Internet Draft
Document: <draft-niccolini-hash-descr-00.txt>
Expires: April 2004

S. Niccolini
University of Pisa
M. Molina
NEC Europe Ltd.
Nick Duffield
AT&T Labs -
- Research

October 2003

Hash functions description for packet selection

Status of this Memo

Abstract

This document is concerned with the specification and properties of Hash Functions for packet selection in PSAMP. Having a detailed and exhaustive description of the Hash Function, in terms of input parameters, output value, selection range and internal operations is fundamental for coherent configuration of the same hash based selection at different points in the network. This coherency is fundamental for all network measurement applications needing consistent packet selection.

Table of Contents

1. Introduction.....2
1.1 Hash-based Packet Selection.....2
1.2 Consistent Packet Selection.....3

2. Criteria for PSAMP Hash Functions.....3
3. Hash function description.....4
3.1 MMH (Multilinear Modular Hashing) function.....4
3.1.1 Input parameters.....5
3.1.2 Built in parameters.....5
3.1.3 Output.....5
3.1.4 Functioning.....5
3.2 "Bob" hash function.....7
3.2.1 Input Parameters.....7
3.2.2 Built in parameters.....7
3.2.3 Output.....8
3.2.4 Functioning.....8
4. References.....10
5. Author's Addresses.....11
6. Intellectual Property Statement.....11
7. Full Copyright Statement.....11

1. Introduction

This document is concerned with the specification and properties of Hash Functions for packet selection in PSAMP. Having a detailed and exhaustive description of the Hash Function, in terms of input parameters, output value, selection range and internal operations is fundamental for coherent configuration of the same hash based selection at different points in the network. This coherency is fundamental for all network measurement applications needing consistent packet selection.

1.1 Hash-based Packet Selection

The PSAMP framework document for passive packet measurement [DuFw] divides packet selection techniques in two broad classes: sampling and filtering. Filtering is defined as follows:

> ææa filter is a selection operation that selects a packet deterministically based on the packet content, its treatment, and functions of these occurring in the selection state.

Examples include match/mask filtering, and hash-based selection.ÆÆ

The deterministic property is of particular importance for measurement applications that require the consistent selection of packet at multiple points in the network. This is described more fully in the next section.

This document is concerned with the specification and properties of hash functions to be used for packet selection in PSAMP. We start

by recalling the following definitions of the components of hash-based selection from [DuFw].

* Hash-based selection: a filter specified by a hash domain, a hash function, and hash range and a hash selection range.

* Hash domain: a subset of the packet content and the packet treatment, viewed as an N-bit string for some positive integer N.

* Hash range: a set of M-bit strings for some positive integer M.

* Hash function: a deterministic map from the hash domain into the hash range.

* Hash selection range: a subset of the hash range. The packet is selected if the action of the hash function on the hash domain for the packet yields a result in the hash selection range.

1.2 Consistent Packet Selection

Hash functions are deterministic in nature, in the sense that they always produce the same output from a given input. This is a useful property for some network measurement applications, since it enables the consistent selection of a given packet at different points in the network, in a sense we now describe.
Suppose that a hash-based selector operates at multiple points on a packetÆs path. Consider the following conditions:

1. The same hash function is employed at each point
2. The hash domain is the same at each point and contains only invariant portions of the packet header and/or payload,

i.e., those portions that do not change from point to point
3. the hash selection range is the same at each point.

If, and only if, conditions 1,2, and 3 are all satisfied, then the hash-based selectors at each point are consistent in the sense that a given packet is selected at either all the points or at none. Consistent packet selection using hash functions was proposed in [DuGr01].

This document will specify (in its final form) certain hash functions suitable to be used in the PSAMP protocol.

2. Criteria for PSAMP Hash Functions.

The number of hash functions that could conceivably be used for consistent packet selection is virtually infinite. However, for the applications considered in [DuFw], two properties are particularly important: computation speed and mixing strength. The latter means that small changing in the input should produce big changes in the

---

output (the hash value). Jointly satisfying these two properties is difficult, therefore the choice of a good hash function often implies a trade off decision.

The purpose of this document is to give a formal description of some hash function rather that indicate which is the best one. Nevertheless, we have included some that, according to tests performed [NiTa03], are candidates to satisfy both properties. Other hash functions may be added in later version of this draft, along with better indications about their relative performances.

The description of each hash functions that we propose in this draft comprises the following items:

1. input parameters;
2. built-in parameters;
3. output (including selection range);
4. hash function operation;

The first three items need to be formalized in the information model for packet selection [ZeTech] and the PSAMP MIB [DiRoCla], in order that the hash function, consistently configured, can be activated in the desired measurement points. The last item provides

a consistent reference for the implementation of the hash function by defining the precise mapping that the hash function makes from its input to its output. Some reference code is included for this purpose.

3. Hash function description

3.1 MMH (Multilinear Modular Hashing) function

MMH is a hash function suitable for fast software implementation able to hash a string of variable size [HaKr97].
In order to achieve fast software implementation while preserving the low collision probability, the MMH hash function is built implementing a division-less modular reduction. For sake of simplicity, we refer to the implementation done on 32-bit word machines. The generalization of such implementation may be considered for adapting the MMH to different word length machines, but the reference code presented in 3.1.4 is optimized for 32-bit word machines and requires a compiler that supports the multiplication of two 32-bit words and feeds the result into a 64 bit string (long long type)

3.1.1 Input parameters

The only input parameter of the MMH is:

- the length of the input string (key) to be hashed, in bytes. The MMH operates on elementary blocks of 32 bits, therefore the key length must be a multiple of 4 (if the key is not a multiple of 4, it must be zero-padded)

3.1.2 Built in parameters

The MMH uses the following built-in parameters:

- ro: a prime number in the range of [2^32 ; 2^32 + 2^16]. It is used to implement the division-less modular reduction;
- A vector of (different) 32-bit numbers used to implement the so called inner-product operation. A good choice is to take only prime numbers;
- The number of elements of such a vector, which must be at least equal to the key length divided by 4.

3.1.3 Output

The output of the MMH function is a 32-bit number. It should be specified:
- A 32 bit mask to apply to the output
- The selection range as a list of non overlapping intervals [start value, end value] where value is in [0,2^32]

3.1.4 Functioning

To obtain the hash value, MMH first computes two quantities called inner-product-low and inner-product-high, respectively. These are obtained adding together the results of the multiplication of the words of the input key by the built-in vector.
The two inner products computed are 64-bit long, they need to be firstly modular reduced to the prime number range ([0 ; ro]) and then to the 32-bit range ([0 ; 2^32]). The output of the reduction (and of the function itself) is the hash value.

Reference code:

```
/*----------------------mmh hash function ----------------------------
-*/
```

```
/*
*/
/* The mmh_table size limits the maximum key length that can be hashed
*/
/* With a table of size T we can hash keys up to 4*T bytes long keys
*/
/*
*/
/*----------------------------------------------------------------------
-*/

#define DO(i) sum += key2[i] * (unsigned long long) mmh_table[i]

/* mmh signature table with the first 40 prime numbers: */
/* we can hash string up to 160 char long               */
static unsigned long mmh_table[40] =
  {   2,      3,      5,      7,     11,
```

```
     13,     17,     19,     23,     29,
     31,     37,     41,     43,     47,
     53,     59,     61,     67,     71,
     73,     79,     83,     89,     97,
    101,    103,    107,    109,    113,
    127,    131,    137,    139,    149,
    151,    157,    163,    167,    173};

/**
 **  FUNCTION: mmh
 **  PARAMETERS:
 **  - key: pointer to the char string to be hashed
 **  - len: length of the char string to be hashed
 **  RETURN: hashvalue in an unsigned long variable
 **  PURPOSE: Accepts a pointer to a string to be hashed
 **  and returns an unsigned long.
 **  NOTE: using this function requires that
 **  the key is a 4-byte multiple otherwise
 **  it gives erroneus results. The padding with 0x0 bytes
 **  of keys non multiple of 4 must be performed
 **  before calling this function!
 **/
unsigned long mmh(char *key, unsigned short len)
  {
    unsigned short      i;
    signed long long    stmp;
    unsigned long long  utmp;

    unsigned long long  sum = 0LL;  /*running sum*/

    unsigned long       ret;        /*return value*/

    unsigned long       *key2 = (unsigned long *)key;
```

```
    for (i=0; i<(len/4); i++)
      DO(i);

    /*********return (sum % 0x10000000fLL);***************/

    stmp = (sum & 0xffffffffLL) - ((sum >> 32) * 15);       /*lo - hi
* 15 */
```

```
    utmp = (stmp & 0xffffffffLL) - ((stmp >> 32) * 15);     /*lo - hi
* 15 */

    ret = utmp & 0xffffffff;

    if (utmp > 0x10000000fLL)     /* if larger than p - subtract 15
again */
      ret -=15;

    return ret;
  }
```

3.2 "Bob" hash function

"Bob" hash function is a hash function designed for having each bit of the input affecting every bit of the return value and using both 1-bit and 2-bit deltas to achieve the so called avalanche effect [Jenk97]. This function was originally built for hash table lookup with fast software implementation.

3.2.1 Input Parameters

The input parameters of such a function are:
- the length of the input string (key) to be hashed, in bytes. The elementary input blocks of Bob hash are the single bytes, therefore no padding is needed.
- an init value (an arbitrary 32-bit number).

3.2.2 Built in parameters

The Bob Hash uses the following built-in parameter:

- the golden ratio (an arbitrary 32-bit number used in the hash function computation: its purpose is to avoid mapping all zeros to all zeros);

Note: the mix sub-function (see mix (a,b,c) macro in the reference code in 3.2.4) has a number of parameters governing the shifts in the registers. The one presented is not the only possible choice.

It is an open point whether these may be considered additional built-in parameters to specify at function configuration.

3.2.3 Output

The output of the MMH function is a 32-bit number. It should be specified:
- A 32 bit mask to apply to the output
- The selection range as a list of non overlapping intervals [start value, end value] where value is in [0,2^32]

3.2.4 Functioning

The hash value is obtained computing first an initialization of an internal state (composed of 3 32-bit numbers, called a, b, c in the reference code below), then, for each input byte of the key the internal state is combined by addition and mixed using the mix sub-function. Finally, the internal state mixed one last time and the third number of the state (c) is chosen as the return value.

```
typedef  unsigned long  int  ub4;   /* unsigned 4-byte quantities */
typedef  unsigned       char ub1;   /* unsigned 1-byte quantities */

#define hashsize(n) ((ub4)1<<(n))
#define hashmask(n) (hashsize(n)-1)

/*
--------------------------------------------------------------------
mix -- mix 3 32-bit values reversibly.
For every delta with one or two bits set, and the deltas of all three
  high bits or all three low bits, whether the original value of a,b,c
  is almost all zero or is uniformly distributed,
* If mix() is run forward or backward, at least 32 bits in a,b,c
  have at least 1/4 probability of changing.
* If mix() is run forward, every bit of c will change between 1/3 and
  2/3 of the time.  (Well, 22/100 and 78/100 for some 2-bit deltas.)
mix() was built out of 36 single-cycle latency instructions in a
  structure that could supported 2x parallelism, like so:
      a -= b;
      a -= c; x = (c>>13);
      b -= c; a ^= x;
      b -= a; x = (a<<8);
      c -= a; b ^= x;
      c -= b; x = (b>>13);
      ...
  Unfortunately, superscalar Pentiums and Sparcs can't take advantage
  of that parallelism.  They've also turned some of those single-cycle
  latency instructions into multi-cycle latency instructions
--------------------------------------------------------------------
*/
```

```
#define mix(a,b,c)  \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8); \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12);  \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5); \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}

/*
--------------------------------------------------------------------
hash() -- hash a variable-length key into a 32-bit value
  k       : the key (the unaligned variable-length array of bytes)
  len     : the length of the key, counting by bytes
  initval : can be any 4-byte value
Returns a 32-bit value.  Every bit of the key affects every bit of
the return value.  Every 1-bit and 2-bit delta achieves avalanche.
About 6*len+35 instructions.

The best hash table sizes are powers of 2.  There is no need to do
mod a prime (mod is sooo slow!).  If you need less than 32 bits,
use a bitmask.  For example, if you need only 10 bits, do
  h = (h & hashmask(10));
In which case, the hash table should have hashsize(10) elements.

If you are hashing n strings (ub1 **)k, do it like this:
  for (i=0, h=0; i<n; ++i) h = hash( k[i], len[i], h);

By Bob Jenkins, 1996.  bob_jenkins@burtleburtle.net.  You may use this
code any way you wish, private, educational, or commercial.  It's free.

See http://burtleburtle.net/bob/hash/evahash.html
Use for hash table lookup, or anything where one collision in 2^^32 is
acceptable.  Do NOT use for cryptographic purposes.
--------------------------------------------------------------------
*/

ub4 bob_hash(k, length, initval)
register ub1 *k;        /* the key */
register ub4  length;   /* the length of the key */
register ub4  initval;  /* an arbitrary value */
{
```

```
   register ub4 a,b,c,len;

   /* Set up the internal state */
   len = length;
   a = b = 0x9e3779b9;  /* the golden ratio; an arbitrary value */
   c = initval;         /* another arbitrary value */

   /*---------------------------------- handle most of the key */
```



```
   while (len >= 12)
   {
      a += (k[0] +((ub4)k[1]<<8) +((ub4)k[2]<<16) +((ub4)k[3]<<24));
      b += (k[4] +((ub4)k[5]<<8) +((ub4)k[6]<<16) +((ub4)k[7]<<24));
      c += (k[8] +((ub4)k[9]<<8) +((ub4)k[10]<<16)+((ub4)k[11]<<24));
      mix(a,b,c);
      k += 12; len -= 12;
   }

   /*------------------------------ handle the last 11 bytes */
   c += length;
   switch(len)       /* all the case statements fall through */
   {
   case 11: c+=((ub4)k[10]<<24);
   case 10: c+=((ub4)k[9]<<16);
   case 9 : c+=((ub4)k[8]<<8);
      /* the first byte of c is reserved for the length */
   case 8 : b+=((ub4)k[7]<<24);
   case 7 : b+=((ub4)k[6]<<16);
   case 6 : b+=((ub4)k[5]<<8);
   case 5 : b+=k[4];
   case 4 : a+=((ub4)k[3]<<24);
   case 3 : a+=((ub4)k[2]<<16);
   case 2 : a+=((ub4)k[1]<<8);
   case 1 : a+=k[0];
     /* case 0: nothing left to add */
   }
   mix(a,b,c);
   /*-------------------------------- report the result */
   return c;
}
```

4. References

[DuFw] N. Duffield: A Framework for Passive Packet Measurement, Internet Draft, <draft-ietf-psamp-framework-03.txt>, work in progress, June 2003

[DuGr01] N. G. Duffield and M. Grossglauser, Trajectory Sampling for Direct Traffic Observation, IEEE/ACM Trans. on Networking, 9(3), 280-292, June 2001.

[ZeTech] T.Zseby, M.Molina, F.Raspall, N.Duffield: Sampling and Filtering Techniques for IP Packet Selection, Internet Draft < draft-ietf-psamp-sample-tech-02.txt>, work in progress, June 2003

[DiRoCla] T.Dietz, D.Romascanu, B. Claise: Definitions of Managed Objects for Packet Sampling, Internet Draft <draft-ietf-psamp-mib-00.txt>, work in progress, June 2003

[Jenk97] B. Jenkins: Algorithm Alley, Dr. Dobb's Journal, September 1997. http://burtleburtle.net/bob/hash/doobs.html

[HaKr97] S. Halevi, H. Krawczyk : MMH: Software Message Authentication in the Gbit/second Rates, LNCS vol. 1267, Springer, 1997. Pages 172-189

[NiTa03] S. Niccolini, S. Tartarelli, F. Raspall, M. Molina : On time synchronization and hashing for passive One-Way Delay measurements, work in progress, available at http://www.ccrle.nec.de/mmolina/papers/paper_hash_sync.pdf

5. Author's Addresses

Saverio Niccolini
University of Pisa
Via Caruso 1
56122 Pisa
Italy
Phone: +39 050 2217-592
EMail: s.niccolini@iet.unipi.it

Maurizio Molina
NEC Europe Ltd., Network Laboratories

Adenauerplatz 6
69115 Heidelberg
Germany
Phone: +49 6221 90511-18
Email: molina@ccrle.nec.de

Nick Duffield
AT&T Labs - Research
Room B-139
180 Park Ave
Florham Park NJ 07932, USA
Phone: +1 973-360-8726
Email: duffield@research.att.com

6. Intellectual Property Statement

AT&T Corporation may own intellectual property applicable to this contribution. The IETF has been notified of AT&T's licensing intent for the specification contained in this document. See http://www.ietf.org/ietf/IPR/ATT-GENERAL.txt for AT&T's IPR

statement.

7. Full Copyright Statement