mb RAM DDR400. Execution times are in milliseconds.

The most important fact is regarding time needed for update. As can be seen in Table 2, the final time for updating an ACL is much faster in our proposal (note that search operation needs a final combine step, and thus represents the more costly update operation). The difference between our proposal and the trivial or the optimized ones is dramatic. If several update operations, *op*, are going to be done over the ACL, these time results must be multiplied by *op*, since they are done in sequence.

However, ADT build times are very high, compared with time needed for update operations (ACL_{allow} plus ACL_{deny} times have been measured here). Fortunately, ADTs can be instantiated only once, and then be maintained. Thus, build time should be taken as the start-up time, and needs to be amortized. Our proposal begins to be faster than the optimized trivial algorithm from 8-9 sequential updates and up (for all ACL sizes). Thus, it is possible to wait to 8-9 update operations or more and execute them in a burst. Effectiveness of this approach depends on ACL update frequency.

5 RELATED WORKS

Baboescu et al. (Baboescu, 2003) provide algorithms to detect inconsistencies in router filters that are 40 times faster than $O(f^2)$ the trivial one for the general case of *k* selectors per rule. They also provide modifications to its algorithms and data structures for rule updates. It experimentally improves other previous works of detection algorithms. However, they preprocess the ACL and convert selector ranges to prefixes (Srinivasan, 1998). The range to prefix conversion technique could need to split a range in several prefixes and thus the final number of rules could increase over the original ACL. This kind of conversion could be inefficient: in the worst case, a range covering *w-bit* port numbers may require *2(w-1)* prefixes (Taylor, 2003). Furthermore, results are given over a modified ACL.

Other research woks (Al-Shaer, 2004) (Pozo2, 2008) complemented the diagnosis process with a characterization of the faults. However, minimal diagnosis and characterization is NP.

6 CONCLUSIONS

In this paper we have showed a divide-and-conquer process, ADTs, and algorithms, capable of solving the inconsistency detection problem during an ACL update operation in worst case linear complexity divided by a big constant. The process is *O(1)* in best and average cases (no inconsistency found). Experimental results that support our theoretical complexity analysis have been provided.

ACKNOWLEDGEMENTS

This work has been partially funded by Spanish *Ministry of Science and Education project* under grant DPI2006-15476-C02-01, and by FEDER.

REFERENCES

Al-Shaer, E., Hamed, H. Modeling and Management of Firewall Policies". IEEE eTransactions on Network and Service Management (eTNSM) Vol.1, No.1, 2004.

Baboescu, F., Varguese, G. "Fast and Scalable Conflict Detection for Packet Classifiers." Elsevier Computers Networks (42-6) (2003) 717-735.

Cormen, T., Leiserson, C., Rivest, R., Stein, C. Introduction to Algorithms, McGraw-Hill, 2001.

Edelsbrunner, H. A new approach to rectangle intersections, Part II. International Journal on Computational Mathematics. Vol.13, pp. 221-229, 1983.

Fantacci, R., Maccari, L., Neira, P., Gasca, R. M. "Efficient Packet Filtering in Wireless Ad Hoc Networks". IEEE Communications Magazine Vol.46, No.2, 2008.

Pozo1, S., Ceballos, R., Gasca, R.M. "AFPL, An Abstract Language Model for Firewall ACLs". 8th International Conference on Computational Science and Its Applications (ICCSA). Perugia, Italy. Springer-Verlag, 2008.

Pozo2, S., Ceballos, R., Gasca, R.M. "Improving Computational Complexity of the Inconsistency Characterization Problem in Firewall Rule Sets". International Conference on Security and Cryptography (SECRYPT). Porto, Portugal. INSTICC Press, 2008.

Pozo3, S., Ceballos, R., Gasca, R.M. "Fast Algorithms for Local Inconsistency Detection in Firewall ACL Updates". 1st International Workshop on Dependability and Security in Complex and Critical Information Systems (DEPEND). Cap Esterel, France. IEEE Computer Society Press, 2008.

Srinivasan, V., Varguese, G, Suri, S., Waldvogel, M. "Fast and Scalable Layer Four Switching." Proceedings of the ACM SIGCOMM conference on Applications, Technologies, Architectures and Protocols for Computer Communication, Vancouver, British Columbia, Canada, ACM Press, 1998.

Taylor, David E. Survey and taxonomy of packet classification techniques. ACM Computing Surveys, Vol. 37, No. 3, 2005. Pages 238 – 275.