SipHash: a fast short-input PRF

Jean-Philippe Aumasson, Daniel J. Bernstein

SipHash: a fast short-input MAC

Jean-Philippe Aumasson, Daniel J. Bernstein

UMAC

(Black, Halevi, Krawczyk, Krovetz, Rogaway; 2000)

	43 bytes	256 bytes	1500 bytes	256 kbytes
UMAC32	**16.3**	**3.8**	**2.1**	**1.9**
UMAC-STD	52.9	12.3	3.8	1.9
UMAC16	**14.0**	**2.7**	**1.2**	**1.0**
UMAC-MMX	35.9	4.5	1.7	1.0

http://fastcrypto.org/umac/update.pdf

1 cycle/byte on a Pentium III !

$$\text{UMAC}(m) = H(k1, m) \oplus \text{AES}(k2, n)$$

UMAC's universal hash

Polynomial-evaluation using 64-bit multipliers with Horner's rule

$$\left(\sum_{i=1}^{\ell/2} ((m_{2i-1} + k_{2i-1}) \bmod 2^w) \cdot ((m_{2i} + k_{2i}) \bmod 2^w)) \right) \bmod 2^{2w}$$

UMAC fast C implementation

2000+ LoC

(without AES)

Not portable

```c
int uhash(uhash_ctx_t ahc, char *msg, long len, char *res)
/* assumes that msg is in a writable buffer of length divisible by */
/* L1_PAD_BOUNDARY. Bytes beyond msg[len] may be zeroed.           */
{
    UINT8 nh_result[STREAMS*sizeof(UINT64)];
    UINT32 nh_len;
    int extra_zeroes_needed;

    /* If the message to be hashed is no longer than L1_HASH_LEN, we skip
     * the polyhash.
     */
    if (len <= L1_KEY_LEN) {
        if (len == 0)                     /* If zero length messages will not */
            nh_len = L1_PAD_BOUNDARY;  /* be seen, comment out this case    */
        else
            nh_len = ((len + (L1_PAD_BOUNDARY - 1)) & ~(L1_PAD_BOUNDARY - 1));
        extra_zeroes_needed = nh_len - len;
        zero_pad((UINT8 *)msg + len, extra_zeroes_needed);
        nh(&ahc->hash, (UINT8 *)msg, nh_len, len, nh_result);
        ip_short(ahc,nh_result, res);
    } else {
        /* Otherwise, we hash each L1_KEY_LEN chunk with NH, passing the NH
         * output to poly_hash().
         */
        do {
            nh(&ahc->hash, (UINT8 *)msg, L1_KEY_LEN, L1_KEY_LEN, nh_result);
            poly_hash(ahc,(UINT32 *)nh_result);
            len -= L1_KEY_LEN;
            msg += L1_KEY_LEN;
        } while (len >= L1_KEY_LEN);
```

```
* ------------------------------------------------------------------- */

/* ////////////////////////// IMPORTANT NOTES /////////////////////////////////////
 *
 * 1)  This version does not work properly on messages larger than 16MB
 *
 * 2)  If you set the switch to use SSE2, then all data must be 16-byte
 *     aligned
 *
 * 3) When calling the function umac(), it is assumed that msg is in
 * a writable buffer of length divisible by 32 bytes. The message itself
 * does not have to fill the entire buffer, but bytes beyond msg may be
 * zeroed.
 *
```

http://fastcrypto.org/umac/2004/src/umac.c

UMAC uses a PRG to expand the key to 33280 bits

Using a PRG, map Key to $K = K_1 K_2 \cdots K_{1024}$, with each K_i a 32-bit word, and to A, where $|A| = 512$.

RFC4418 replaces UMAC's PRG with an AES-based KDF…

```
3.2.1.    KDF Algorithm

    Input:
      K, string of length KEYLEN bytes.
      index, a non-negative integer less than 2^64.
      numbytes, a non-negative integer less than 2^64.
    Output:
      Y, string of length numbytes bytes.

    Compute Y using the following algorithm.

      //
      // Calculate number of block cipher iterations
      //
      n = ceil(numbytes / BLOCKLEN)
      Y = <empty string>

      //
      // Build Y using block cipher in a counter mode
      //
      for i = 1 to n do
        T = uint2str(index, BLOCKLEN-8) || uint2str(i, 8)
        T = ENCIPHER(K, T)
        Y = Y || T
      end for

      Y = Y[1...numbytes]

      Return Y
```

… and uses AES and this KDF in a "Pad-Derivation Function"

```
3.3.1.     PDF Algorithm

Input:
   K, string of length KEYLEN bytes.
   Nonce, string of length 1 to BLOCKLEN bytes.
   taglen, the integer 4, 8, 12 or 16.
Output:
   Y, string of length taglen bytes.

Compute Y using the following algorithm.

   //
   // Extract and zero low bit(s) of Nonce if needed
   //
   if (taglen = 4 or taglen = 8)
     index = str2uint(Nonce) mod (BLOCKLEN/taglen)
     Nonce = Nonce xor uint2str(index, bytelength(Nonce))
   end if

   //
   // Make Nonce BLOCKLEN bytes by appending zeroes if needed
   //
   Nonce = Nonce || zeroes(BLOCKLEN - bytelength(Nonce))

   //
   // Generate subkey, encipher and extract indexed substring
   //
   K' = KDF(K, 0, KEYLEN)
   T = ENCIPHER(K', Nonce)
   if (taglen = 4 or taglen = 8)
     Y = T[1 + (index*taglen) ... taglen + (index*taglen)]
   else
     Y = T[1...taglen]
   end if

   Return Y
```

SipHash

Simple ARX round function

Simple JH-like message injection

No key expansion

No external primitive

No state between messages

SipHash initialization

256-bit state v0 v1 v2 v3

128-bit key k0 k1

$v0 = k0 \oplus 736f6d6570736575$

$v1 = k1 \oplus 646f72616e646f6d$

$v2 = k0 \oplus 6c7967656e657261$

$v3 = k1 \oplus 7465646279746573$

SipHash initialization

256-bit state v0 v1 v2 v3

128-bit key k0 k1

$v0 = k0 \oplus$ "somepseu"

$v1 = k1 \oplus$ "dorandom"

$v2 = k0 \oplus$ "lygenera"

$v3 = k1 \oplus$ "tedbytes"

SipHash compression

Message parsed as 64-bit words **m0**, **m1**, …

$v3 \oplus=$ **m0**

c iterations of SipRound

$v0 \oplus=$ **m0**

SipHash compression

Message parsed as 64-bit words **m0**, **m1**, …

$v3 \oplus= $ **m1**

c iterations of SipRound

$v0 \oplus= $ **m1**

SipHash compression

Message parsed as 64-bit words **m0**, **m1**, …

v3 $\oplus$= **m2**
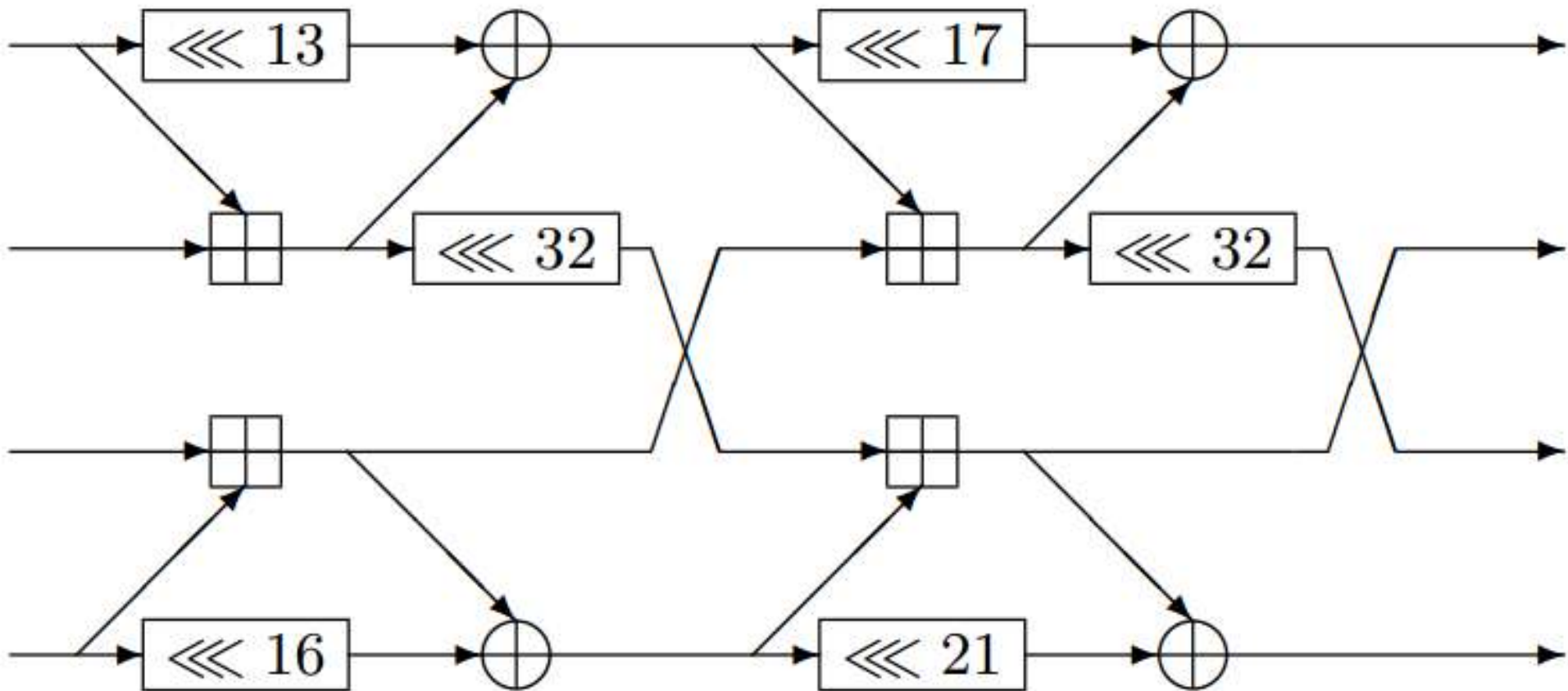
c iterations of SipRound

v0 $\oplus$= **m2**

SipHash compression

Message parsed as 64-bit words **m0**, **m1**, …
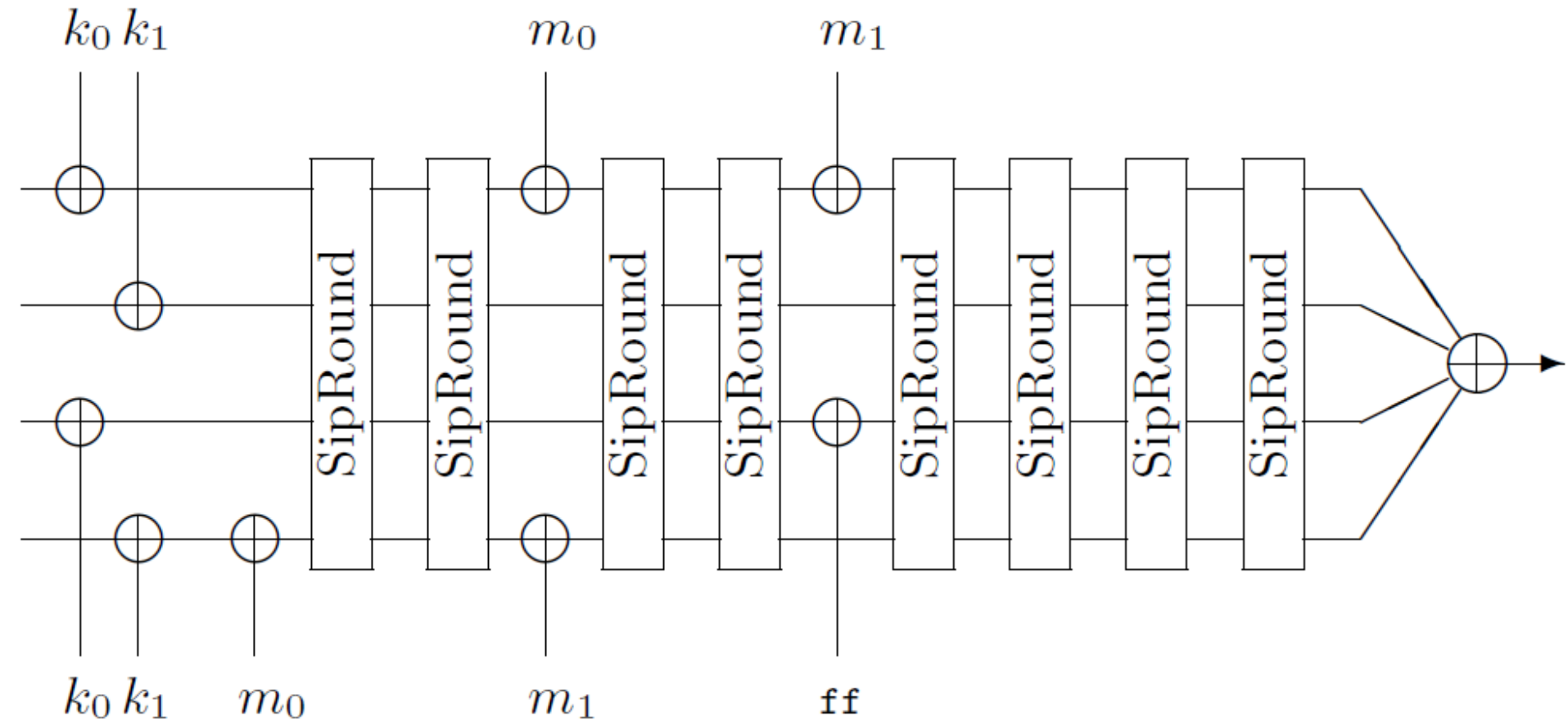
Etc.

SipRound

SipHash finalization

v2 $\oplus$= 255

d iterations of SipRound

Return v0 $\oplus$ v1 $\oplus$ v2 $\oplus$ v3

SipHash-2-4 hashing 15 bytes

Family SipHash-**c**-**d**

Fast proposal: SipHash-**2**-**4**

Conservative proposal: SipHash-**4**-**8**

Weaker versions for cryptanalysis:

SipHash-1-0, SipHash-2-0, etc.

SipHash-1-1, SipHash-2-1, etc.

Etc.

(Many) short inputs?

Filter: tcp.dstport eq 80 or tcp.dstport eq 443   ▼   Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Length	Info
2373	140.671983			TCP	66	[TCP Keep-Alive] 53517 > http [ACK] Seq=1 Ack=1 Win=1104 Len=0 TSval=76672448 TSecr=18065658
2519	149.923984			TCP	66	53518 > http [ACK] Seq=2 Ack=2 Win=255 Len=0 TSval=76674761 TSecr=1806619891
2542	151.325449			HTTP	831	Continuation or non-HTTP traffic
2543	151.325475			HTTP	1333	Continuation or non-HTTP traffic
2549	151.504048			TCP	54	45886 > http [ACK] Seq=8269 Ack=40432 Win=65535 Len=0
2551	151.504058			TCP	54	45886 > http [ACK] Seq=8269 Ack=41120 Win=65535 Len=0
2553	151.509701			TCP	54	45886 > http [ACK] Seq=8269 Ack=42480 Win=65535 Len=0
2555	151.509713			TCP	54	45886 > http [ACK] Seq=8269 Ack=43168 Win=65535 Len=0
2557	151.517987			TCP	54	45886 > http [ACK] Seq=8269 Ack=44528 Win=65535 Len=0
2559	151.517999			TCP	54	45886 > http [ACK] Seq=8269 Ack=45216 Win=65535 Len=0
2561	151.524712			TCP	54	45886 > http [ACK] Seq=8269 Ack=46576 Win=65535 Len=0
2563	151.524725			TCP	54	45886 > http [ACK] Seq=8269 Ack=47264 Win=65535 Len=0
2565	151.527101			TCP	54	45886 > http [ACK] Seq=8269 Ack=48624 Win=65535 Len=0
2567	151.527112			TCP	54	45886 > http [ACK] Seq=8269 Ack=49312 Win=65535 Len=0
2569	151.532604			TCP	54	45886 > http [ACK] Seq=8269 Ack=50672 Win=65535 Len=0
2571	151.532612			TCP	54	45886 > http [ACK] Seq=8269 Ack=51360 Win=65535 Len=0
2573	151.535491			TCP	54	45886 > http [ACK] Seq=8269 Ack=52720 Win=65535 Len=0
2575	151.535503			TCP	54	45886 > http [ACK] Seq=8269 Ack=53408 Win=65535 Len=0
2577	151.537818			TCP	54	45886 > http [ACK] Seq=8269 Ack=54768 Win=65535 Len=0
2579	151.537828			TCP	54	45886 > http [ACK] Seq=8269 Ack=55456 Win=65535 Len=0
2583	151.543724			TCP	54	45886 > http [ACK] Seq=8269 Ack=57504 Win=65535 Len=0
2585	151.548877			TCP	54	45886 > http [ACK] Seq=8269 Ack=58501 Win=65535 Len=0
2645	155.535986			TCP	66	53517 > http [ACK] Seq=2 Ack=2 Win=1104 Len=0 TSval=76676164 TSecr=1806625501
2682	158.265727			TCP	66	53517 > http [FIN, ACK] Seq=2 Ack=2 Win=1104 Len=0 TSval=76676846 TSecr=1806625501
2684	158.265970			TCP	66	53518 > http [FIN, ACK] Seq=2 Ack=2 Win=255 Len=0 TSval=76676846 TSecr=1806619891
3306	196.607981			TCP	54	[TCP Keep-Alive] 45886 > http [ACK] Seq=8268 Ack=58501 Win=65535 Len=0

Type: IP (0x0800)
▷ Internet Protocol Version 4, Src: , Dst:
▼ Transmission Control Protocol, Src Port: 53518 (53518), Dst Port: http (80), Seq: 2, Ack: 2, Len: 0
    Source port: 53518 (53518)
    Destination port: http (80)
    [Stream index: 0]
    Sequence number: 2   (relative sequence number)

Hash tables

```
h = {}                 # empty table
h['foo'] = 'bar'       # insert 'bar'
Print h['foo']         # lookup
```

Non-crypto functions to produce 'foo':

```
for (; nKeyLength > 0; nKeyLength -=1) {
hash = ((hash << 5) + hash) + *arKey++;
}
```

Hash flooding attacks

Multicollisions forcing worst-case complexity of $\Theta(n^2)$, instead of $\Theta(n)$

[when table implemented as linked lists]

djbdns/cache.c, 1999

```c
    nextpos = prevpos ^ get4(pos);
    prevpos = pos;
    pos = nextpos;
    if (++loop > 100) return 0; /* to protect against hash flooding */
  }

  return 0;
}
```

USENIX 2003

Denial of Service via Algorithmic Complexity Attacks

Scott A. Crosby
scrosby@cs.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science, Rice University

Abstract

We present a new class of low-bandwidth denial of service attacks that exploit algorithmic deficiencies sume $O(n)$ time to insert n elements. However, if each element hashes to the same bucket, the hash table will also degenerate to a linked list, and it will take $O(n^2)$ time to insert n elements.

Vulnerabilities in Perl, web proxy, IDS

CCC 2011

Effective Denial of Service attacks against web application platforms

We are the 99% (CPU usage)

This talk will show how a common flaw in the implementation of most of the popular web programming languages and platforms (including PHP, ASP.NET, Java, etc.) can be (ab)used to force web application servers to use 99% of CPU for several minutes to hours for a single HTTP request.

SPEAKERS

Alexander 'alech' Klink

Julian | zeri

SCHEDULE

Day	Day 2 - 2011-12-28
Room	Saal 1
Start time	14:00
Duration	01:00

INFO

ID	4680
Event type	Lecture
Track	Hacking
Language used for presentation	English

Affected: PHP, ASP.net, Python, etc.

n.runs AG

http://www.nruns.com/ security(at)nruns.com

n.runs-SA-2011.004 28-Dec-2011

_____

Vendors: PHP, http://www.php.net

Oracle, http://www.oracle.com

Microsoft, http://www.microsoft.com

Python, http://www.python.org

Ruby, http://www.ruby.org

Google, http://www.google.com Affected Products: PHP 4 and 5

Java

Apache Tomcat

Apache Geronimo

Jetty

Oracle Glassfish

ASP.NET

Python

Plone

CRuby 1.8, JRuby, Rubinius

v8

Vulnerability: Denial of Service through hash table

multi-collisions

How short?

OpenDNS cache: **27 bytes** on average

Ruby on Rails web application: **<20 bytes**

Why SipHash?

Minimizes hash flooding

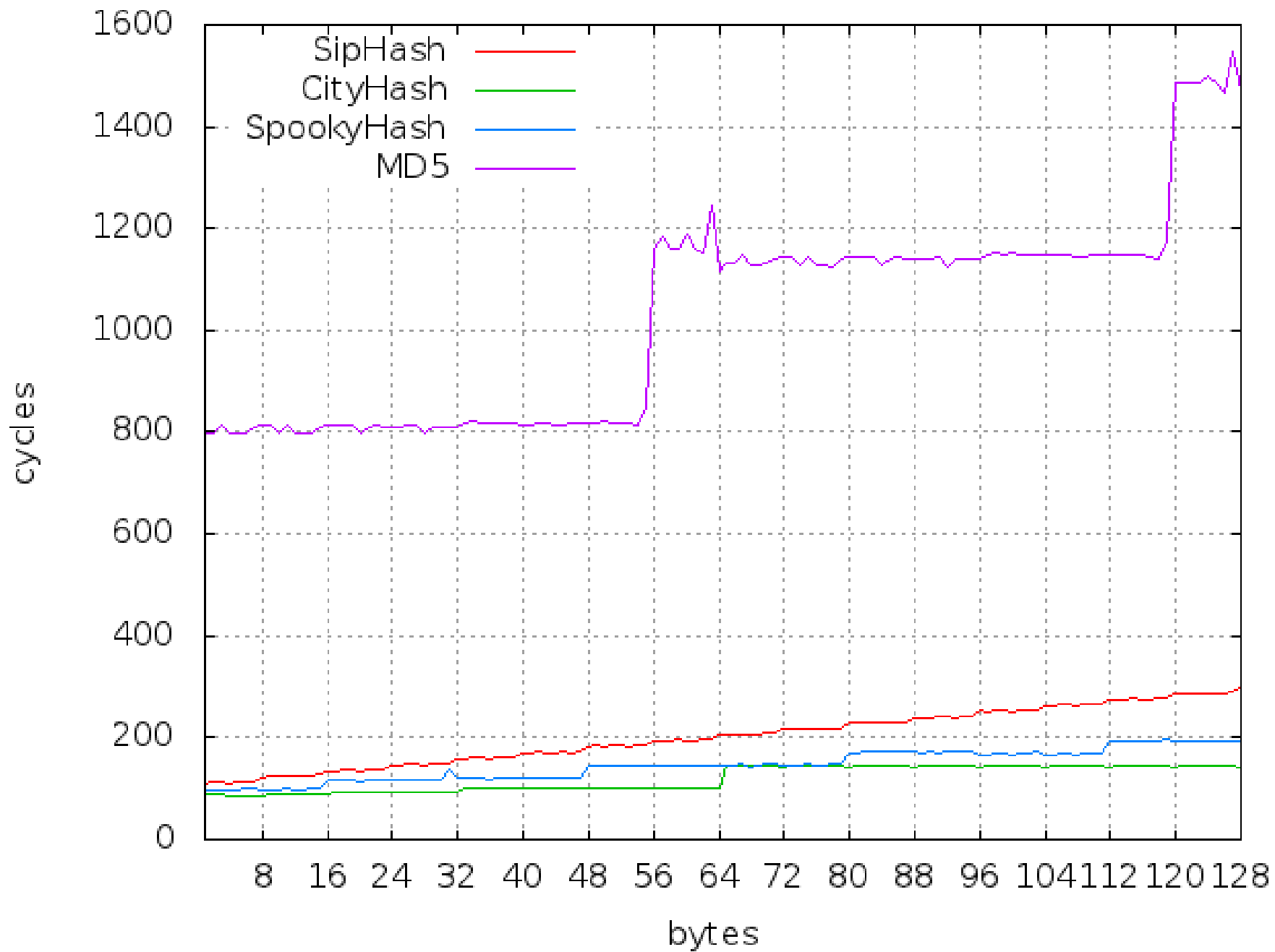→ impact limited to sqrt(communication)

Well-defined security goal (PRF)
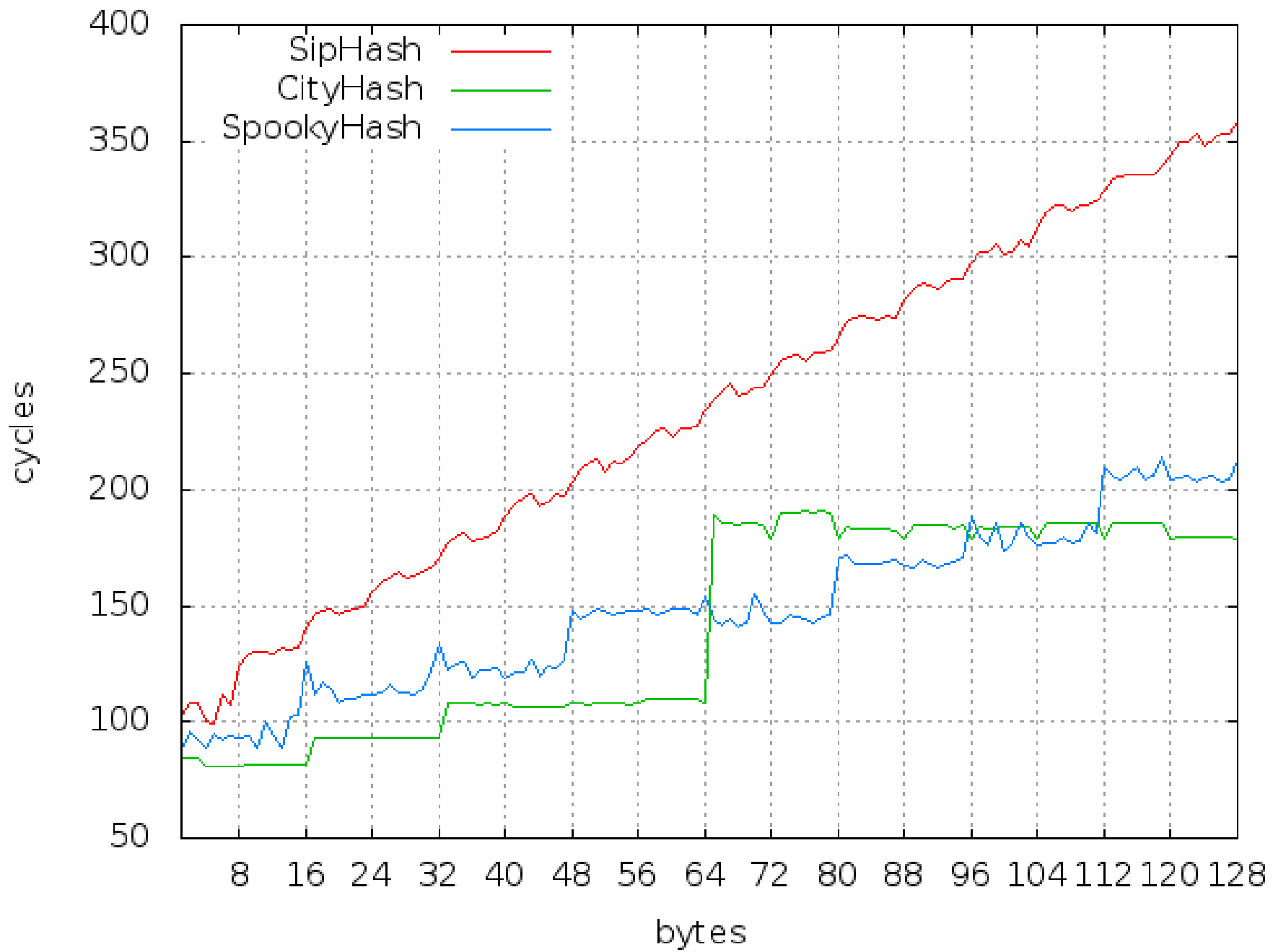
Competitive in speed with non-crypto hashes
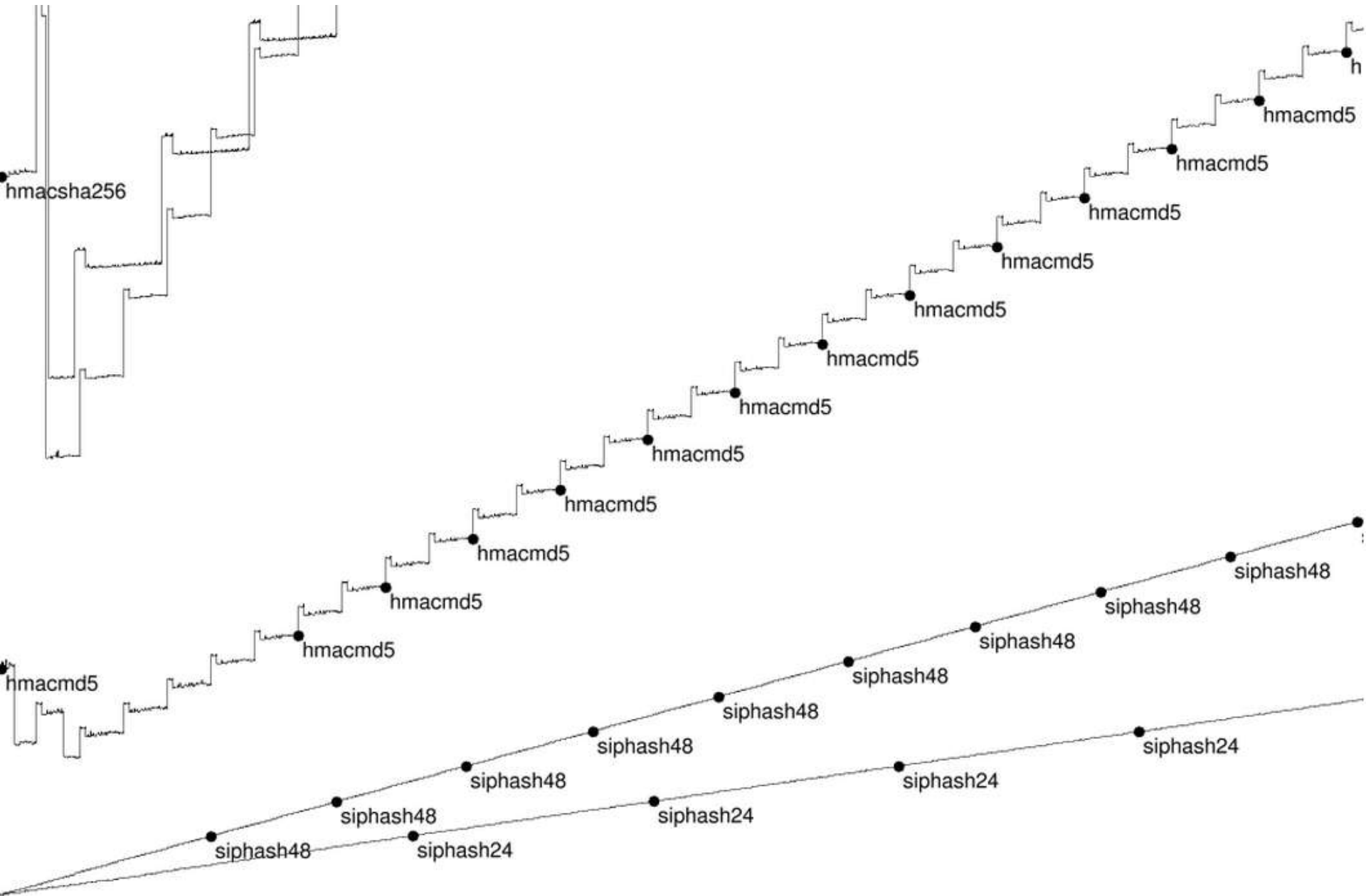
How fast?

SipHash-2-4 on an AMD Athlon II Neo

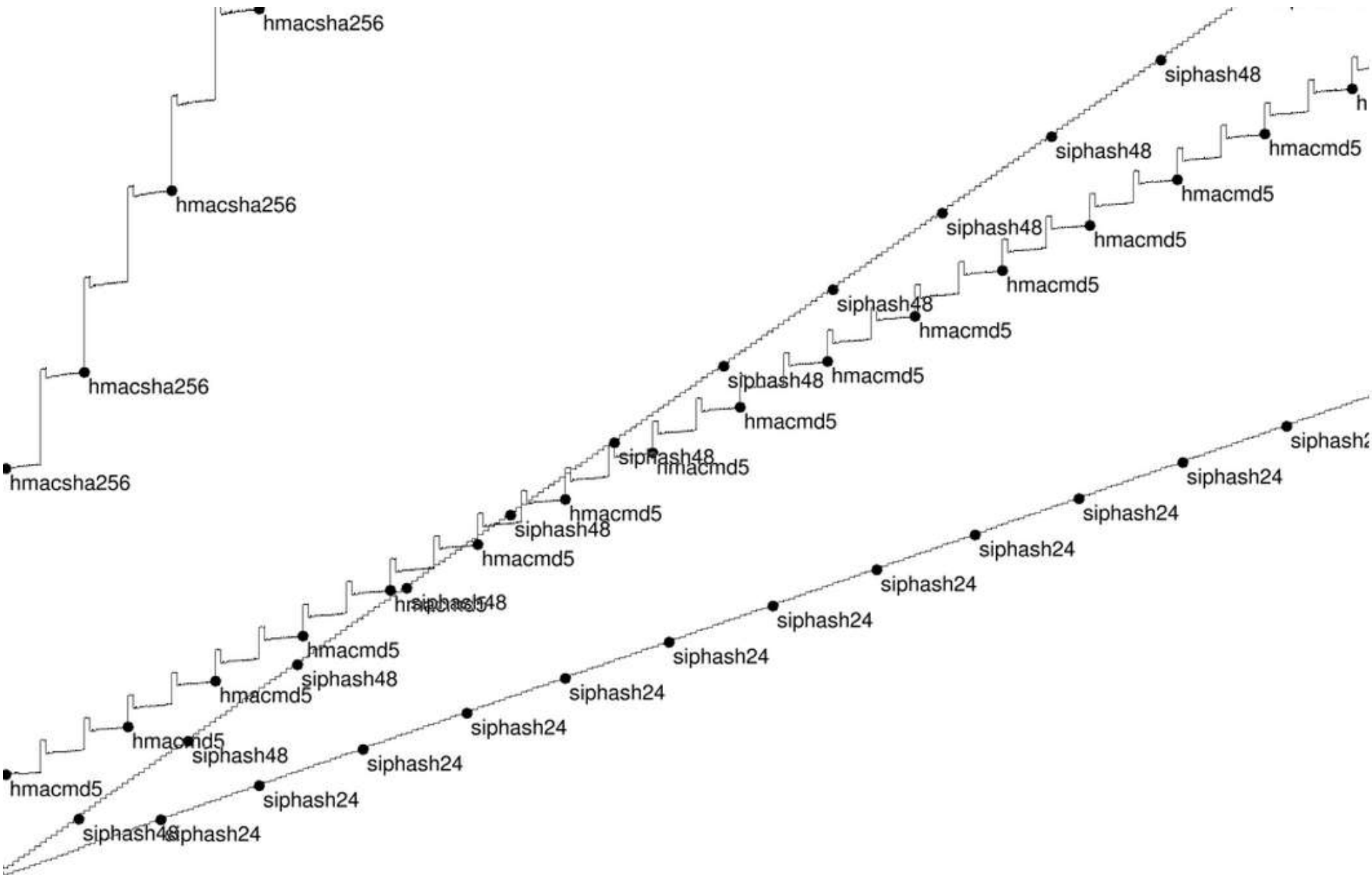| Byte length	8	16	32	64
Cycles (per byte)	123 (15.38)	134 (8.38)	158 (4.25)	204 (3.19)

Long data: **1.44** cycles/byte

Cryptanalysis

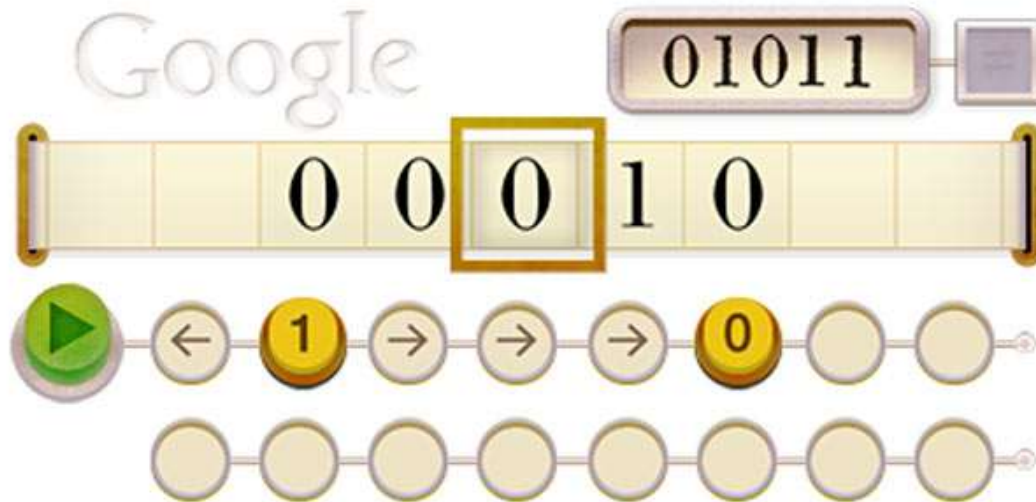# Generic attacks

$\approx 2^{128}$ key recovery

$\approx 2^{192}$ state recovery

$\approx 2^{128}$ internal-collision forgeries

$\approx 2^{s}$ forgery attack with success probability 2^{s-64}

Round	Differences				Prob.
1	`................`	`................`	`................`	`8...............`	1 (1)
	`................`	`................`	`8...............`	`8..........8...`	
2	`8..........8...`	`8...............`	`........8.......`	`8....1...1.8...`	13 (14)
	`....8...........`	`.............9...`	`8....1.8.1.8...`	`8.1.......1.....`	
3	`..1.8.....1.....`	`8.....11a.1.1...`	`8.1.1...8.....1.`	`8.1.82.......2..`	42 (56)
	`a...1...8.1.8.11`	`8.12b413a2......`	`....92..8....21.`	`82..92..82..82..`	
4	`22..82...21..211`	`e835621322.1.235`	`22...21.8.122613`	`621.c21.42..42.3`	103 (159)
	`2.11..24ca35e.13`	`66778453..57bd22`	`4.1.c...c212641.`	`82..82..8.11.6..`	
5	`a21182244a24e613`	`2ec144fcb8.115dd`	`c245d93226674453`	`e2.18..48a34a6.3`	152 (311)
	`f225f3ce8cd.c6d8`	`a44f51d8d.9e5616`	`2.445936ac53e25.`	`a.4.d3.2.a5...51`	
6	`52652.cc868.c689`	`27baa9d2d.e.fcd8`	`7ccdb44684.b.8ee`	`32246acc8cb4ce93`	187 (498)
	`566.3a5175df891e`	`2.e5d3.249fb3ea6`	`4ee9de8a.8bfc67d`	`2425523ec62cf459`	

Characteristic verified with ARXtools

http://www.di.ens.fr/~leurent/arxtools.html

Proof of insecurity

SipRound(0) = 0

That is, SipRound is not ideal

Therefore SipHash is insecure

Proof of simplicity

June 20: paper published online

June 28: **18** third-party implementations

C (Floodyberry, Boßlet, Neves); **C#** (Haynes)
Cryptol (Lazar); **Erlang**, **Javascript**, **PHP** (Denis)
Go (Chestnykh); **Haskell** (Hanquez);
Java, **Ruby** (Boßlet); **Lisp** (Brown);

More on SipHash:

http://131002.net/siphash

Thanks to all implementers!