HASH WORKSHOP 2006

August 25, 2006 UCSB

LASH

(+ comments on “provably secure” hash functions)

K. Bentahar, Bristol
D. Page, Bristol
J. H. Silverman, NTRU
M.-J. O. Saarinen, Royal Holloway
N. P. Smart, Bristol

LASH is a Hash Function

LASH-x computes a x-bit hash from an input bit sequence of arbitrary length. There are four concrete proposals:

Variant	**n**	**m**
LASH-160	640	40
LASH-256	1024	64
LASH-384	1536	96
LASH-512	2048	128

Where n is the size of the input to compression function in bits, and m is the size of the chaining variable in 8-bit bytes. We have for all versions $m = n/16$.

A Pseudorandom Sequence

Start with $y_0 = 54321$ and iterate

$$y_{i+1} = y_i^2 + 2 \pmod{2^{31} - 1}.$$

We define an additional sequence that results in reducing y_i to byte length:

$$a_i = y_i \ (\mathrm{mod}\ 2^8)$$

The first ten members of this sequence are

$$a_0 = 49, a_1 = 100, a_2 = 135, a_3 = 237, a_4 = 95,$$
$$a_5 = 26, a_6 = 139, a_7 = 214, a_8 = 163, a_9 = 194.$$

Matrix H

We take H to be the m-by-n circulant matrix associated to the sequence $a_0, \ldots, a_n$ generated by the "Pollard PRNG"

$$H = \begin{pmatrix} a_0 & a_{n-1} & a_{n-2} & \cdots & a_2 & a_1 \\ a_1 & a_0 & a_{n-1} & \cdots & a_3 & a_2 \\ a_2 & a_1 & a_n & \cdots & a_4 & a_3 \\ \vdots & & \ddots & & & \vdots \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_{m+1} & a_m \end{pmatrix}.$$

Because of the "circulant" nature of H, storage requirement in implementations is m bytes (rather than mn).

Compression Function

The compression function can be represented as

$$f(r, s) = (r \oplus s) + f_H(r \| s) \pmod{q},$$

where f_H is the linear function obtained from multiplying a matrix H, defined using the sequence $a_0, a_1, \ldots,$ by the column vector $(r\|s)^t$, considered as a bit vector.

Thus the compression function is based on a combination of addition modulo 256 and XORing.

This is a “wide variant” of the Miyaguchi-Preneel mode.

LASH Compression Function $\mathbf{t} = f(\mathbf{r}, \mathbf{s})$

for $i = 0, 1, \ldots, m-1$ **do**
  $t_i \leftarrow r_i \oplus s_i$
end for
for $i = 0, 1, \ldots, n$ **do**
  if $i < 8m$ **then**
    $x \leftarrow \lfloor 2^{-(7-(i \bmod 8))} r_{\lfloor i/8 \rfloor} \rfloor \mod 2$
  else
    $x \leftarrow \lfloor 2^{-(7-(i \bmod 8))} s_{(\lfloor i/8 \rfloor - m)} \rfloor \mod 2$
  end if
  if $x = 1$ **then**
    for $j = 0, 1, \ldots, m-1$ **do**
      $t_j \leftarrow t_j + a_{((n+j-i) \bmod n)} \mod 256$
    end for
  end if
end for

LASH

for $i = 0, 1, \ldots, m-1$ **do**
  $r_i = 0$ {Initialize chaining variable.}
end for
for $i = 0, 1, \ldots, \lceil l/8m \rceil - 1$ **do**
  for $j = 0, 1, \ldots, m-1$ **do**
    $s_i = v_{m \times i + j}$ {Get a message block, padded}
  end for
  $\mathbf{r} \leftarrow f(\mathbf{r}, \mathbf{s})$ {Run the compression function.}
end for
for $i = 0, 1, \ldots, m-1$ **do**
  $s_i \leftarrow \lfloor l/2^{8i} \rfloor \mod 256$ {Message length in little-endian format.}
end for
$\mathbf{r} \leftarrow f(\mathbf{r}, \mathbf{s})$ {Final iteration of the compression function.}
for $i = 0, 1, \ldots, m/2 - 1$ **do**
  $t_i = 16\lfloor r_{2i}/16 \rfloor + \lfloor r_{2i+1}/16 \rfloor$
end for {Return the $m/2$-byte hash result.}

That's it!

LASH is perhaps the only practical hash function that can be easily memorized, which helps with analysis and implementation.

Only XOR and bytewise addition is used and there is a high level of parallelism. Hence the implementations run fast on SIMD platforms, but can be implemented on any microcontroller (implementation size less than 100 bytes!).

Why LASH ?

- **L**inear **A**lgebra based **S**ecure **H**ash : As the main component is simply a matrix-vector product.

- **LA**ttice based **S**ecure **H**ash : Because inverting/finding collisions in the linear component of the hash function is closely related to the hard problem of finding short/close vectors in lattices.

- **L**ight-weight **A**rithmetical **S**ecure **H**ash : Because the design is very short and easy to remember.

- Royal Navy traditions ? (W. Churchill)

Speed comparison, 160 bits

Name	Implementation	Storage	Cycles/byte
SHA1-160	without SIMD	0 bytes	26.29
SHA1-160	with SIMD	64 bytes	16.86
LASH-160	without SIMD, store all matrix	25600 bytes	689.64
LASH-160	without SIMD, store one row	640 bytes	774.42
LASH-160	with SIMD, store all matrix	25600 bytes	392.83
LASH-160	with SIMD, store one row	640 bytes	523.26

Speed comparison, 256 bits

Name	Implementation	Storage	Cycles/byte
SHA2-256	without SIMD	256 bytes	55.16
SHA2-256	without SIMD	288 bytes	31.34
SHA2-256	with SIMD	256 bytes	45.20
LASH-256	without SIMD, store all matrix	65536 bytes	859.83
LASH-256	without SIMD, store one row	1024 bytes	1027.74
LASH-256	with SIMD, store all matrix	65536 bytes	344.81
LASH-256	with SIMD, store one row	1024 bytes	597.01

Speed comparison, 384 bits

Name	Implementation	Storage	Cycles/byte
SHA2-384	without SIMD	640 bytes	124.57
SHA2-384	without SIMD	704 bytes	117.45
LASH-384	without SIMD, store all matrix	147456 bytes	1078.58
LASH-384	without SIMD, store one row	1536 bytes	1355.09
LASH-384	with SIMD, store all matrix	147456 bytes	805.47
LASH-384	with SIMD, store one row	1536 bytes	1090.41

Speed comparison, 512 bits

Name	Implementation	Storage	Cycles/byte
SHA2-512	without SIMD	640 bytes	124.98
SHA2-512	without SIMD	704 bytes	117.52
LASH-512	without SIMD, store all matrix	262144 bytes	1351.39
LASH-512	without SIMD, store one row	2048 bytes	1730.14
LASH-512	with SIMD, store all matrix	262144 bytes	1036.70
LASH-512	with SIMD, store one row	2048 bytes	1220.54

Security issues

- The underlying problem is clearly a variant of subset sum / knapsack / short vector problem. A proof is given which relates collision resistance to a lattice - type problem.

- Current security parameter selection is based on careful analysis of standard cryptanalytic attacks, including generalized birthday attack. Prior versions have been broken.

- Internal state (chaining variable) is *twice* the size of the hash output, therefore making the hash resistant to multicollision attacks.

It’s simple!

The structure (parameter selection) is very flexible, reduced versions can be studied in a straightforward way.

We conjecture that security of the presented versions can be extrapolated from the security of reduced versions.

We also note that LASH is not secure without the final round and truncation of the final result.

Security Proof of LASH

□

(I don't understand it but I **think** it has something to do with a lettuce.)

LASH vs VSH

"VSH is not a hash function"

– Arjen Lenstra, Eurocrypt 2006

VSH and LASH have similar speed, and both can be described easily.

Collision resistance of LASH and VSH can be reduced to a plausible **security guess** (related to factoring in case of VSH).

VSH has weak preimage resistance. See my paper "Security of VSH in the Real World," `eprint.iacr.org/2006/103.pdf`

VSH hashes are very long (RSA modulus). Even if it is difficult to find 1024-bit collisions, that does not mean that finding collisions in 1023 bits is difficult.

LASH vs FSB

D. Augot, M. Finiasz, N. Sendrier, “A Family of Fast Syndrome Based Hash Functions”, Proc. MyCrypt 2005.

FSB is based on a very similar problem than LASH, but the security proof uses reduction to an NP-complete problem in Coding Theory.

There’s a 2^{30} attack in an upcoming paper of mine, based on simple linear algebra manipulation. **Worst case** complexity of the underlying “hard problem” is of course almost irrelevant to the security of the hash function..

“Provable Security” in Hash Functions

LASH, FSB, VSH, and the FFT hash (presented in this workshop) reduce collision resistance to a “hard problem”. In each one of these cases the exact “hard problem” was not well defined before the publication of the paper..

If (say) LLL can be used to break something, it does not mean that LLL is the *best* way of breaking something!

Collision resistance alone does not imply any other important properties of a general-purpose hash function.

FIN.

Have fun breaking LASH!