Improving Distributed Firewalls Performance through Vertical Load Balancing

Olivier Paul

GET/INT, LOR Department,
91000 Evry, France
Olivier.Paul@int-evry.fr

Abstract. In this paper we present an extension to an existing hash based packet classification technique in order to improve its performance in a distributed network access control environment. We show that such architecture can be modified so that flow states can be kept in a distributed fashion thus reducing the space needed for packet filtering in each component of the architecture. We also show how such approach can, in some cases, improve the overall time complexity of packet filtering operations by reducing the number of packet classification operations.

1 Introduction

As network architectures become more complex and interconnected, the number of interconnection devices regularly increases. At the same time, the number of network devices with security features in general and access control capabilities in particular is also rising. These capacities are widely used in companies in order to partition networks and limit the ability of users to interact with each others. The problem of automating and optimizing the configuration of such distributed access control architectures has recently raised a lot of interest. However most techniques have either focused on the optimization of distributed access control policies independently from packet classification schemes or focused on optimizing packet classification schemes independently from the distributed nature of distributed firewalls. In this paper we explore the optimization of a specific (while popular) packet classification scheme [4] in the context of a distributed access control architecture. We show that such a scheme can widely benefit from a distributed implementation.

Section 2 provides an overview of existing schemes for access control performance improvements. Section 3 presents our proposal to improve an hash based stateful packet classification scheme in a distributed access control environment. We address dimensioning issues in section 4 and perform a comparison with the non distributed approach. We also briefly present an implementation of our architecture with the *ns* simulator. Finally section 5 summarizes the pro and cons of our proposal and highlights possible extensions.

2 Related Work

The management of distributed access control architectures has generated a lot of work in the last few years [1][2]. The goal is usually to automate the configuration of a set of access control devices under a single administrative domain. An interesting aspect regarding these techniques is that they acknowledge the fact that current access control architectures are usually made of several "layers" thus leading to the traversal of multiple redundant filters for most communications. Although this fact is not completely new, we believe that the integration of access control functions in a wide range of network devices as well as a better understanding of insider risks has led to a wide increase in this redundancy over the last decade. Our architecture takes advantage of this redundancy to distribute parts of the access control functions over several filters.

Load balancing is another well known technique to share processing load between several components. However the load is usually shared by a front device (e.g. DNS server, load balancer) among several components. Compared to such approaches our architecture may be considered as a load balancing scheme where the load balancing occurs between "serial" devices instead of occurring between "parallel" devices as usual, the distribution function being implemented in each component. Consequently our scheme does not necessitate more devices than those available in a network while providing similar benefits in term of processing performance. On the other hand load balancing architectures often provide fault resilience which is not our case.

Tunneling techniques can be used to bypass packet filters by short-cutting packet filtering functions or by obfuscating necessary filtering information. A similar result may be reached by disabling access control or state maintenance functions or limiting their use to specific communications. However they require some sort of signaling in order to set up a context in each device which can induce strong performance penalties in particular in the case of short flows. Moreover choosing the right source and destination for tunnels in order to bypass selected filtering components and making sure that the filtering policy is still enforced can be a difficult task.

3 A state distribution scheme

Stateful packet classification [3] is usually performed in two phases. The "dot" classification phase occurs first. The goal is to find if the packet can be matched to an existing connection. When it can, the action specified for the packet can be directly retrieved from the connection context. In the other case, the whole classification policy has to be searched for a relevant rule using "rectangle" classification. "dot" classification is distinguished from "rectangle" classification because it is usually much faster since it deals with specific values while "rectangle" classification deals with ranges thus leading to more complex treatments. As a result keeping a state about connection brings benefits from security and performance points of view. However "dot" classification is only possible if a connection context has been previously set up using "rectangle" classification. Several methods exist today to store such a context,

however the method we focus on here is an hash based answer. The hash function is involved in two operations.

- After "rectangle" classification, a hash value is computed over a limited number of fields (i.e. protocol, source and destination addresses, source and destination ports). This hash value points to a bucket that will be used to store the connection state.
- For "dot" classification, the same fields are extracted from each packet and used to compute a hash value that points to the bucket used to store the connection state.

In the case where no corresponding connection is found (either because the bucket is empty or because no matching connection is found), "rectangle" classification has to be performed in order to fill the hash table with the corresponding connection.

Keeping a state in a single hash table can result in high collision rates thus leading to a large number of states kept in a single bucket and an increased "dot" classification time. A way to solve this problem is to use a multiple hash functions [4]. The state hash table of size m is divided into n smaller hash tables of size m/n where each table uses a separate hash function. n hash values are now computed over the packet content and point to a specific bucket in each hash table. In order to limit the number of concurrent states in each bucket, one selects the hash table including the bucket with the lowest depth. This approach however has two main drawbacks:

- n hash functions have now to be computed for each packet after rectangle classification and for hash classification.
- When performing hash classification, one bucket in each n tables has now to be searched in order to find the relevant state.

Another point we want to highlight is that contexts are usually memory consuming. For example keeping one state in pf requires roughly 150 bytes of memory which means that keeping states for a 10Gbits/s Ethernet links would require several gigabytes (We later detail how we reach these numbers).

3.1 State distribution problem

Let's now consider a distributed filtering architecture where several filters are available on the path between any source and destination. Let's also assume that each filter implements the basic hash classification process described earlier. Each filter has to compute at least one hash value over the packet content in order to perform access control functions. Our goal is to take advantage of these hash functions to simulate the behavior of a single packet filter implementing the multiple hash functions scheme. This would allows us to keep the improvements brought by a multiple hash function scheme while avoiding the corresponding drawbacks.

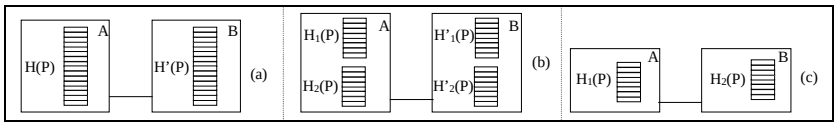

Fig. 1. Various distributed hash based filtering scenario. (a) represents two single hash filters. (b) represents two multiple hash filters. (c) represents our proposal

Figure 1 illustrate the difference between our proposal and other hash based "dot" classification methods in a distributed environment. (c) represents the suggested approach in which tables that were previously included in a single device are now scattered over several devices thus reducing the memory requirements for each filter.

Although this approach may seem superior it bears several problems:

1. The selection algorithm cannot be used as is since there is no way for device A to know how many packets are stored in device B. Packets stored in B may come from filters others than A. Additionally, A and B may have different resources.
2. When receiving a packet P, B doesn't know if P has been already filtered by A. Packets with spoofed addresses may reach B, be considered as already treated without actually going through any access control check.
3. If a corresponding state cannot be found in A when receiving a packet, it is impossible for A to know if the state is missing because no packet has been previously received for this connection or because the state is located on B.

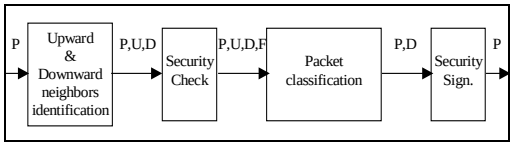

Fig. 2. Overall filtering process

We address these problems in the three following sections. The overall filtering process is presented in Figure 2. Given a packet P, we first identify the upward and downward neighbor filtering components U and D. We then check that the packet is not fraudulent (F) using the upward filtering component identity. Given this information, we execute the distributed packet classification algorithm. Finally, we sign the packet using the downward filter identity and send the packet to its destination.

3.2 Neighboring Filters Identification

The goal of stateful filtering being to match packets coming back and forth, symmetric routing is mandatory for the operation of our architecture. As a result we expect that firewalls [1] or at least the firewall management architecture [2] will be able, for each couple (source, destination) to define which filtering components are on the path. As routing tables may hold a limited view of the network topology (e.g. devices using a default route) this information may have to be provided to filtering components so that upward (U) and downward (D) components can be identified.

Classical routing structures such as a Patricia trie can be used. However in order to store filtering components information, we extend each routing entry with the downstream filtering component to the destination of the packet. The lookup process includes two addresses structure lookups providing the addresses of the upward and downward components if any.

3.3 Buckets state estimation

Bucket State Transmission

As mentioned earlier we expect symmetric routing to be used among our distributed filtering components. Our approach to estimate the state of a bucket is to transmit this information between filtering components. In order to do so we define an experimental DSCP value using the first three bits (a,b,c) of the TOS field. This new field indicates the percentage of state space in use in 1/8th as indicated in Figure 3. Bit f indicates if the packet has already been filtered by an upstream component. Finally, the next two bits indicate an experimental DSCP as specified in [5].

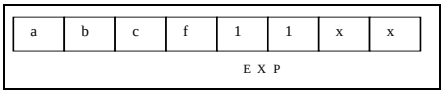

Fig. 3. TOS Field

Filtering components maintain a state table indicating the number of states used for each bucket in each adjacent filtering component. We say that two filtering components A and B are adjacent for a communication (Co) and a routing topology when there is no other filtering component C that Co would cross after going through A and before going through B and reciprocally. Let's consider the set made of the n filtering components adjacent to A, $S = \{B_1..B_n\}$. A maintains a state table $TB_i[1..SB_i]$ where $TB_i[j]$ indicates the proportion of states used in bucket j for the state table in B_i and SB_i represents the number of buckets used in B_i.

When a packet P is received, the DSCP value is retrieved after the packet is validated and used to update the adjacent component B_j, TB_j. This DSCP value is set to the value in TA for packet P before the packet gets forwarded.

Packet treatment

When receiving a packet P, A extracts the set of invariant fields used to build an invariant descriptor P'. Note that the way P' is built depends on the direction of the packet. Depending on P' three cases may happen:

1. P belongs to a flow that is handled locally in which case a bucket in TA should include the corresponding state.
2. P belongs to a flow that is handled remotely. This flow can be either treated by an upstream component (2.1) or by a downstream component (2.2).
3. P belongs to a new flow that is not yet treated.

Using the local and adjacent filtering component identifiers A and B_i, A generates hash values H=H(P', A) and H'=H(P', B_i). In order to test case (1) we lookup the state in table TA[1..SA] by going through the states located in bucket TA[H%SA].

Using TB_i is unfortunately not sufficient to differentiate cases (2) and (3). We therefore need an additional table to distinguish flows that are handled remotely from unknown flows. This table has to allow us to clean connections that have timed out and therefore has to allow us to keep a timer for each flow. In order to do so we build a "temporal" bloom filter BF[1..SF] in which each component of the couple (BF[H%SF], BF[H'%SF]) can take two types of value.

– 00 indicates that the corresponding flow is not present in the filter.
– (01, 10, 11) indicates the value of the timer the last time a packet belonging to the flow was received.

Table 1. Meaning of timer values

(xx,yy) value	**Meaning**
(00,00)	No flow.
(00,xx), (xx,00)	Partial collision.
(xx,xx)	No collision, Total collision with same timer values.
(xx,yy), (yy,xx)	Partial collision, Total collision with different timer values.

The couple (BF[H%SF], BF[H'%SF]) can therefore indicate four types of events as indicated in Table 1. Note that partial collisions are not problem here since half timing information is still accurate. Total collisions are more problematic since they can transform a non existing or timed out flow into a valid one. We later show how the probability for these collisions can be controlled by rightly designing the bloom filter.

For a timeout value T we maintain a timer CT that is increased every T/3 and can take values (01,10,11). Before increasing CT we set to "00" entries that hold a value equal to the upcoming CT. Note that timers are not kept here for security purposes but only to limit the number of "foreign" states that have to be stored locally.

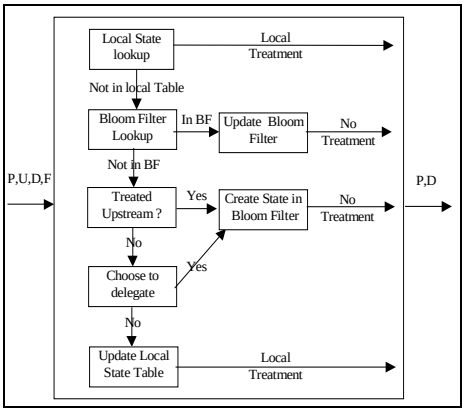

Fig. 4. Detailed packet treatment

If (3) applies to P (i.e. BF[H]=00 or BF[H']=00), we follow the multiple hash function algorithm [4] to decide where the state should be stored. To do so we compare TA[H%SA] and TB_i[H'%SB_i]

– If TA[H%SA] > TBi[H'%SBi] we store the packet value in the bloom filter (i.e. we set BF[H%SF] and BF[H'%SF] to CT).
– If TA[H%SA] <= TBi[H'%SBi] we perform classical packet classification operations and store the corresponding state in TA. We also mark the TOS f bit.

If (2) applies to P the action depends on the value of BF[H] and BF[H']. If both carry the same value, we update the bloom filter with the current timer. In the other

case, we update the oldest identifiers. This is aimed to make sure that only one connection gets updated in the case of a partial collision.

Figure 4 provides the whole process followed to distinguish case 1, 2.1, 2.2 and 3.

3.4 Packet checking

As mentioned earlier, our scheme does not provide a way for B, when receiving a packet, to know if it has been previously filtered and accepted by A or if it reaches B because of the state distribution technique. In order to solve this problem we suggest a weak authentication by overloading the ID field in the IP packet header. The usage of this field for operations unrelated to packet fragmentation as been suggested for packet tracking operations one of the reason being the small proportion (i.e. less than 0.5% on average) of fragmented traffic on the Internet [7]. Our scheme is also based on the ability for adjacent filtering components to share a common secret key K.

When a packet P reaches filtering component U, we use the downward filtering component identity D to retrieve the corresponding secret key K_{DU}. We then use the invariant part of the packet P' and K_{DU} to compute a HMAC value H'' and store H'' in the ID field of packet P before sending it toward its destination. When receiving packet P, a filtering component first identifies the neighboring filter U previously crossed by P. As routing is expected to be symmetric, this can be performed with an extension to the neighbor identification. It then retrieves the corresponding key K_{DU}, the invariant part P' of P, computes H''' using P' and K_{DU} and checks if H''' matches the content of the ID field. If the two values match, we perform previously specified operations. In the other case, we mark the F selector so that P is treated locally.

As the size of the authentication field makes it vulnerable to brute force attack we need a mechanism to discourage such attempts. The mechanism we present here does not prevent a determined attacker to find the key but instead gives the security officer enough time to identify such attempts. To do so, we maintain a counter C_{DU} for each key K_{DU} indicating the number of failed authentication operations since the last key change. When the number of tests performed by the attacker passes over 0.1% of the possible key space (i.e. roughly 65 packets), we reconfigure the filtering process so that every packet supposedly going through U is treated locally. In order to stop this local processing, we maintain a counter P_{DU} indicating the number of packets supposedly coming from U. When P_{DU} passes over 30.R where R is the maximum number of packets that D can handle per second, we perform a new key exchange and reset counters. As there is no difference in the filtering process once all packets are filtered, an attacker cannot learn anything from the system after the first 65 packets. He then has to wait for at least 30 seconds before being able to make another attempt. The probability for him to succeed to find the key evolves slowly from 10% after 2 hours, 50% after 5 hours to 90% after 20 hours.

Finally, we mention that this weak authentication process can be avoided in some cases when all communications received by a filtering component go through another filter first. This is in particular the case for filtering components that are collocated within a single device.

4 Dimensioning issues and Evaluation

Bucket state/DSCP Code Relationship

As mentioned earlier two filtering components can use hash tables with different sizes, as a result the DSCP value is not absolute. We therefore need to define a relation between the number of states kept in a specific bucket, the number of states kept by the filtering component and the DSCP value. Given a worst case state lookup speed, and the fluid limit simulation results provided in [4], we can bound the maximal number of states expected for a bucket in the state table (Smax). Using this limit and the number of states S located in bucket TA[H%SA], we define the (a,b,c) DSCP bits as: DSCP = TA[H%SA] = $|(S/\ Smax)\ .\ 2^3|$ where |x| represents the integer value for x.

Bloom Filter Dimensions

In order to dimension the bloom filter, we first select the bandwidth for which we want to dimension the filter (i.e. 10Gbps in our case) and consider that the bloom filter will store most of the traffic. We then compute the maximal and average number of new flows treated during a few selected lengths of time. To evaluate the likelihood of each flow duration we use an approximation of the flow duration probability distribution provided in [6]. This approximation is provided in Table 2.

Table 2. Distribution of flows

Flow dur. (s)	**% flows in class**	**Flows/s (worst/avg)**
[c0=0;c1=1]	p0=90	[28125.10^3; 351562]
[c1=1;c2=10]	p1=9	[28125.10^2; 35156]
[c2=10;c3=100]	p2=0.9	[281250; 3515]
[c3=100;c4=1000]	p3=0.09	[28125; 352]
[c4=1000;c5=10000]	p4=0.01	[3125; 39]

For the worst case scenario we consider single packet flows where the size of each packet is minimal (i.e. 40 bytes). For the average case, we use the internet average packet size (i.e. 320 bytes) and an average of 10 packets per flow. Although we expect our architecture to be subjected to the worst case scenario during short period of time (i.e. DDOS attacks), we do not expect these periods to exceed a few seconds as external mechanisms should be put in place to avoid such traffic conditions.

A point however not considered in Table 2 is the timeout value for the bloom filter. This value must be sufficiently low to avoid high collision rates and sufficiently high to prevent active connections to get cleared. Consequently we compute maximal collision probabilities for various bloom filter sizes and timeout values. To do so we first evaluate the number of flows in the filter at time t, Mf(t).

$$Mf\ (t) = (N \cdot To) + Nr\ (t)) \quad \textbf{(1)}$$

Where N represents the number of new flows received each second, To represents the clearing timeout value and Nr(t) represents the number of flows from previous timeout periods that are still active at time t. Nr(t) can be computed as follows:

$$Nr(t) = N \cdot \sum_{i=1}^{i=t} (1 - (p_0 + \sum_{j=1}^{j=\min(4,\lfloor \log_{10}(i) \rfloor + 1)} \frac{(\min(i, c_{j+1}) - c_j)}{(c_{j+1} - c_j)} \cdot p_j)) \quad (2)$$

Mf(t) is maximized when t > 10000 so that:

$$Mf_{\max} \approx N \cdot (To + 2) \quad (3)$$

As we expect most of the states to be in the current CT time period, we compute a collision rate as if we were using a classical bloom filter. We give here the maximal collision probability (Cmax) in a bloom filter of size 2^m after storing Mfmax flows.

$$C_{\max} = (1 - (1 - 1/m)^{2 \cdot Mf_{\max}})^2 \quad (4)$$

Figure 5 provides the maximal false positive probability for several bloom filters sizes (16Mb, 64Mb), flow rates (average and worst case scenarios) and timeout values. A 64Mbytes bloom filter with a 30 seconds timeout seems appropriate.

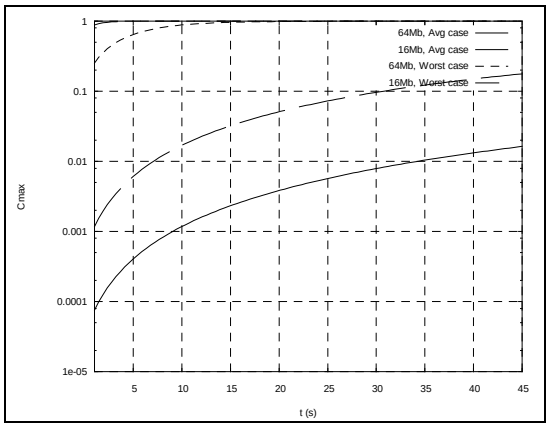

Fig. 5. False positive probability for various bloom filter sizes, flow rates and timeout values

Independently from the timeout value, flows identifiers can be overwritten by newer flows. In our case, flows under 100 seconds are unlikely to get overwritten (around 6%). On the other hand flows over 400 seconds are very likely to be overwritten in the filter (more than 50%). By being overwritten, a terminated flow will appear as still active for the duration of the overwriting flow. However, as shown in Table 2, the likelihood of the overwriting flows to have a long duration is very small. As a result overwriting operations will have a limited impact (around 4 seconds on average) on the evaluation of long flows.

Another point is that packets generating a collision do not bypass the access control architecture. As each packet goes through several filtering components the overall

probability to reach the last filtering component and being misclassified in each one decreases exponentially with the number of filtering component crossed.

Finally if some packets reach the last filter without being treated we expect the identification process to set mandatory filtering (F) for remaining packets.

Neighbor States tables storage

In order to evaluate the size for these tables we need to set a limit to the number of concurrent states that can be stored in a bucket. In order to do so we could have used formula (1). We instead choose to use indicators computed from real-life packet traces which allows us to evaluate state numbers and assess the validity of Mf(t) at the same time. We therefore use indicators of average link utilization (U_{avg}), average number of active flow entries (E_{avg}) and maximum number of active flow entries (E_{max}) computed over 51 long packet traces totaling to more than 900 hours of network monitoring. These indicators were computed by the Sprint IPMON project [9] on traces captured in 2002 and 2003 at several locations within Sprint operational network. We use these indicators to define ratios $R_{avg} = E_{avg}/U_{avg}$ (average number of states per bps) and $R_{max} = E_{max}/U_{avg}$ (maximum number of states per bps).

Table 3. Number of buckets and memory requirements for various bucket depths

Scenario	Num. of buckets	Max. Num. of states	Memory required
Worst Case	10.10^6	9	5Mbytes
	40.10^6	6	15Mbytes
Average Case	$2.75.\ 10^6$	9	1.3Mbytes
	11.10^6	6	4.1Mbytes

In our case R_{avg} values remain within $[4.10^{-4};\ 40.10^{-4}]$ with an average value $R_{avg}=11.10^{-4}$ (Note that our previous model provides a similar value) while R_{max} values remain within $[7.10^{-4};\ 80.10^{-4}]$. In order to compute the number of states kept we use two indicators:

- The maximal R_{max} value (0.0080) models the worst case situation.
- The average R_{avg} value (0.0011) models an average scenario.

Table 4. Memory requirements and gains per filtering component depending on the average number of filtering components between a source and a destination

Num. filt. comp	Max memory req.	Avg memory req.	Gain per filter
Original scheme	12 GB	1.6 GB	1.0/1.0
2	6.2 GB	905 MB	1.9/1.8
5	2.6 GB	425 MB	4.6/3.8
10	1.4 GB	265 MB	8.6/6.1

Using these two indicators, we can compute the average and maximal number of flows. Combined with fluid limit simulation results provided in [4], we estimate the storage requirement for a component depending on several bucket depths (Table 3).

Assuming an average number of 10 neighbors we estimate the total storage to lie between 13 and 150 Mbytes depending on the maximum number of states per bucket.

As a consequence the total storage required to implement our scheme stands between 77 and 214Mbytes. Table 4 provides memory requirements, overhead per filter and gain per filter depending on the number of filters crossed by a communication.

Processing requirements

In order to evaluate our scheme, we compare the processing requirements to the original classification scheme [4]. Table 5 provides the number of CPU cycles used by both approaches in each filter where C represents the cost for "rectangle" classification, H the cost for a hash over 40 bytes, M the cost for an HMAC over 40 bytes, L the cost for an address lookup and N the number of packets per flow.

Table 5. Operations performed by Classical and Distributed architectures

Filter #	Classical	Distributed
1	C+2HN	C+N(2H+2M+2L)
2,3,...	C+2HN	N(2H+2M+2L)

Figure 6 shows the difference of time complexity in processor cycles between both schemes using C=10000 (linear search through 200 rules using 5 fields [3]), H=196, M=272 (UMAC performance tests [8]), L=250 and several N values. The distributed scheme is clearly more interesting for short flows (i.e. less than 6 packets per flow) while the regular scheme is more interesting for long flows (i.e over 15 packets per flow). It should however be noted that a faster packet classification scheme would render our scheme less attractive from a processing requirement point of view.

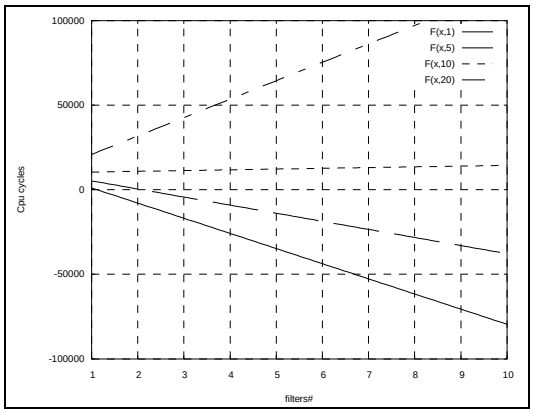

Fig. 6. Difference in term of processing requirements between distributed and regular approaches depending on the number of packet filters

Implementation and tests

This architecture has been implemented under the *ns* simulator. The goals for the implementation were twofold, we wanted to check the correct behavior of the architecture and test its behavior under real life traffic. The neighbor identification module as well as the classification algorithm were implemented as an extension to the current address classifier class. As *ns* does not support real IP addresses in packets, we also

extended IP packets allowing us to transport the packet header information required by our scheme as well as protocol specific information. A few modifications to other parts of the simulator were also performed in order to facilitate our tests. At the time of this writing only the protocol behavior was tested using generated traffic.

5 Conclusion

We present in this paper an architecture for connection states maintenance that was designed to reduce memory requirements as well as limit processing requirements in some specific cases. Our main contribution is to show how an existing state maintenance scheme can be distributed and that such a distribution can bring large improvements. Although this paper only focuses on a single state maintenance technique, we believe that such distribution could apply to other techniques.

On the other hand, our approach makes the implementation of functions like flow monitoring more difficult and renders functions like diffserv or fragmentation impossible. Additionally our scheme may generate flow interruptions in the case of route changes when the filter holding the state of the rerouted flows is longer part of the path between the source and the destination. However, this problem would also happen with a regular access control architecture.

This work is currently continued in two directions. We currently work on a real life implementation that would allow us to perform tests with real life traffic. This would allows us to compare results with traffic models used in this paper. Another direction is to explore improved bloom filter structures in order to limit collisions by performing a separation of short and long flows.

References

1. Joshua D. Guttman. Filtering Postures: Local Enforcement for Global Policies. IEEE Symposium on Security and Privacy. Oakland. May 1997.
2. Yair Bartal, Alain Mayer, Kobbi Nissim, Avishai Wool. Firmato, A Novell Firewall Management Toolkit. IEEE Symposium on Security and Privacy. Oakland. May 1999.
3. Daniel Hartmeier. Design and Performance of the OpenBSD Stateful Packet Filter (pf). Usenix Annual Technical Conference. June 2002.
4. Andrei Broder, Michael Mitzenmacher, Using Multiple Hash Functions to Improve IP Lookups, In proceedings of IEEE Infocom 2001. Anchorage, Alaska, April 2001.
5. K. Nichols and al. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers (RFC 2474). December 1998.
6. Yin Zhang and al. On the Characteristics and Origins of Internet Flow Rates. ACM SIGCOMM 2002. Pittsburgh, USA. August 2002.
7. Colleen Shannon and al.. Beyond Folklore: Observations on Fragmented Traffic". IEEE/ACM Transactions on Networking. December 2002.
8. John Black et al. UMAC: Fast and Secure Message Authentication. Advances in Cryptology - CRYPTO '99. Lecture Notes in Computer Science, vol. 1666, Springer-Verlag, 1999.
9. Sprint Labs. IP Monitoring Project. Available at http://ipmon.sprint.com/ipmon.php.