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:

<table>
<thead>
<tr>
<th>Variant</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>LASH-160</td>
<td>640</td>
<td>40</td>
</tr>
<tr>
<td>LASH-256</td>
<td>1024</td>
<td>64</td>
</tr>
<tr>
<td>LASH-384</td>
<td>1536</td>
<td>96</td>
</tr>
<tr>
<td>LASH-512</td>
<td>2048</td>
<td>128</td>
</tr>
</tbody>
</table>

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:

$$a_i = y_i \pmod{2^8}.$$  

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”

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

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 \( t = f(r, s) \)

\[
\text{for } i = 0, 1, \ldots, m - 1 \text{ do} \\
\quad t_i \leftarrow r_i \oplus s_i \\
\text{end for} \\
\text{for } i = 0, 1, \ldots, n \text{ do} \\
\quad \text{if } i < 8m \text{ then} \\
\quad \quad x \leftarrow \lfloor 2^{-7 \cdot (i \mod 8)} \cdot r_{[i/8]} \rfloor \mod 2 \\
\quad \text{else} \\
\quad \quad x \leftarrow \lfloor 2^{-7 \cdot (i \mod 8)} \cdot s_{([i/8]-m)} \rfloor \mod 2 \\
\quad \text{end if} \\
\quad \text{if } x = 1 \text{ then} \\
\quad \quad \text{for } j = 0, 1, \ldots, m - 1 \text{ do} \\
\quad \quad \quad t_j \leftarrow t_j + a((n+j-i) \mod n) \mod 256 \\
\quad \quad \text{end for} \\
\quad \text{end if} \\
\text{end for}
\]
LASH

for \( i = 0, 1, \ldots, m - 1 \) do
    \( r_i = 0 \) \hspace{1cm} \{Initialize chaining variable.\}
end for

for \( i = 0, 1, \ldots, \lfloor l/8m \rfloor - 1 \) do
    for \( j = 0, 1, \ldots, m - 1 \) do
        \( s_i = v_{m \times i + j} \) \hspace{1cm} \{Get a message block, padded\}
    end for
    \( r \leftarrow f(r, s) \) \hspace{1cm} \{Run the compression function.\}
end for

for \( i = 0, 1, \ldots, m - 1 \) do
    \( s_i \leftarrow \lfloor l/2^8i \rfloor \mod 256 \) \hspace{1cm} \{Message length in little-endian format.\}
end for

\( r \leftarrow f(r, s) \) \hspace{1cm} \{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 \hspace{1cm} \{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 ?

- **Linear Algebra based Secure Hash**: As the main component is simply a matrix-vector product.

- **LAttice based Secure Hash**: 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.

- **Light-weight Arithmetical Secure Hash**: Because the design is very short and easy to remember.

- **Royal Navy traditions ?** (W. Churchill)
## Speed comparison, 160 bits

<table>
<thead>
<tr>
<th>Name</th>
<th>Implementation</th>
<th>Storage</th>
<th>Cycles/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1-160</td>
<td>without SIMD</td>
<td>0 bytes</td>
<td>26.29</td>
</tr>
<tr>
<td>SHA1-160</td>
<td>with SIMD</td>
<td>64 bytes</td>
<td>16.86</td>
</tr>
<tr>
<td>LASH-160</td>
<td>without SIMD, store all matrix</td>
<td>25600 bytes</td>
<td>689.64</td>
</tr>
<tr>
<td>LASH-160</td>
<td>without SIMD, store one row</td>
<td>640 bytes</td>
<td>774.42</td>
</tr>
<tr>
<td>LASH-160</td>
<td>with SIMD, store all matrix</td>
<td>25600 bytes</td>
<td>392.83</td>
</tr>
<tr>
<td>LASH-160</td>
<td>with SIMD, store one row</td>
<td>640 bytes</td>
<td>523.26</td>
</tr>
</tbody>
</table>
# Speed comparison, 256 bits

<table>
<thead>
<tr>
<th>Name</th>
<th>Implementation</th>
<th>Storage</th>
<th>Cycles/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA2-256</td>
<td>without SIMD</td>
<td>256 bytes</td>
<td>55.16</td>
</tr>
<tr>
<td>SHA2-256</td>
<td>without SIMD</td>
<td>288 bytes</td>
<td>31.34</td>
</tr>
<tr>
<td>SHA2-256</td>
<td>with SIMD</td>
<td>256 bytes</td>
<td>45.20</td>
</tr>
<tr>
<td>LASH-256</td>
<td>without SIMD, store all matrix</td>
<td>65536 bytes</td>
<td>859.83</td>
</tr>
<tr>
<td>LASH-256</td>
<td>without SIMD, store one row</td>
<td>1024 bytes</td>
<td>1027.74</td>
</tr>
<tr>
<td>LASH-256</td>
<td>with SIMD, store all matrix</td>
<td>65536 bytes</td>
<td>344.81</td>
</tr>
<tr>
<td>LASH-256</td>
<td>with SIMD, store one row</td>
<td>1024 bytes</td>
<td>597.01</td>
</tr>
</tbody>
</table>
# Speed comparison, 384 bits

<table>
<thead>
<tr>
<th>Name</th>
<th>Implementation</th>
<th>Storage</th>
<th>Cycles/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA2-384</td>
<td>without SIMD</td>
<td>640 bytes</td>
<td>124.57</td>
</tr>
<tr>
<td>SHA2-384</td>
<td>without SIMD</td>
<td>704 bytes</td>
<td>117.45</td>
</tr>
<tr>
<td>LASH-384</td>
<td>without SIMD, store all matrix</td>
<td>147456 bytes</td>
<td>1078.58</td>
</tr>
<tr>
<td>LASH-384</td>
<td>without SIMD, store one row</td>
<td>1536 bytes</td>
<td>1355.09</td>
</tr>
<tr>
<td>LASH-384</td>
<td>with SIMD, store all matrix</td>
<td>147456 bytes</td>
<td>805.47</td>
</tr>
<tr>
<td>LASH-384</td>
<td>with SIMD, store one row</td>
<td>1536 bytes</td>
<td>1090.41</td>
</tr>
</tbody>
</table>
# Speed comparison, 512 bits

<table>
<thead>
<tr>
<th>Name</th>
<th>Implementation</th>
<th>Storage</th>
<th>Cycles/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA2-512</td>
<td>without SIMD</td>
<td>640 bytes</td>
<td>124.98</td>
</tr>
<tr>
<td>SHA2-512</td>
<td>without SIMD</td>
<td>704 bytes</td>
<td>117.52</td>
</tr>
<tr>
<td>LASH-512</td>
<td>without SIMD, store all matrix</td>
<td>262144 bytes</td>
<td>1351.39</td>
</tr>
<tr>
<td>LASH-512</td>
<td>without SIMD, store one row</td>
<td>2048 bytes</td>
<td>1730.14</td>
</tr>
<tr>
<td>LASH-512</td>
<td>with SIMD, store all matrix</td>
<td>262144 bytes</td>
<td>1036.70</td>
</tr>
<tr>
<td>LASH-512</td>
<td>with SIMD, store one row</td>
<td>2048 bytes</td>
<td>1220.54</td>
</tr>
</tbody>
</table>
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**


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!