Complexity Attack Resistant Flow Lookup Schemes for IPv6: A Measurement Based Comparison

David Malone and R. Joshua Tobin

Abstract

In this paper we look at the problem of choosing a good flow state lookup scheme for IPv6 firewalls. We want to choose a scheme which is fast when dealing with typical traffic, but whose performance will not degrade unnecessarily when subject to a complexity attack. We demonstrate the existing problem and, using captured traffic, assess a number of replacement schemes that are hash and tree based. Our aim is to improve FreeBSD's ipfw firewall, and so finally we implement the most promising replacement schemes. We show that even though they are more costly computationally, they do not noticeably degrade IPv6 forwarding performance.

1 Introduction

In [6] the danger of using algorithms that are open to complexity attacks was highlighted. In a complexity attack, an attacker causes a system to perform poorly by choosing particular inputs that cause the algorithms used by the system to exhibit worst case rather than typical complexity. The canonical example is a hash table: if the attacker controls some of the data used as a key to the hash function, then they may choose the inputs so that there are many hash collisions and lookup performance is poor. In particular cases, this may result in denial of service or incorrect operation of the system. In the case of a firewall, if per-packet processing becomes too expensive then maximum throughput will be reduced and valid packets may be dropped.

In this paper, we are considering flow lookup for a firewall supporting IPv6. A flow key will typically consist of two IP addresses, two port numbers and a protocol (TCP/UDP/ICMP). An attacker will typically control much of the source address, some of the destination address and some of the port numbers. In IPv4 this amounts to a modest number of bits, however in IPv6 this may amount to hundreds of bits, giving an attacker wide scope for generating hash collisions.

The aim of this work is to assess several different options for IPv6 flow lookup that are resistant to complexity attacks. This is with a view to replacing the lookup scheme used in FreeBSD's ipfw firewall, which currently uses an xor based hash for flow lookup. In particular, we want to choose a scheme that has good performance characteristics under typical circumstances while also offering resistance to complexity attacks. To understand typical performance we have collected a trace of IPv6 traffic so we can assess the performance of different schemes on it. Importantly, we want to choose a scheme that has good performance on both smaller CPUs (as might be found in small home routers) and more powerful CPUs (that might be found in servers or high end firewalls).

There are a number of choices for replacing the lookup scheme. One is to replace the hash function with one that is resistant to collisions. This is commonly achieved either by using a cryptographic hash function or by choosing a hash function randomly at system startup. The other option is to use algorithms that have bounds on their performance that are independent of the inputs. In the first class we will look at some hashes based on universal hashing [5] and Pearson's hash [10], which are essentially a collection of parameterised or keyed hashes. In the second class we will look at some tree based algorithms, such as treaps [2], splay trees [13] and red-black trees[3]. We will compare these to baseline results for an xor based hash and a binary tree with no explicit balancing.

We note that we consider the problem of looking up state for a specific flow, rather than the problem of finding a best-matching firewall rule (e.g. [7, 11]) or doing a longest-prefix route lookup (see the review in [1], but the literature continues to grow). Our problem is somewhat easier: as it does not involve matching a range of flows, and so should be amenable to simpler algorithms.

The remainder of the paper is as follows. In Section 2 we demonstrate the problem of hash collisions for a number of commonly used hash functions using a number of simple attacks. In Section 3 we will describe our method for assessing the different lookup schemes. In Section 4 we will describe our results. We summarise our finding and draw conclusions in Section 5.

978-0-7695-3479-4/08 $25.00 © 2008 IEEE
DOI 10.1109/EC2ND.2008.9

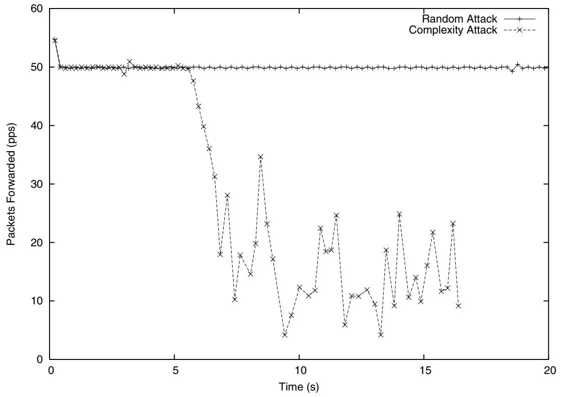

Figure 1. Forwarding rate for a 50 pps stream for a Soekris Net 4501 when subject to a attack streak of 3000 pps. In one case the attacking packets are randomly addressed, in the other they are chosen to all hash to the same value.

2 Simple Attacks on Hashes

In order to demonstrate the vulnerability of ipfw's current xor hash function we ran a basic proof of concept attack. We set up a Soekris Net 4501 board as a router between two servers. We sent packets from one to the other, and counted how many were received at the destination server. After about 3s we began to send packets from a third machine to the router: in the first case randomly addressed packets, and then packets maliciously designed to collide under the xor-based hash. In both tests the packets being forwarded from the first server to the second were being sent at a rate of 50 packets per second, and the packets from the third machine were sent at the same rate of 3000 packets per second.

Figure 1 clearly shows that the router could handle this rate of packets under normal circumstances. However, if they are designed to collide then the forwarding capacity is quickly diminished. As a result, packets are dropped and throughput is limited.

Though we have demonstrated this attack using xor, many hashes are likely to be subject to quite simple attacks unless they have been designed with resistance to collision based attacks in mind. To demonstrate this, consider a hash table with 256 different chains. We will hash 36 byte strings (enough space for two IPv6 addresses and two port numbers) and insert them into this table with different hash functions. The different hash functions used are shown in Table 1. The low eight bits returned from the hash function are used as the index into the hash table.

Name	Description

Universal	$h \leftarrow 0$
	foreach ($byte[i]$) $h \leftarrow h + K[i] * byte[i]$
	return $h \mod 65537$

Xor	$h \leftarrow 0$
	foreach ($byte[i]$) $h \leftarrow h \oplus byte[i]$
	return h

XorSum	$h \leftarrow 0$
	foreach ($byte[i]$) $h \leftarrow h + (byte[i] \oplus K[i])$
	return h

SumXor	$h \leftarrow 0$
	foreach ($byte[i]$) $h \leftarrow h \oplus (byte[i] + K[i])$
	return h;

DJB2	$h \leftarrow 5381$
	foreach ($byte[i]$) $h \leftarrow 33 * h + byte[i]$
	return h

Pearson	$h_1 \leftarrow h_2 \leftarrow 0$
	foreach ($byte[i]$) $h_1 \leftarrow T_1[byte[i] \oplus h_1]$
	$h_2 \leftarrow T_2[byte[i] \oplus h_2]$
	return $h_1 + h_2 * 256$

MD5	return two bytes of MD5(bytes)
SHA	return two bytes of SHA(bytes)

Table 1. Table of Hash Algorithms. Multiplication is denoted by $*$, addition by $+$ and xor by $\oplus$. $K[.]$ is an array of randomly selected bytes. $T_i[.]$ is a randomly selected permutation of a byte. The constant 65537 in the Universal hash is selected as a suitable sized prime.

Sequence	Description of i^{th} string
Sequential	The usual binary representation of i.
Hamming	First all zeros, then each string with one bit set, then each string with two bits set, ...
xor zero	The binary representation of i with each byte repeated twice.
xor ones	As xor zero, but with ones compliment of every second byte.
sum zero	As xor zero, but with twos compliment of every second byte.
random	Filled randomly using the arc4 PRNG.

Table 2. Sequences of strings used as input to hashes.

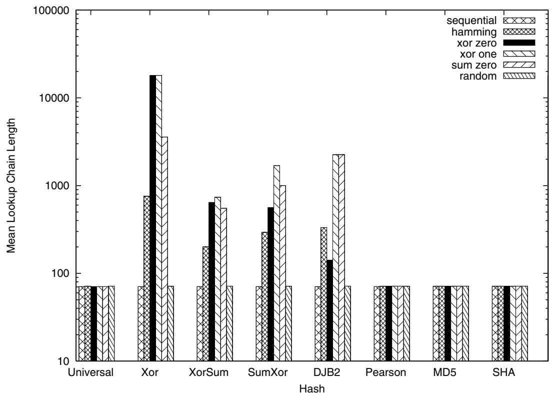

Figure 2. The mean lookup chain length different combinations of hashes and inputs. Note log scale.

This table shows some unkeyed algorithms (Xor and DJB2), some keyed algorithms using simple operations (XorSum and SumXor), some keyed algorithms that use more complicated operations to mix in their key state (Universal and Pearson) and some algorithms designed with cryptographic properties in mind (MD5 and SHA). More details of these hashes can be found in Appendix A.

We use these algorithms to hash several different sequences of strings. The sequences are summarised in Table 2. We insert a sequence of 36,000 of these strings into the hash table. We then calculate the average number elements examined to do a lookup in the table. If the strings are well distributed throughout the table, the lookup times will be small. However, if the strings are unevenly distributed, then the mean lookup time may be large.

Figure 2 shows the results of this experiment. We see that the more complex hashes perform equally well on all input sequences, achieving close to the optimal number of approximately 70. Also, all the hashes perform close to optimally on random and sequential inputs.

However, all of the more simplistic hashes exhibit substantially higher lookup times for the other sequences. As intended, the xor zero and xor one inputs actually perform in the worst possible way for the simple xor hash, by putting all entries in a single hash chain.

Even though these sequences are not specifically targeted at XorSum, SumXor or DJB2, they clearly cause performance problems for them. This is not any kind of indictment of the DJB2 hash: it is targeted at typical ASCII strings and was not designed to be resistant to collision attacks. However, XorSum and SumXor demonstrate that keyed hashes need to be designed carefully if they are to be collision resistant. We also see that we do not need to cause all the strings to hash to the same value to get bad performance, it is sufficient to get the strings into a small number of values with moderate probability.

While keeping the hash chains well-balanced is important to hash table performance, the real CPU cost to the system also includes the cost of calculating the hash itself. While the exact cost depends on many factors, the cost of calculating SHA could easily be more than an order of magnitude greater than just xoring bytes together. When using a hash table, we must always calculate a hash value, regardless of if the hash is under attack. Thus we would prefer to choose a hash that is resistant to attack and not too costly.

3 Method for Assessment of Lookup Schemes

As described in Section 1, we aim to choose a lookup scheme which is resistant to complexity attacks, but which offers reasonable performance in usual circumstances. Of course, usual performance is determined by inputs used, in our case the IP addresses and port numbers from packets. For this reason, we have based our assessment on a number of collected packet traces from links with live IPv6 traffic.

We collected a trace from an IPv6 only link with a total of 4664565 packets. Not all of these were IPv6 packets and we processed 4588721 packets corresponding to 30566 flows. The number of packets in the n^{th} largest flow is plotted against n in Figure 3. We can see typical sorts of features that we expect for Internet traffic: small numbers of large flows and a large number small flows. For some of our tests we used only subset of this data. This small trace contains 386290 packets and 12229 flows.

In order to assess different lookup mechanisms we built a simple framework that reads each packet in the trace, parses the headers, looks up the corresponding flow and adds a flow entry if the flow was not present. This provided a convenient environment for developing and assessing different schemes, without the complexity of a testbed setup or kernel development. Each scheme was assessed in terms of CPU

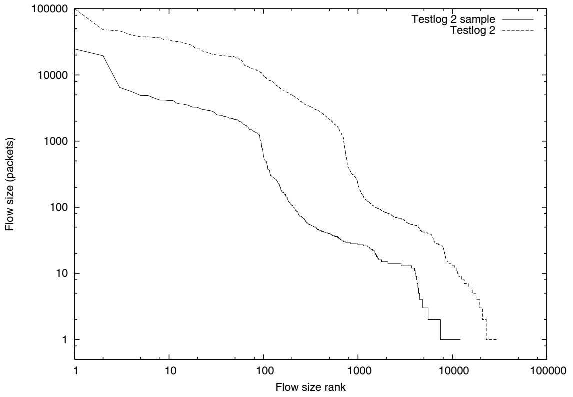

Figure 3. Size of the n^{th} largest flow against n. Note log-log scale.

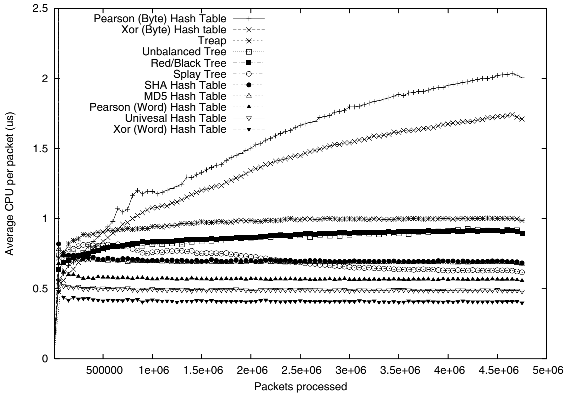

Figure 4. Mean cost of flow lookup over time for high-end processor.

used per packet processed.

Note that in this framework we never remove flows, we only lookup or add them. This is a deliberate choice as we want to see how the lookup schemes behave as the number of flows we are tracking state for increases. Unfortunately, this will not exactly replicate the pattern of lookups for links with different numbers of active flows, but it should provide some indication of the likely performance.

The schemes that we chose to assess were: 65536 entry hash tables using xor, Universal, Pearson, MD5 and SHA; 256 entry hash tables using xor and Pearson; splay tree, unbalanced binary tree, red-black tree and treap. The xor based hash tables and unbalanced binary tree are subject to complexity attacks, but serve as a baseline to compare the other schemes to. Again, some more details of these schemes can be found in Appendix A.

Naturally, some aspects of this test framework are not that realistic: reading pcap data from a file in a C environment is not that similar to receiving packets from an Ethernet device in a kernel. Once we had performed an assessment of the schemes, we implemented the more promising ones in the ipfw kernel module. We then checked the impact of these schemes on peak packet forwarding speed.

4 Results

Figure 4 shows the flow-lookup time per packet for a modern processor (Pentium Core 2 Duo 2 GHz) for the full trace. The value plotted is the cumulative CPU time divided by the total time. The points shown are also averaged over at least six runs. We can see how quickly startup costs (such as the initialisation of keys for the keyed hashes) are amortised over time. While startup costs are unlikely to be an issue for a firewall initialised at boot time, this might be an issue for other applications that create tables of IPv6 flows on the fly.

The total number of flows is large relative to the 256 entry hash tables, so we can see the performance of the 256 entry tables degrade over time. The fastest lookup method is always the 65536 entry hash table using just plain xor. However, the Universal hash is not far behind and the Pearson-based 65536 entry hash table is not far behind that. The MD5 and SHA based hashes are comparable, but slower again.

Interestingly, the tree based methods are not that competitive. The treap is most expensive, followed by unbalanced trees and red-black trees, followed by splay trees. In fact splay tree's performance appears to be gradually increasing, and is almost competitive with the hash based schemes for large numbers of packets.

The fact that the unbalanced trees perform relatively well indicates that the balancing is not often actually required in practice. Treaps are quite likely to perform a lot of balancing steps, and this is proving to be wasted effort. Red-black trees have some measure of how balanced a tree is, and so probably perform less unnecessary balancing.

Splay trees do not directly balance the trees, but instead move frequently accessed nodes towards the top of the tree. For this sort of application, where some flows have many packets and packets are likely to occur in bursts, we expect quick access to these flows to be helpful. Indeed, we find that towards the end of our trace, the number of packets attributable to the top 10% of flows is increasing, which may explain the improved performance of splay trees towards the end of the test.

In summary, for a high-end CPU, it looks like using the Universal hash would be a reasonable choice for an attack resistant lookup method. However, we would like to check the cost on a lower-end CPU. Our lookup schemes exhibit

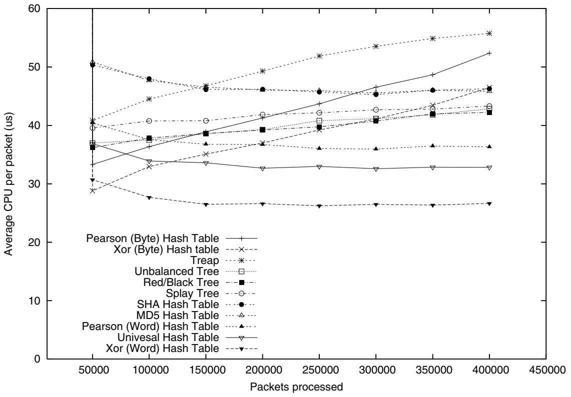

Figure 5. Mean cost of flow lookup over time for soekris.

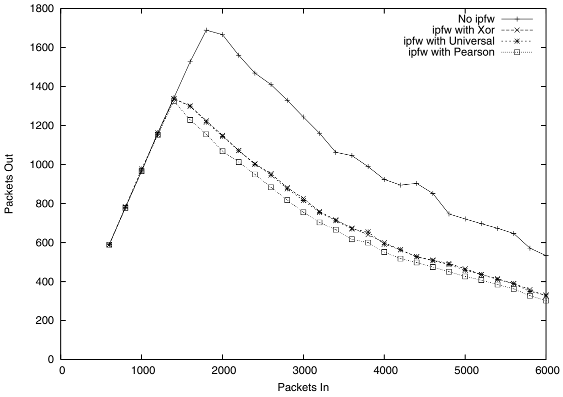

Figure 6. Packets in versus packets out for a single IPv6 flow on a Soekris Net 4501 using different hash functions.

different memory access patterns (for example, Pearson will access T_i randomly while Universal will access K sequentially) and instruction mixes. As the relative cost of memory accesses, cache lookups and more complex operations, such as multiplication, can vary greatly from CPU to CPU, the relative performance of the lookup schemes may vary.

Figure 5 shows the results of the same tests for a Soekris Net 4501 board on the smaller trace. The Net4501 is a device that has been popular choice for building embedded devices using FreeBSD. This board has an AMD Elan SC520 microcontroler as a CPU. This is a much more modest CPU with no L2 cache and with low power consumption, which can be found in small router products. While the absolute cost per-packet is higher, we see basically the same results as for the fast CPU. Note that since we have only used the smaller trace, we can not see the full large-dataset behaviour, but we expect the trends to be the same.

We conclude that Universal hashing is likely to be our best choice for a collision resistant hash, with Pearson's hash as a backup choice. We now aim to check the performance of these in comparison to the present xor hash in a more realistic setting.

We implemented variants of the ipfw module using each of these hashes and tested them on the Soekris Net 4501 board running FreeBSD 7.0. We configured the Net 4501 as a router between two high-end servers, with ipfw configured to keep and check flow state. When forwarding small packets, the Net 4501 is limited by its CPU. To test the impact of the different hashes, we sent 67-byte UDP packets from one server to the other at different rates and observed the behaviour of the Net 4501. To bring out the difference in cost between calculating the hashes, we use a single flow for this test.

Figure 6 shows the results of this experiment. We can see that when the firewall is not active, the Net 4501 can forward about 1700 packets per second. When the firewall is active it can forward just less than 1400 packets per second using either xor or the Universal hash. Pearson slightly trails the performance of Universal and xor, reflecting our earlier results. The larger gap for Pearson's hash can possibly be attributed to the larger key size which is not accessed sequentially.

If we offer more packets than the peak forwarding, the performance of all schemes degrades. This is due to receive livelock [12] issues and can be mitigated using known techniques, such as enabling FreeBSD's interface polling, implemented by Luigi Rizzo.

We note that this highlights that a device may be overwhelmed by an attacker with sufficient resources, even when using algorithms that are resistant to complexity attacks. However, using attack resistant algorithms increases the resources required by an attacker.

5 Conclusion

In this paper we considered the problem of IPv6 flow lookup that is efficient in typical situations, but also resistant to complexity attacks. Our aim was to replace the xor-based hash lookup in FreeBSD's ipfw firewall. We found that despite the more complex operations involved in our universal hashing, it seems to offer good performance on both the hardware platforms considered.

There are some closely related questions still to be answered. We have begun looking at some other traffic traces and see similar results. Looking at performance on traces containing a mix of IPv4 and IPv6 traffic would be useful, as in practice a firewall will often be tracking state for both.

It would also be interesting to look at the BSD C macro based implementations of splay trees and red-black trees to see if they offer different performance characteristics: our initial tests indicate that these macros are more efficient than our implementations, but do not change our results. Finally, it may be interesting to consider lookup schemes based on Cuckoo hashing [9] and related innovations.

Finally, it would be useful to have a way to assess the suitability of a hash function for situations where complexity attacks might occur. For example, after a discussion with Pearson, we believe that the 1-byte Pearson hash has structure that may leave it open to attack. We would like to study this further to understand which hashes may be open to similar attacks.

References

[1] M. A. R.-S. adn E. W. Biersack and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network*, 15(2):8–23, 2001.

[2] C. R. Aragorn and R. Seidel. Randomized search trees. In *Annual Symposium on Foundations of Computer Science*, pages 540–545, 1989.

[3] R. Bayer. Symmetric binary B-trees: Data structures and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] D. Bernstein. Re: Hash??? what are the latest hot-shot methods? `http://groups.google.com/group/comp.lang.c/msg/6b82e964887d73d9`, December 1990.

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[6] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[7] P. Gupta and N. Mckeown. Dynamic algorithms with worst-case performance for packet classification. In *IFIP Networking*, pages 528–539, 2000.

[8] D. Hartmeier. Design and performance of the openbsd stateful packet filter (pf). In *Proceedings of the USENIX 2002 Annual Technical Conference, Freenix Track*, pages 171–180, 2002.

[9] R. Pagh and F. Rodler. Cuckoo hashing. Technical Report RS-01-32, BRICS Report Series, Dept. of Computer Science, University of Aarhus, 2001.

[10] P. K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, 1990.

[11] L. Qiu, G. Varghese, and S. Suri. Fast firewall implementations for software and hardware-based routers. In *Proceedings of ACM SIGMETRICS*, 2001.

[12] K. K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *IEEE Global Telecommunications Conference*, volume 1, pages 622–626, 1992.

[13] D. D. Selator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686.

A Algorithm Details

Let us briefly comment on the background of hashes used in Section 2. The Xor hash is often used as a cheap hash function. The DJB2 hash is attributed to Dan Bernstein and was designed as a quick-to-compute hash function of C strings [4]. XorSum and SumXor are not well designed hash functions, but are intended to be keyed hashes built from cheap operations. The Universal hash implementation is based on the CW hashes described in [6] and is in the class of hashes described in [5]. Our MD5 and SHA implementations are from the OpenSSL library. They are unkeyed, but could easily be keyed by, say, xoring input bytes with the key.

Pearson's hash uses a randomly generated permutation table of the integers $\{0, 1, \ldots, 255\}$, denoted by T. We first initialise h to 0. Then we iterate over each byte in the data to be hashed, using the formula: $h = T[h \oplus x_n]$, where x_n is the nth byte of the data. The final value of h is the hash of the string. Clearly this hash function only has the range of a single byte, which makes it unsuitable for some purposes. In order to increase this range, we generate two permutation tables, T_1 and T_2, hash the data in both (giving h_1 and h_2) and then return $h_1 + h_2 * 256$. We refer to this as Pearson (Word) and the former method as Pearson (Byte).

The Universal implementation also makes use of a randomly generated table, which we refer to as K. In this case, the table is the same length as the data we want to hash, and each byte in the table is randomly generated. In this case, $h(x) = \sum_i K[i] * x_i \mod 65537$.

Treaps, Splay Trees and Red-Black Trees are well known, but for the sake of completeness we will briefly describe them. A Treap is a tree in which each node is also assigned a random priority. Each node's key then obeys the tree property (the parent's key is greater than the left child's key and less than the right child's key) and each node's priority obey the heap property (both children's priorities are less than the parent's priority). Tree rotations are used to enforce the heap property.

Splay trees invoke a *splay* operation, which brings the most recently accessed node to the top of the tree. This helps to balance the tree, and decreases lookup time for frequently accessed nodes.

Finally, Red-Black trees colour each node either red or black according to certain rules. This results in the deepest node being no more than twice as far from the root as the most shallow node, hence balancing the tree. AVL trees were originally used in OpenBSD's pf firewall [8], however these have been replaced by red-black trees. The pf firewall, is also available in FreeBSD.