Comparing Hash Function Algorithms for the IPv6 Flow Label

Lewis Anderson
land062@aucklanduni.ac.nz
Nevil Brownlee
n.brownlee@auckland.ac.nz
Brian E. Carpenter
brian@cs.auckland.ac.nz

Department of Computer Science
The University of Auckland

Computer Science Technical Report (2012-002)
March 2012

ISSN 1173-3500

Abstract

We compare several stateless hash algorithms for generating IPv6 flow labels, by testing them against trace files of IPv6 traffic measured at four different sites. The criteria for comparison were uniformity of the resulting distribution of hash values and computing time. Of the algorithms tested, we recommend FNV1a-32. We also note how the hash values may be made hard for a third party to predict.

Introduction

Every IPv6 packet header includes a 20-bit flow label in a fixed position [RFC2460] whose formal definition is given by [RFC6437]. The choices made in that definition are explained in [RFC6436]. The primary usage envisioned for the flow label is to act as a “handle” for some form of load distribution in the network, as described in [RFC6438] and [BALANCE]. The flow label is not proof against forgery, so it cannot be used for reliable end-to-end signalling; for this and other reasons, RFC 6437 recommends that it should be treated as a “best effort” field, and in particular that it should be set to the same value for all packets considered to be a single application flow, in the source computer or by a router as close to it as as possible. This value should be chosen from a statistically uniform distribution, both to assist in its use for fair load balancing and to make it hard for a malicious third party to predict its value in future flows.

RFC 6437 and RFC 6438 further recommend that the flow label value should be chosen by using a stateless hash function applied to header fields in the packet. Typically those fields would be the classical IP header 5-tuple:

- Source IP address
- Destination IP address
- Protocol number (in IPv6, the final Next Header value in the header chain)
- Source port number (for TCP, UDP, etc.)
- Destination port number

The advantage of a stateless function (one with no input other than the 5-tuple) is to avoid any need for per-flow storage or per-packet lookup in the node computing and inserting the flow label value. This is particularly advantageous if the flow label is inserted by a router, which runs at full line speed and cannot afford to store and look up per-flow state for every packet. Other techniques could be envisaged for use in the source computer (e.g. a counter and a cryptographic hash function as in [GONT], or an application layer mechanism as hinted in [BALANCE]). However, this report is focussed entirely on the choice of a stateless algorithm.

RFC 6437 avoids specifying an algorithm, but it does give one as an example. It is based on an algorithm by von Neumann [VONN] that is known to give a reasonably uniform distribution and is rapid to compute, since it uses only simple logical operations. The objective of the present report is to evaluate this and other algorithms, both in terms of the uniformity of the resulting distribution of flow label values, and of the computing time needed. The method is to test various algorithms against the packet headers in real IPv6 packet traces of actual traffic. The remainder of the report describes the algorithms tested, the trace files used as datasets, and the results obtained.

The Algorithms

The hash function algorithm given as an example in RFC 6437 is as follows:

```
“For each packet for which a flow label must be generated,
execute the following steps:

1.  Split the destination and source addresses into two 64-bit values
    each, thus transforming the 5-tuple into a 7-tuple.
2.  Add the following five components together using unsigned 64-bit
    arithmetic, discarding any carry bits: both parts of the source
    address, both parts of the destination address, and the protocol
    number.
3.  Apply the von Neumann algorithm to the resulting string of 64
    bits:
    1.  Starting at the least significant end, select two bits.
    2.  If the two bits are 00 or 11, discard them.
    3.  If the two bits are 01, output* a 0 bit.
    4.  If the two bits are 10, output* a 1 bit.
    5.  Repeat with the next two bits in the input 64-bit string.
    6.  Stop when 16 bits have been output (or when the 64-bit string
        is exhausted).
4.  Add the two port numbers to the resulting 16-bit number.
5.  Shift the resulting value 4 bits left, and mask with 0xfffff.
6.  In the highly unlikely event that the result is exactly zero, set
    the flow label arbitrarily to the value 1.”
```

* The first bit output becomes the MSB of the resulting hash value.

This algorithm was identified as ***Algorithm 2*** in our tests. Note that it does not include a step intended to reduce predictability, which the RFC recommends. This can easily be achieved by adding a host-specific nonce value into the initial sum in step 2, without material effect on the results.

The other algorithms based on [VONN] that we tested were as follows:

Algorithm 1: This was included in a draft version of RFC 6437, with the following steps:

```
“1.  Split the destination and source addresses into two 64 bit values
     each, thus transforming the 5-tuple into a 7-tuple.
 2.  Add the seven components together using unsigned 64 bit
     arithmetic, discarding any carry bits.
 3.  Apply the von Neumann algorithm to the resulting string of 64
     bits:
     1.  Starting at the least significant end, select two bits.
     2.  If the two bits are 00 or 11, discard them.
     3.  If the two bits are 01, output* a 0 bit.
     4.  If the two bits are 10, output* a 1 bit.
     5.  Repeat with the next two bits in the input 64 bit string.
     6.  Stop when 20 bits have been output (or when the 64 bit string
         is exhausted).
 4.  In the highly unlikely event that the result is exactly zero, set
     the flow label arbitrarily to the value 1.”
```

Algorithm 3:

```
1.  Split the destination and source addresses into two 64 bit values
    each, thus transforming the 5-tuple into a 7-tuple.
2.  Shift the port numbers 48 bits left (as 64 bit quantities).
3.  Add the seven components together using unsigned 64 bit
    arithmetic, discarding any carry bits.
4.  Apply the von Neumann algorithm to the resulting string of 64
    bits:
    1.  Starting at the least significant end, select two bits.
    2.  If the two bits are 00 or 11, discard them.
    3.  If the two bits are 01, output a 0 bit.
    4.  If the two bits are 10, output a 1 bit.
    5.  Repeat with the next two bits in the input 64 bit string.
    6.  Stop when 16 bits have been output (or when the 64 bit string
        is exhausted).
5.  Shift the resulting value 4 bits left and mask with 0xfffff.
6.  In the highly unlikely event that the result is exactly zero, set
    the flow label arbitrarily to the value 1.
```

Algorithm 3b: This was a faulty version of Algorithm 3, in which the two port numbers were added in a second time, as in Algorithm 2.

In ***Algorithm 4***, the input to the von Neumann algorithm was 96 bits - the sum of the two port numbers as the low order 16 bits and the sum of the other fields as the high order 64 bits. The maximum loop was 20.

Algorithm 5 was similar to Algorithm 4, but with the port numbers as as the high order 16 bits.

Algorithm 6 was the same as Algorithm 2, except that the high order halves of the two addresses were omitted.

Algorithm 7 was Algorithm 6 but with the port numbers entirely omitted.

Algorithm 8 was a complete change of tack, leaving the von Neumann algorithm behind. Instead of using the von Neumann algorithm, it uses the Fowler/Noll/Vo (FNV) algorithm [FNV], and specifically 32-bit FNV1a with the result XOR-folded to 20 bits. The whole IPv6 header 5-tuple is fed into the algorithm, one byte at a time, in the following order:

```
The source address, starting with the low-order byte.
The destination address, starting with the low-order byte.
The protocol number.
The source port, starting with the low-order byte.
The destination port, starting with the low-order byte.
```

As usual, in the highly unlikely event that the result is exactly zero, set the flow label arbitrarily to the value 1.

The FNV algorithm involves multiplication and the use of some large numerical constants, so we expected it to be noticeably slower than any of the von Neumann algorithms.

Algorithm 9 was a trivial variant of Algorithm 8, in which FNV's arbitrary "offset basis" value was changed by a few units to investigate whether this had any impact on the algorithm's performance.

Datasets

We decided to work only on TCP traffic, because TCP flows can be recognised trivially and it is sufficient to capture packets containing only a SYN flag (i.e., not SYN/ACK packets) in order to have one packet from each outgoing or incoming flow. Other methods of identifying new flows are much more complex. One side-effect is that if the TCP initiator receives no SYN/ACK, it will repeat the same SYN packet several times - this proved to be quite common in the traces obtained, but was not counted as a hash collision. Of course, all SYN packets captured have the same protocol number, but since TCP is the dominant protocol on the Internet we did not consider this to be a significant disadvantage.

The trace files were collected using *tcpdump, Wireshark* or equivalent tools, in standard *pcap (bpf)* format. An appropriate capture filter is

ip6 and tcp and ip6[53:1]=2

(Note that the *tcp[tcpflags]* construct does not work for IPv6.) An appropriate Wireshark display filter is

ipv6 and tcp.flags.syn == 1 and tcp.flags.ack != 1

The traces we obtained were as follows:

Trace 1 2011-05-19 Brian Carpenter's desktop traffic

Trace 2 2011-05-31 Brian Carpenter's desktop traffic

Trace 3 2011-05-30/31 University of Auckland site border (3703 SYN packets)

Trace 4 2011-05-31 Griffith College, Dublin (107340 SYN packets)

Trace 5 2011-05-31 potaroo.net (25486 SYN packets)

Trace 6 2011-12-30 Chinese ISP (7319279 SYN packets)

The first two were short traces used for software debugging; the others were substantial files of production traffic, with the packets truncated for privacy reasons.

Methods of Analysis

First, a preliminary analysis was made of all algorithms against traces 3, 4 and/or 5. For this, we used a *ruby* program that itself depends on the *RubyLibtrace* library. The *ruby* program loops over all packets in a given trace file. For each packet, it performs the following steps:

- Extract the 5-tuple from the packet.
- Call the hash function under test, which returns a 20 bit hash value.
- If the hash value is new, adds a new entry containing the 5-tuple to a hash table.
- If the hash value is not new and the 5-tuple is different, counts a collision.

At the end, the program outputs the list of hash codes generated with their collision counts (1 = no collision) and some summary statistics. The hash code distribution may then easily be plotted as a histogram or otherwise analysed.

The actual hash algorithm is a *ruby* function coded in *C*; a variant was coded for each of the 9 algorithms described above.

The results from this preliminary analysis allowed a visual judgment of the uniformity or otherwise of the distribution produced by a given algorithm.

Second, a more complete test harness was built which could conveniently test any algorithm against any trace, including performing a Kolmogorov-Smirnov test for statistical uniformity (using the R language implementation of this test [RPROJ]), and the measurement of CPU time required by each algorithm. Due to memory limits, trace 6 had to be split into 7 pieces for analysis, with the results recombined afterwards. The CPU time measurements were made on a dedicated PC running Linux, using *getrusage()*. The absolute values of CPU time are not significant; our interest was only in comparative times for different algorithms on the same hardware.

The results from this test harness allowed an objective comparison of the uniformity of distribution of a given algorithm and the relative performance of different algorithms.

Results – preliminary

Trace 3

Algorithm	Distinct hashes	Hashes with collisions	Total collisions	Max collisions per hash	Collision rate
1	1571	519	1434	15	33%
2	2710	104	145	5	4%
3	1698	538	1366	28	32%
3b	2746	70	90	4	2.5%
8	2807	9	12	4	0.3%

Trace 4

Algorithm	Distinct hashes	Hashes with collisions	Total collisions	Max collisions per hash	Collision rate
1	14330	7088	90152	235	49%
2	48452	28953	55999	12	60 %
3	7421	4231	97789	945	57%
3b	49669	29160	54865	14	58 %
4	16142	7409	88124	384	46%
5	7803	4256	97335	1193	55%
6	46575	28094	57912	16	60%
7	1523	1274	104975	8349	84%
8	98288	4697	4987	9	5%

Trace 5

Algorithm	Distinct hashes	Hashes with collisions	Total collisions	Max collisions per hash	Collision rate
1	12902	3525	9595	33	27%
2	15249	3535	5868	12	23%
3	13244	3757	9254	28	32%
3b	16217	3042	4604	12	19%
8	19658	180	227	4	0.9%

From visual inspection of these results, it seems clear that for traces 3 and 5, Algorithm 2 is the best of the von Neumann algorithms (fewer collisions). Algorithm 3b, the faulty version of Algorithm 3, is unintentionally almost identical to Algorithm 2. For Trace 4, Algorithm 2 is also probably the best choice; although it has a high collison rate, so do all the others, but Algorithm 2 has the lowest maximum number of collisions per hash.

It is clear on sight that Algorithm 8, FNV1a-32, is considerably superior to Algorithm 2.

To complete the preliminary comparison, here are histograms showing the hash value distribution for Trace 4 for the two most promising algorithms. These figures visually confirm that Algorithm 8 appears to produce a far more uniform distribution than Algorithm 2. Other histograms confirm that Algorithm 8 appears much better than *any* of the von Neumann algorithms, and that Algorithm 9 appears to perform identically to Algorithm 8.

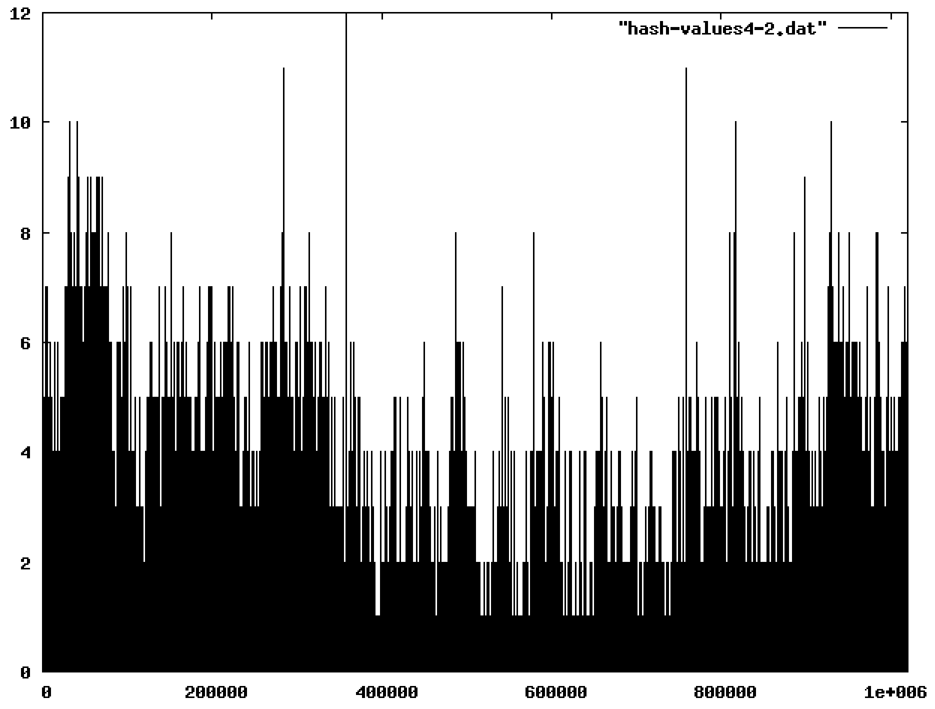

Fig. 1 *Hash value distribution for Algorithm 2 applied to Trace 4.*

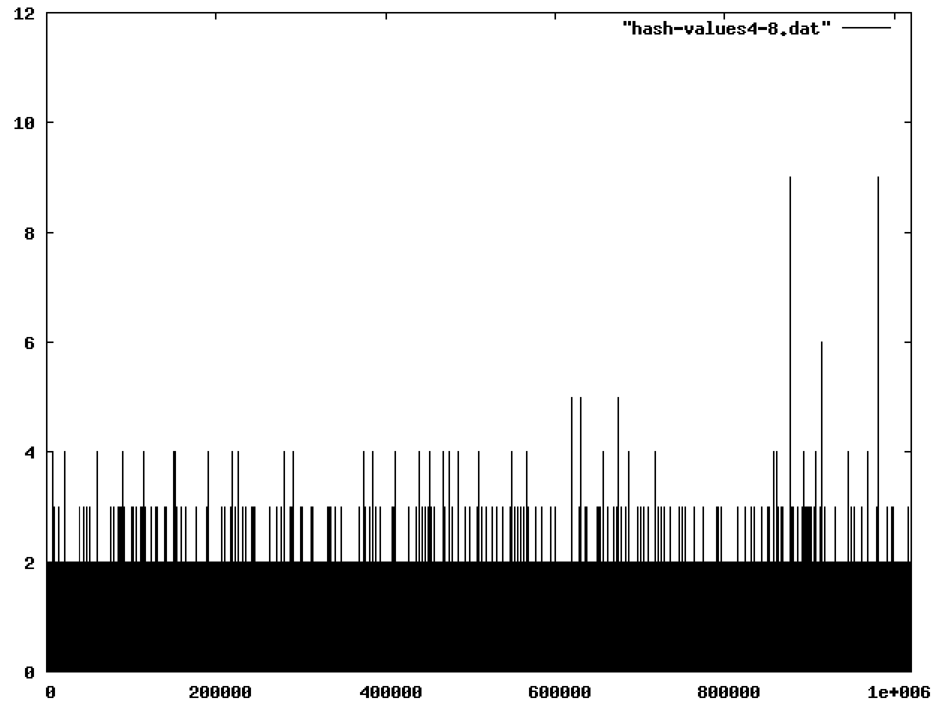

Fig. 2 *Hash value distribution for Algorithm 8 applied to Trace 4.*

Results - objective

Using the test harness, uniformity and performance were tested for various combinations of algorithm and dataset.

Algorithm 3 was used as representative of the von Neumann algorithms, and compared with the two FNV algorithms. Firstly we show the computed D-statistic for the Kolmogorov-Smirnov 2-way test, when comparing the observed distribution against a perfectly uniform distribution. In an idealised world, a D-statistic of zero would indicate that the observed distribution was perfectly uniform. The corresponding p values of the K-S test are also shown, but their interpretation is an art as much as it is science.

Trace file	Algorithm 3		Algorithm 8		Algorithm 9	
	D	p	D	p	D	p
3	0.4084	2E-16	0.0417	0.9985	0.0246	0.0657
4	0.2512	2E-16	0.0021	0.7446	0.003	0.3232
5	0.2861	2E-16	0.0058	0.5232	0.00104	0.02386
6	0.2626	2E-16	0.0004	0.8344	0.0004	0.644

The D values may be interpreted by calculating the critical values of the D-statistic for the various sample sizes. At the 95% confidence level, the critical value is $1.36/\sqrt{N}$ where N is the sample size. Thus we obtain:

Trace file	Sample size N	Critical value @ 95%
3	3703	0.0223
4	107340	0.0042
5	25486	0.0085
6	7319279	0.0005

An observed D value less than the critical value is interpreted to imply a 95% confidence level that the distribution is uniform. We may conclude that the distributions produced by Algorithms 8 and 9 are indistinguishable from the uniform distribution at the 95% confidence level in 5 cases, but with somewhat lower confidence in 3 cases. By contrast, Algorithm 3 in no case reaches a reasonable confidence level. This is confirmed by its very low p-value in all cases.

We can also compare the algorithms graphically, to better illustrate what these numbers mean. The graphs below show the cumulative distribution functions for Algorithm 3 (left) and 8 (right), for trace file 6; as expected for a uniform distribution, Algorithm 8 shows a straight line. Algorithm 9 is not shown, but appears identical. In contrast, Algorithm 3 deviates significantly from a straight line due to its lack of uniformity.

We feel confident in concluding that the Algorithms 8 and 9 provide a very good approximation to a uniform distribution, and that Algorithm 3 (and the other von Neumann-based algorithms) do not.

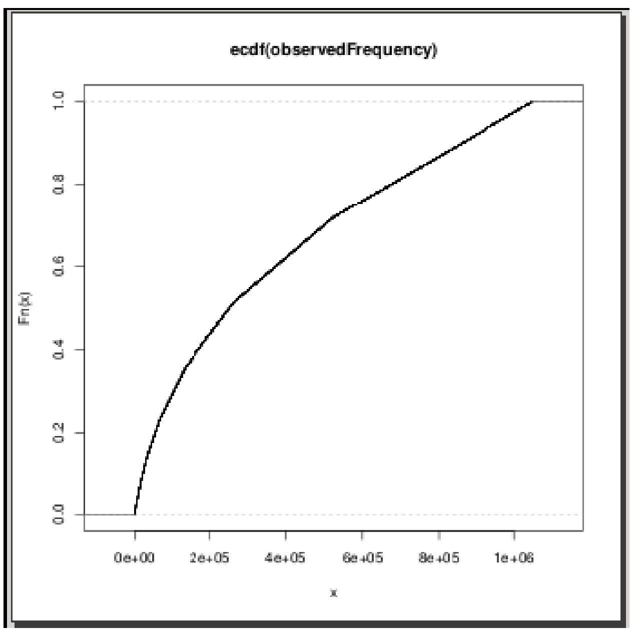

Fig. 3: *CDF for Algorithm 3*

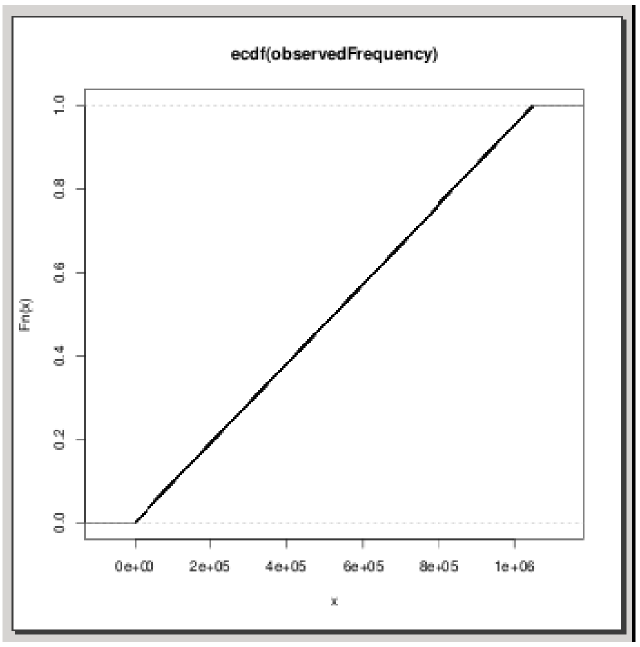

Fig. 4: *CDF for Algorithm 8*

As far as timing goes, the absolute values that we measured are of limited interest as they concern a pure software implementation of the algorithms on general purpose PC hardware (Intel® Core©2 CPU, 800 Mhz RAM) using Ubuntu 11.04 (Natty), Linux 2.6.38-13-generic kernel and the *gcc* compiler. Nevertheless we present the measured values for the largest trace files in the following table. The times are of the order of 100 µs per packet, but no effort was made to optimise the code so these values should not be considered meaningful except for comparison among themselves. It is the relative speed of the various algorithms that is interesting.

Trace file	Packet count	Algorithm 3 CPU sec.	Algorithm 3 µs/packet	Algorithm 8 CPU sec.	Algorithm 8 µs/packet	Algorithm 9 CPU sec.	Algorithm 9 µs/packet
4	107340	7.196449	67	4.476279	41	5.388336	50
5	25486	1.232076	48	0.996062	39	0.924057	36
6	7319279	646.6	88	1046.84	143	985.46	135

A noticeable aspect is that Algorithm 3 is slowest for the smaller datasets but fastest for the largest dataset. A characteristic of datasets 4 and 5 is that they were recorded close to specific sites, such that one address in every 5-tuple has many of its bits in common. Dataset 6 is from a provider's network, so such commonality among the address bits is much rarer. This appears to indicate an important point - the computing time of the various algorithms is noticeably data-dependent. For this reason, we do not combine these performance results statistically. We consider that the result for dataset 6 is the most significant as far as mixed traffic is concerned: Algorithms 8 and 9 differ in execution time only by about 6%, but Algorithm 3 is some 37% faster. However, this result cannot be assumed valid for traffic directed from or to a specific site, such that the address bits are less variable. Overall, however, the difference in computing time between the von Neumann and FNV approaches is relatively modest and, due to its data-dependence, clearly not a deciding factor.

Another point to note is that changing the FNV offset basis value (the difference between Algorithms 8 and 9) does have a noticeable impact on performance, in the range -8% to +20% for the datasets listed above. Again, this effect is data-dependent, not merely statistical.

Predictability

If all nodes use the same algorithm, they will all generate exactly the same flow label value for the same IP header 5-tuple. From a security viewpoint, this is undesirable, because it allows a malicious third party to predict flow label values for future flows, if it can predict the 5-tuple (for example, when flows use consecutive port numbers).

Such predictability can be avoided, even without a cryptographic hash function, if a node that is generating flow labels includes a node-specific nonce in the input to the hash function. Such a nonce could be generated once during system start-up and retained until the node is restarted (at which point all bets are off for sesssions in progress anyway).

In the present study, this mechanism was simulated by the difference between Algorithms 8 and 9. Algorithm 8 used the standard offset basis for FNV1a-32, 2166136261. Algorithm 9 replaced this by 2166136263. This produced a completely different set of hash values but with an equally uniform distribution. Thus, a locally generated and secret offset basis is sufficient to avoid predictability by a third party.

Conclusion

The principal conclusion is that the FNV1a-32 algorithm, applied to an IPv6 header 5-tuple considered as a string of bytes, with the result XOR-folded to 20 bits, appears very suitable for use as a stateless hash function for the generation of IPv6 flow labels, as recommended by [RFC6437]. It produces an observably uniform distribution of values, and has a computing time not very different from the superficially simpler von Neumann algorithms also tested. The latter, however, do not in fact produce uniform distributions and are not recommended.

A secondary conclusion is that both uniformity and computing time are data-dependent. In particular, an algorithm that is best for use on or near a single site (where most packets will have some address bits in common) may not be best for use in the heart of the network (where addresses will be completely diverse). Even so, FNV seems to be a reasonable choice.

When unpredictability is required, this can be achieved by using a locally generated nonce as the offset basis for FNV1a-32.

Acknowledgements

Gavin McCullagh provided a trace from Griffith College, Dublin. Geoff Huston (APNIC) provided a trace from potaroo.net, and Zhang Qianli (Network Research Centre, Tsinghua University) provided an ISP trace from China. Hu Qinwen (University of Auckland) assisted us in retrieving the latter.

Cris Calude suggested the von Neumann algorithm.

David Malone noticed a bug in the original Algorithm 3 code.

Donald Eastlake pushed us to look at the FNV algorithm.

References

[RFC2460] S. Deering, R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460, December 1998.

[RFC6436] S. Amante, B. Carpenter, and S. Jiang, *Rationale for Update to the IPv6 Flow Label Specification,* RFC 6436, November 2011

[RFC6437] S. Amante, B. Carpenter, S. Jiang and J. Rajahalme, *IPv6 Flow Label Specification*, RFC 6437, November 2011

[RFC6438] S. Amante, and B. Carpenter, *Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels* , RFC 6438, November 2011

[BALANCE] B. Carpenter, S. Jiang, W. Tarreau, *Using the IPv6 Flow Label for Server Load Balancing*, draft-carpenter-v6ops-label-balance, work in progress, 2012.

[FNV] G. Fowler, L. Knoll, K. Vo, D. Eastlake, *The FNV Non-Cryptographic Hash Algorithm*, draft-eastlake-fnv-02.txt, work in progress, 2011.
Also see http://www.isthe.com/chongo/tech/comp/fnv/index.html.

[GONT] F.Gont, *Security Assessment of the IPv6 Flow Label*, draft-gont-6man-flowlabel-security, work in progress, 2012.

[RPROJ] *The R Project for Statistical Computing*, http://www.r-project.org/

[VONN] J. von Neumann, *Various techniques used in connection with random digits*, National Bureau of Standards Applied Math Series 12, 36-38, 1951.