Cryptanalysis of LASH

Scott Contini[1], Krystian Matusiewicz[1], Josef Pieprzyk[1], Ron Steinfeld[1], Jian Guo[2], San Ling[2], and Huaxiong Wang[1,2]

[1] Advanced Computing – Algorithms and Cryptography, Department of Computing, Macquarie University
{scontini,kmatus,josef,rons,hwang}@ics.mq.edu.au

[2] Nanyang Technological University, School of Physical & Mathematical Sciences
{guojian,lingsan,hxwang}@ntu.edu.sg

Abstract. We show that the LASH-x hash function is vulnerable to attacks that trade time for memory, including collision attacks as fast as $2^{\frac{4}{11}x}$ and preimage attacks as fast as $2^{\frac{4}{7}x}$. Moreover, we describe heuristic lattice based collision attacks that use small memory but require very long messages. Based upon experiments, the lattice attacks are expected to find collisions much faster than $2^{x/2}$. All of these attacks exploit the designers' choice of an all zero IV.

We then consider whether LASH can be patched simply by changing the IV. In this case, we show that LASH is vulnerable to a $2^{\frac{7}{8}x}$ preimage attack. We also show that LASH is trivially not a PRF when any subset of input bytes is used as a secret key. None of our attacks depend upon the particular contents of the LASH matrix – we only assume that the distribution of elements is more or less uniform.

Additionally, we show a generalized birthday attack on the final compression of LASH which requires $O\left(x2^{\frac{x}{2(1+\frac{107}{105})}}\right) \approx O(x2^{x/4})$ time and memory. Our method extends the Wagner algorithm to truncated sums, as is done in the final transform in LASH.

1 Introduction

The LASH hash function [3] is based upon the provable design of Goldreich, Goldwasser, and Halevi (GGH) [7], but changed in an attempt to make it closer to practical. The changes are:

1. Different parameters for the m by n matrix and the size of its elements to make it more efficient in both software and hardware.
2. The addition of a final transform [8] and a Miyaguchi-Preneel structure [10] in attempt to make it resistant to faster than generic attacks.

The LASH authors note that if one simply takes GGH and embeds it in a Merkle-Damgård structure using parameters that they want to use, then there are faster than generic attacks. More precisely, if the hash output is x bits, then they roughly describe attacks which are of order $2^{x/4}$ if n is larger than approximately m^2, or $2^{(7/24)x}$ otherwise[3]. These attacks require an amount of memory of the same order as the computation time. The authors hope that adding the second changes above prevent faster than generic attacks. The resulting proposals are called LASH-x, for LASH with an x bit output.

Although related to GGH, LASH is *not* a provable design: one can readily see in their proposal that there is no security proof [3]. Both the changes of parameters from GGH and the addition of the Miyaguchi-Preneel and final transform prevent the GGH security proof from being applied.

Our Results. In this paper, we show:

– LASH-x is vulnerable to collision attacks which trade time for memory (Sect. 4). This breaks the LASH-x hash function in as little as $2^{(4/11)x}$ work (i.e. nearly a cube root attack). Using similar techniques, we can find preimages in $2^{(4/7)x}$ operations. These attacks exploit LASH's all zero IV, and thus can be avoided by a simple tweak to the algorithm.
– Again exploiting the all zero IV, we can find *very* long message collisions using lattice reduction techniques (Sect. 6). Experiments suggest that collisions can be found much faster than $2^{x/2}$ work, and additionally the memory requirements are low.
– Even if the IV is changed, the function is still vulnerable to a short message (1 block) preimage attack that runs in time/memory $O(2^{(7/8)x})$ – faster than exhaustive search (Sect. 5). Our attack works for *any* IV.
– LASH is not a PRF (Sect. 3.1) when keyed through any subset of the input bytes. Although the LASH authors, like other designers of heuristic hash functions, only claimed security goals of collision resistance and preimage resistance, such functions are typically used for many other purposes [6] such as HMAC [2] which requires the PRF property.
– LASH's final compression (including final transform) can be attacked in $O\left(x2^{\frac{x}{2(1+\frac{107}{105})}}\right) \approx O(x2^{x/4})$ time and memory. To do this, we adapt Wagner's generalized birthday attack [13] to the case of truncated

[3] The authors actually describe the attacks in terms of m and n. We choose to use x which is more descriptive.

sums (Sect. 6). As far as we are aware, this is the fastest known attack on the final LASH compression.

Before we begin, we would like to make a remark concerning the use of large memory. Traditionally in cryptanalysis, memory requirements have been mostly ignored in judging the effectiveness of an attack. However, recently some researchers have come to question whether this is fair [4, 5, 14]. To address this issue in the context of our results, we point out that the design of LASH is motivated by the assumption that GGH is insufficient due to attacks that use large memory and run faster than generic attacks [3]. We are simply showing that LASH is also vulnerable to such attacks so the authors did not achieve what motivated them to change GGH.

After doing this work, we have learnt that a collision attack on the LASH compression function was sketched at the Second NIST Hash Workshop [9]. The attack applies to a certain class of circulant matrices. However, after discussions with the authors [11], we determined that the four concrete proposals of x equal to 160, 256, 384, and 512 are not in this class (although certain other values of x are). Furthermore, the attack is on the compression function only, and does not seem to extend to the full hash function.

2 Description of LASH

2.1 Notation

Let us define $\mathsf{rep}(\cdot) : \mathbb{Z}_{256} \to \mathbb{Z}_{256}^8$ as a function that takes a byte and returns a sequence of elements $0, 1 \in \mathbb{Z}_{256}$ corresponding to its binary representation in the order of most significant bit first. For example, $\mathsf{rep}(128) = (1, 0, 0, 0, 0, 0, 0, 0)$. We can generalize this notion to sequences of bytes. The function $\mathsf{Rep}(\cdot) : \mathbb{Z}_{256}^m \to \mathbb{Z}_{256}^{8 \cdot m}$ is defined as $\mathsf{Rep}(s) = \mathsf{rep}(s_1) || \ldots || \mathsf{rep}(s_m)$, e.g. $\mathsf{Rep}((192, 128)) = (1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$. Moreover, for two sequences of bytes we define $\oplus$ as the usual bitwise XOR of the two bitstrings.

We index elements of vectors and matrices starting from zero.

2.2 The LASH-x Hash Function

The LASH-x hash function maps an input of length less than 2^{2x} bits to an output of x bits. Four concrete proposals were suggested in [3]: $x =$ 160, 256, 384, and 512.

The hash is computed by iterating a compression function that maps blocks of $n = 4x$ bits to $m = x/4$ bytes ($2x$ bits). The measure of n in bits and m in bytes is due to the original paper. Always $m = n/16$. Below we describe the compression function, and then the full hash function.

Compression Function of LASH-*x*. The compression function is of the form $f : \mathbb{Z}_{256}^{2m} \to \mathbb{Z}_{256}^{m}$. It is defined as

$$f(r, s) = (r \oplus s) + H \cdot [\mathsf{Rep}(r)||\mathsf{Rep}(s)]^T \ , \tag{1}$$

where $r = (r_0, \ldots, r_{m-1})$ and $s = (s_0, \ldots, s_{m-1})$ belong to $\mathbb{Z}_{256}^{m}$. The vector r is called the *chaining variable.*

The matrix H is a circulant matrix of dimensions $m \times (16m)$ defined as

$$H_{j\,,k} = a_{\,(j-k) \bmod 16m} \ ,$$

where $a_i = y_i \pmod{2^8}$ is a reduction modulo 256 of elements of the sequence y_i based on the Pollard pseudorandom sequence

$$y_0 = 54321, \qquad\qquad y_{i+1} = y_i^2 + 2 \pmod{2^{31} - 1} \ .$$

Our attacks do not use the circulant matrix properties or any properties of this sequence.

A visual diagram of the LASH-160 compression function is given in Figure 1, where t is $f(r, s)$.

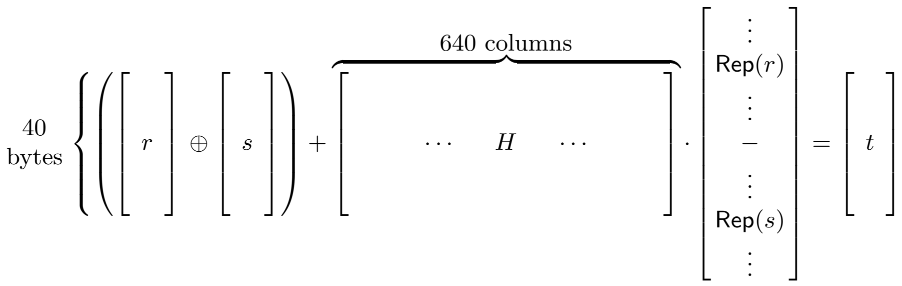


Fig. 1. Visualizing the LASH-160 compression function.

The Full Function. Given a message of l bits, padding is first applied by appending a single '1'-bit followed by enough zeros to make the length a multiple of $8m = 2x$. The padded message consists of $\kappa = \lceil (l+1)/8m \rceil$ blocks of m bytes. Then, an extra block b of m bytes is appended that contains the encoded bit-length of the original message, $b_i = \lfloor l/2^{8i} \rfloor \pmod{256}$, $i = 0, \ldots, m-1$.

Next, the blocks $s^{(0)}, s^{(1)}, \ldots, s^{(\kappa)}$ of the padded message are fed to the compression function in an iterative manner,

$$\begin{aligned} r^{(0)} &:= (0, \ldots, 0) \ , \\ r^{(j+1)} &:= f(r^{(j)}, s^{(j)}), \quad j = 0, \ldots, \kappa \ . \end{aligned}$$

The $r^{(0)}$ is call the IV. Finally, the last chaining value $r^{(\kappa+1)}$ is sent through a *final transform* which takes only the 4 most significant bits of each byte to form the final hash value h. Precisely, the i^{th} byte of h is $h_i = 16\lfloor r_{2i}/16 \rfloor + \lfloor r_{2i+1}/16 \rfloor$ $(0 \le i < m)$.

3 Initial Observations

3.1 LASH is Not a PRF

In some applications (e.g. HMAC) it is required that the compression function (parameterized by its IV) should be a PRF. Below we show that LASH does not satisfy this property.

Assume that r is the secret parameter fixed beforehand and unknown to us. We are presented with a function $g(\cdot)$ which may be $f(r, \cdot)$ or a random function and by querying it we have to decide which one we have.

First of all, note that we can split our matrix H into two parts $H = [H_L || H_R]$ and so (1) can be rewritten as

$$f(r, s) = (r \oplus s) + H_L \cdot \mathsf{Rep}(r)^T + H_R \cdot \mathsf{Rep}(s)^T \ .$$

Sending in $s = 0$, we get

$$f(r, 0) = r + H_L \cdot \mathsf{Rep}(r)^T \ . \tag{2}$$

Now, for $s' = (128, 0, \ldots, 0)$ we have

$$\mathsf{Rep}(s') = 10000000\ 00000000\ \ldots\ 0000000$$

and so

$$f(r, s') = (r_0 \oplus 128, r_1, \ldots, r_{m-1}) + H_L \cdot \mathsf{Rep}(r)^T + H_R[\cdot, 0] \ . \tag{3}$$

where $H_R[\cdot, 0]$ denotes the first column of the matrix H_R. Let us compute the difference between (2) and (3):

$$\begin{aligned} f(r, s') - f(r, 0) &= (r_0 \oplus 128, r_1, \ldots, r_{m-1})^T + H_L \cdot \mathsf{Rep}(r)^T + \\ &\quad H_R[\cdot, 0] - r - H_L \cdot \mathsf{Rep}(r)^T \\ &= H_R[\cdot, 0] + ((r_0 \oplus 128) - r_0, 0, 0, \ldots, 0)^T \\ &= H_R[\cdot, 0] + (128, 0, \ldots, 0)^T. \end{aligned}$$

Regardless of the value of the secret parameter r, the output difference is a fixed vector equal to $H_R[\cdot, 0] + (128, 0, \ldots, 0)^T$. Thus, using only two queries we can distinguish with probability $1 - 2^{-8m}$ the LASH compression function with secret IV from a randomly chosen function.

The same principle can be used to distinguish LASH even if most of the bytes of s are secret as well. In fact, it is enough for us to control only one byte of the input to be able to use this method and distinguish with probability $1 - 2^{-8}$.

3.2 Absorbing the Feed-Forward Mode

According to [3], the feed-forward operation is motivated by Miyaguchi-Preneel hashing mode and is introduced to thwart some possible attacks on the plain matrix-multiplication construction. In this section we show two conditions under which the feed-forward operation can be described in terms of matrix operations and consequently absorbed into the LASH matrix multiplication step to get a simplified description of the compression function. The first condition requires one of the compression function inputs to be *known*, and the second requires a special subset of input messages.

First Condition: Partially Known Input. Suppose the r portion of the (r, s) input pair to the compression function is known and we wish to express the output $g(s) \overset{\text{def}}{=} f(r, s)$ in terms of the unknown input s. We observe that each $(8i + j)$th bit of the feedforward term $r \oplus s$ (for $i = 0, \ldots, m - 1$ and $j = 0, \ldots, 7$) can be written as

$$\mathsf{Rep}(r \oplus s)_{8i+j} = \mathsf{Rep}(r)_{8i+j} + (-1)^{\mathsf{Rep}(r)_{8i+j}} \cdot \mathsf{Rep}(s)_{8i+j}.$$

Hence the value of the ith byte of $r \oplus s$ is given by

$$\sum_{j=0}^{7} \left(\mathsf{Rep}(r)_{8i+j} + (-1)^{\mathsf{Rep}(r)_{8i+j}} \cdot \mathsf{Rep}(s)_{8i+j} \right) \cdot 2^{7-j} =$$

$$\left(\sum_{j=0}^{7} \mathsf{Rep}(r)_{8i+j} \cdot 2^{7-j} \right) + \left(\sum_{j=0}^{7} (-1)^{\mathsf{Rep}(r)_{8i+j}} \cdot \mathsf{Rep}(s)_{8i+j} \cdot 2^{7-j} \right) .$$

The first integer in parentheses after the equal sign is just the ith byte of r, whereas the second integer in parentheses is linear in the bits of s with known coefficients, and can be absorbed by appropriate additions to elements of the matrix H_R. Hence we have an 'affine' representation for $g(s)$:

$$g(s) = (D' + H_R) \cdot \mathsf{Rep}(s)^T + \underbrace{r + H_L \cdot \mathsf{Rep}(r)^T}_{m \times 1 \text{ vector}} , \qquad (4)$$

where H_R is the submatrix of H indexed by the bits of s (i.e. the last $8m$ columns of H), and

$$D' = \begin{bmatrix} J_0 & 0_8 & \dots & 0_8 & 0_8 \\ 0_8 & J_1 & \dots & 0_8 & 0_8 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0_8 & 0_8 & \dots & J_{m-2} & 0_8 \\ 0_8 & 0_8 & \dots & 0_8 & J_{m-1} \end{bmatrix} ,$$

where, for $i = 0, \dots, m-1$, we define the 1×8 vectors $0_8 = [0,0,0,0,0,0,0,0]$ and

$$J_i = [2^7 \cdot (-1)^{\mathsf{Rep}(r)_{8i}}, 2^6 \cdot (-1)^{\mathsf{Rep}(r)_{8i+1}}, \dots, 2^1 \cdot (-1)^{\mathsf{Rep}(r)_{8i+6}}, 2^0 \cdot (-1)^{\mathsf{Rep}(r)_{8i+7}}] .$$

Second Condition: Special Input Subset. In addition to the above we also observe that when bytes of one of the input sequences (say, r) are restricted to values $\{0, 128\}$ only (i.e. only the most significant bit in each byte can be set), the XOR operation behaves like the byte-wise addition modulo 256. In other words, if $r^* = 128 \cdot r'$ where $r' \in \{0,1\}^m$ then

$$\begin{aligned} f(r^*, s) &= r^* + s + H \cdot [\mathsf{Rep}(r^*) || \mathsf{Rep}(s)]^T \\ &= (D_J + H) \cdot [\mathsf{Rep}(r^*) || \mathsf{Rep}(s)]^T . \end{aligned} \qquad (5)$$

The matrix D_J recreates values of r^* and s from their representations and is the following block matrix of dimensions $m \times (16m)$,

$$\begin{bmatrix} J & 0_8 & 0_8 & \dots & 0_8 & 0_8 & J & 0_8 & 0_8 & \dots & 0_8 & 0_8 \\ 0_8 & J & 0_8 & \dots & 0_8 & 0_8 & 0_8 & J & 0_8 & \dots & 0_8 & 0_8 \\ 0_8 & 0_8 & J & \dots & 0_8 & 0_8 & 0_8 & 0_8 & J & \dots & 0_8 & 0_8 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0_8 & 0_8 & 0_8 & \dots & J & 0_8 & 0_8 & 0_8 & 0_8 & \dots & J & 0_8 \\ 0_8 & 0_8 & 0_8 & \dots & 0_8 & J & 0_8 & 0_8 & 0_8 & \dots & 0_8 & J \end{bmatrix},$$

where $J = [2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0]$ and $0_8 = [0, 0, 0, 0, 0, 0, 0, 0]$.

Since all the bits apart from the most significant one are always set to zero in r^* we can safely remove the corresponding columns of the matrix $D_J + H$ (i.e. columns with indices $8i + 1, \dots, 8i + 7$ for $i = 0, \dots, 39$). Let us denote the resulting matrix by H'. Then the whole compression function can be represented as

$$f(r', s) = H' \cdot [r' || \mathsf{Rep}(s)]^T$$

that compresses $m + 8m$ bits to $8m$ bits using only matrix multiplication without any feed-forward mode.

4 Attacks Exploiting Zero IV

Collision Attack. In the original LASH paper, the authors describe a "hybrid attack" against LASH without the appended message length and final transform. Their idea is to do a Pollard or parallel collision search in such a way that each iteration forces some output bits to a fixed value (such as zero). Thus, the number of possible outputs is reduced from the standard attack. If the total number of possible outputs is S, then a collision is expected after about $\sqrt{S}$ iterations. Using a combination of table lookup and linear algebra, they are able to achieve $S = 2^{\frac{14}{3}m}$ in their paper. Thus, the attack is not effective since a collision is expected in about $2^{\frac{7}{3}m} = 2^{\frac{7}{12}x}$ iterations, which is more than the $2^{x/2}$ iterations one gets from the standard birthday attack on the full LASH function (with the final output transform).

Here, exploiting the zero IV, we describe a similar but simpler attack on the full function which uses table lookup only. Our messages will consist of a number of all-zero blocks followed by one "random" block. Regardless of the number of zero blocks at the beginning, the output of the compression function immediately prior to the length block being processed is determined entirely by the one "random" block. Thus, we will be using table lookup to determine a message length that results in a hash output value which has several bits in certain locations set to some predetermined value(s).

Refer to the visual diagram of the LASH-160 compression function in Fig. 1. Consider the case of the last compression, where the value of r is the output from the previous iteration and the value of s is the message length being fed in. The resulting hash value will consist of the most-significant half-bytes of the bytes of t. Our goal is to quickly determine a value of s so that the most significant half-bytes from the bottom part of t are all approximately zero.

Our messages will be long but not extremely long. Let α be the maximum number of bytes necessary to represent (in binary) any s that we will use. So the bottom $40 - \alpha$ bytes of s are all 0 bytes, and the bottom $320 - 8\alpha$ bits of $\mathsf{Rep}(s)$ are all 0 bits. As before, we divide the matrix H into two halves, H_L and H_R. Without specifying the entire s, we can compute the bottom $40-\alpha$ bytes of $(r \oplus s) + H_L \cdot \mathsf{Rep}(r)$. Thus, if we pre-computed all possibilities for $H_R \cdot \mathsf{Rep}(s)$, then we can use table lookup to determine a value of s that hopefully causes h (to be chosen later) most-significant half-bytes from the bottom part of t to be 0. See the diagram in Fig. 2. The only restriction in doing this is $\alpha + h \leq 40$.

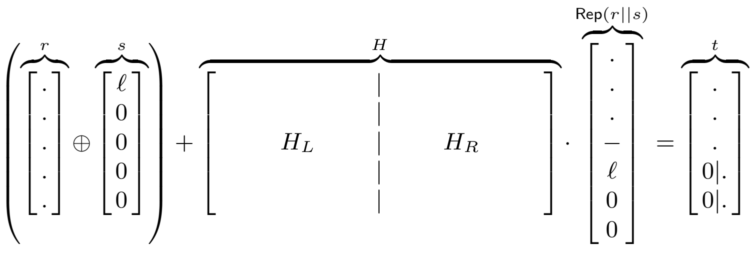


Fig. 2. Visualizing the final block of the attack on the LASH-160 compression function. Diagram is not to scale. Table lookup is done to determine the values at the positions marked with ℓ. Places marked with 0 are set to be zero by the attacker (in the t vector, this is accomplished with the table lookup). Places marked with '.' are outside of the attacker's control.

We additionally require dealing with the padding byte. To do so, we restrict our messages to lengths congruent to 312 mod 320. Then our "random" block can have anything for the first 39 bytes followed by `0x80` for the 40th byte which is the padding. We then assure that only those lengths occur in our table lookup by only precomputing $H_R \cdot \mathsf{Rep}(s)$ for values of s of the form $320i + 312$. Thus, we have $\alpha = \lceil \frac{\log 320 + c}{8} \rceil$ assuming we take all values of i less than 2^c. We will aim for $h = c/4$, i.e. setting the bottom $c/4$ half-bytes of t equal to zero. The condition $\alpha + h \leq 40$ is then satisfied as long as $c \leq 104$, which will not be a problem.

Complexity. Pseudocode for the precomputation and table lookup are given in Table 1. With probability $1 - \frac{1}{e} \approx 0.632$, we expect to find a match in our table lookup. Assume that is the case. Due to rounding error, each of the bottom $c/4$ most significant half-bytes of t will either be 0 or -1 (`0xf` in hexadecimal). Thus there are $2^{c/4}$ possibilities for the bottom $c/4$ half-bytes, and the remaining $m - c/4 = x/4 - c/4$ half-bytes ($x - c$ bits) can be anything. So the size of the output space is $S = 2^{x-c+c/4} = 2^{x-3c/4}$. We expect a collision after we have about $2^{x/2-3c/8}$ outputs of this form. Note that with a Pollard or parallel collision search, we will not have outputs of this form a fraction of about $1/e$ of the time. This only means that we have to apply our iteration a fraction of $1/(1 - \frac{1}{e}) \approx 1.582$ times longer, which has negligible impact on the effectiveness of the attack. Therefore, we ignore such constants. Balancing the Pollard search time with the precomputation time, we get an optimal value with $c = (4/11)x$, i.e. a running time of order $2^{(4/11)x}$ LASH-x operations. The lengths of our colliding messages will be order $\leq 2^{c+\log 2x}$ bits.

For instance, in LASH-160 the optimal value is $c = 58$, yielding a precomputation time of about 2^{58}, a Pollard rho time of about 2^{58}, storage of about 2^{58}, and colliding messages of lengths about 2^{63} bytes. A more realistic number to choose in practice is $c = 40$, which gives precomputation time of 2^{40}, Pollard rho time of 2^{65}, storage of 2^{40}, and colliding messages of 2^{45} bytes.

Experimental Results. We used this method to find collisions in a truncated version of LASH-160. Table 3 lists the nonzero blocks of two long messages that collide on the last 12 bytes of the hash. Note that padding byte needs to be added on to the end of the messages. We used $c = 28$ and two weeks of cpu time on a 2.4GHz PC to find these.

Preimage Attack. The same lookup technique can be used for preimage attacks. One simply chooses random inputs and hashes them such that the looked up length sets some of the output hash bits to the target. This involves 2^c precomputation, 2^c storage, and $2^{x-3c/4}$ expected computation time, which balances to time/memory $2^{(4/7)x}$ using the optimal parameter setting $c = (4/7)x$.

5 Short Message Preimage Attack on LASH with Arbitrary IV

The attacks in the previous section crucially exploit a particular parameter choice made by the LASH designers, namely the use of an all zero Initial Value (IV) in the Merkle-Damgård construction. Hence, it is tempting to try to 'repair' the LASH design by using a non-zero (or even random) value for the IV. In this section, we show that for any choice of IV, LASH-x is vulnerable to a preimage attack faster than the desired security level of $O(2^x)$. Our preimage attack takes time/memory $O(2^{\frac{7}{8}x})$, and produces preimages of short length ($2x$ bits).

The Attack. Let $f : \mathbb{Z}_{256}^{2m} \to \mathbb{Z}_{256}^{m}$ denote the internal LASH compression function and $f_{out} : \mathbb{Z}_{256}^{2m} \to \mathbb{Z}_{16}^{m}$ denote the final compression function, i.e. the composition of f with the final transform applied to the output of f. Given a target value t_{out} whose LASH preimage is desired, the inversion algorithm finds a single block message $s_{in} \in \mathbb{Z}_{256}^{m}$ hashing

Table 1. The two main procedures for the long message attack on LASH-160. Only the bottom $c/4$ bytes of t need to be computed in Lookup(). Similarly, only the bottom $c/4$ bytes of v need to be computed in Precomp().

```
Precomp( int $c$ )
{
    for $i := 0$ to $2^c - 1$ do
        Compute $v := H_R \cdot \mathsf{Rep}(320i + 312)$.
        Round off bottom $c/4$ most significant half-bytes of $v$.
        Store rounded half-bytes and $320i + 312$ in a file.
}

Lookup( uchar $r[40]$, uchar $s[40]$, int c )
{
    Expand $r$ to a 320-bit vector, $v$.
    Compute $t :=$ (uchar *)$(-r - H_L \cdot v)$.
    Round off bottom $c/4$ most significant half-bytes of $t$.
    Look for a match of these half-bytes in a file.
    if match exists then
        Read in corresponding length.
        Encode length into $s$ vector.
    else
        Choose the "closest" data entry from file.
        Read in corresponding length.
        Encode length into $s$ vector.
}
```

First Message	**Second Message**
$l = 3380367992$	$l = 1380208632$
first nonzero block:	first nonzero block:
fc 66 f8 79 ef 7e 97 9c e0 ff	3f 8a b2 44 3f b3 3d 9d e0 ff
ff 0f 00 00 00 00 00 00 00 00	00 02 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00
hash:	hash:
a4 6a df fc 34 27 c4 99 c1 85	71 07 4f 54 7f f1 bd 5c c1 85
7a d8 07 51 97 84 f0 0f 00 ff	7a d8 07 51 97 84 f0 0f 00 ff

Fig. 3. Two long messages that match on the last 12 bytes of the hash.

to t_{out}, i.e. satisfying

$$f_{out}(r_{out}, s_{out}) = t_{out} \text{ and } f(r_{in}, s_{in}) = r_{out},$$

where s_{out} is equal the $8m$-bit binary representation of the integer $8m$ (the bit length of a single message block), and $r_{in} = IV$ is an arbitrary known value. The inversion algorithm proceeds as follows (see Fig. 4):

Step 1: Using the precomputation-based preimage attack on the final compression function f_{out} described in the previous section (with straightforward modifications to produce the preimage using bits of r_{out} rather than s_{out} and precomputation parameter $c_{out} = (20/7)m$), compute a list L of 2^m preimage values of r_{out} satisfying $f_{out}(r_{out}, s_{out}) = t_{out}$.

Step 2: Let $c = 3.5m$ be a parameter (later we show that choosing $c = 3.5m$ is optimal). Split the $8m$-bit input s_{in} to be determined into two disjoint parts $s_{in}(1)$ (of length $6m - c$ bit) and $s_{in}(2)$ (of length $2m + c$ bit), i.e. $s_{in} = s_{in}(1)||s_{in}(2)$. For each of the 2^m values of r_{out} from the list L produced by the first step above, and each of the 2^{6m-c} possible values for $s_{in}(1)$, run the internal compression function 'hybrid' partial inversion algorithm described below to compute a matching 'partial preimage' value for $s_{in}(2)$, where by 'partial preimage' we mean that the compression function output $f(r_{in}, s_{in})$ matches target r_{out} on a fixed set of $m + c = 4.5m$ bits (out of the $8m$ bits of r_{out}). For each such computed partial preimage $s_{in} = s_{in}(1)||s_{in}(2)$ and corresponding r_{out} value, check whether s_{in} is a *full* preimage, i.e. whether $f(r_{in}, s_{in}) = r_{out}$ holds, and if so, output desired preimage s_{in}.

For integer parameter c, the internal compression function 'hybrid' partial inversion algorithm is given a $8m$-bit target value t_{in}, an $8m$-bit

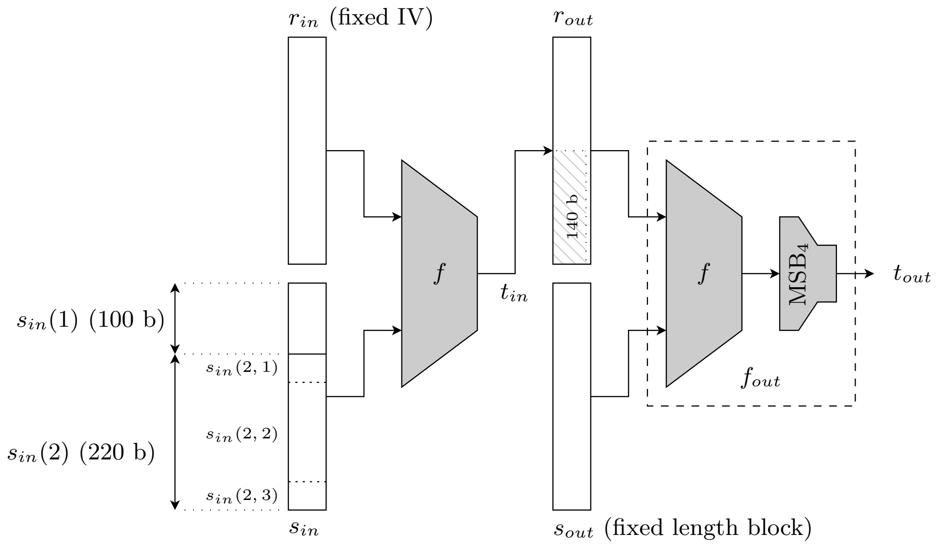


Fig. 4. Illustration of the preimage attack applied to LASH-160.

input r_{in}, and the $(6m-c)$-bit value $s_{in}(1)$, and computes a $(2m+c)$-bit value for $s_{in}(2)$ such that $f(r_{in}, s_{in})$ matches t_{in} on the top $c/7$ bytes as well as on the LS bit of all remaining bytes (a total of $m+c$ matching bits). The algorithm works as follows:

Feedforward Absorption: We use the observation from Section 3.2 that for known r_{in}, the Miyaguchi-Preneel feedforward term $(r_{in} \oplus s_{in})$ can be absorbed into the matrix by appropriate modifications to the matrix and target vector, i.e. the inversion equation

$$(r_{in} \oplus s_{in}) + H \cdot [\mathsf{Rep}(r_{in})||\mathsf{Rep}(s_{in})]^T = t_{in} \bmod 256, \tag{6}$$

where H is the LASH matrix, can be reduced to an equivalent linear equation

$$H' \cdot [\mathsf{Rep}(s_{in})]^T = t'_{in} \bmod 256, \tag{7}$$

for appropriate matrix H' and vector t' easily computed from the known H, t, and r_{in}.

Search for Collisions: To find $s_{in}(2)$ such that the left and right hand sides of (7) match on the desired $m+c$ bits, we use the hybrid method based on [3], which works as follows:

– Initialization: Split $s_{in}(2)$ into 3 parts $s(2,1)$ (length m bits), $s(2,2)$ (length c bits) and $s(2,3)$ (length m bits). For $i = 1,2,3$ let $H'(2,i)$ denote the submatrix of matrix H' from (7) consisting of the columns indexed by the bits in $s(2,i)$ (e.g. $H'(2,1)$ consists of the m columns of H' indexed by the m bits of $s(2,1)$). Similarly, let $H'(1)$ denote the submatrix of H' consisting of the columns of H' indexed by the m bits of $s_{in}(1)$.

- Target Independent Precomputation: For each of 2^c possible values of $s(2,2)$, find by linear algebra over $GF(2)$, a matching value for $s(2,3)$ such that

$$[H'(2,2)\ H'(2,3)] \cdot [\mathsf{Rep}(s(2,2))||\mathsf{Rep}(s(2,3))]^T = [0^m]^T \bmod 2, \tag{8}$$

  i.e. vector $y = [H'(2,2)\ H'(2,3)] \cdot [\mathsf{Rep}(s(2,2))||\mathsf{Rep}(s(2,3))]^T \bmod 256$ has zeros on the LS bits of all m bytes. Store entry $s(2,2)||s(2,3)$ in a hash table, indexed by the string of c bits obtained by concatanating 7 MS bits of each of the top $c/7$ bytes of vector y.
- Solving Linear Equations: Compute $s(2,1)$ such that

$$H'(2,1) \cdot [\mathsf{Rep}(s(2,1))]^T = t'_{in} - H'(1) \cdot [\mathsf{Rep}(s_{in}(1))]^T \bmod 2. \tag{9}$$

  Note that adding (8) and (9) implies that $H' \cdot [\mathsf{Rep}(s_{in}(1))||\mathsf{Rep}(s_{in}(2))]^T = t'_{in} \bmod 2$ with $s_{in}(2) = s(2,1)||s(2,2)||s(2,3)$ for any entry $s(2,2)||s(2,3)$ from the hash table.
- Lookup Hash Table: Find the $s(2,2)||s(2,3)$ entry indexed by the c-bit string obtained by concatanating the 7 MS bits of each of the top $c/7$ bytes of the vector $t'_{in} - H'(2,1) \cdot [\mathsf{Rep}(s(2,1))]^T - H'(1) \cdot [\mathsf{Rep}(s_{in}(1))]^T \bmod 256$. This implies that vector $H' \cdot [\mathsf{Rep}(s_{in}(1))||\mathsf{Rep}(s_{in}(2))]^T$ matches t'_{in} on all top $c/7$ bytes, as well as on the LS bits of all bytes, as required.

Correctness of Attack. For each of 2^m target values r_{out} from list L, and each of the $2^{2.5m}$ possible values for $s_{in}(1)$, the partial preimage inversion algorithm returns $s_{in}(2)$ such that $f(r_{in}, s_{in})$ matches r_{out} on a fixed set of $m + c$ bits. Heuristically modelling the remaining bits of $f(r_{in}, s_{in})$ as uniformly random and independent of r_{out}, we conclude that $f(r_{in}, s_{in})$ matches r_{out} on all $8m$ bits with probability $1/2^{8m-(m+c)} = 1/2^{7m-c} = 1/2^{3.5m}$ (using $c = 3.5m$) for each of the $2^{2.5m} \times 2^m = 2^{3.5m}$ runs of the partial inversion algorithm. Assuming (heuristically) that each of these runs are independent, the expected number of runs which produce a full preimage is $2^{3.5m} \times 1/2^{3.5m} = 1$, and hence we expect the algorithm to succeed and return a full preimage.

Complexity. The cost of the attack is dominated by the second step, where we balance the precomputation time/memory $O(2^c)$ of the hybrid partial preimage inversion algorithm with the expected number 2^{7m-c} of runs to get a full preimage. This leads (with the optimum parameter choice $c = 3.5m$) to time/memory cost $O(2^{3.5m}) = O(2^{\frac{7}{8}x})$, assuming each table lookup takes constant time. To see that second step dominates the cost, we recall that the first step with precomputation parameter c_{out} uses a precomputation taking time/memory $O(2^{c_{out}})$, and produces a preimage after an expected $O(2^{4m-3c_{out}/4})$ time using $c_{out} + (4m - 3c_{out}/4) = 4m + c_{out}/4$ bits of r_{out}. Hence, repeating this attack 2^m times using m additional bits of r_{out} to produce 2^m distinct preimages is expected to take $O(\max(2^{c_{out}}, 2^{5m-3c_{out}/4}))$ time/memory using $5m + c_{out}/4$ bits of r_{out}. The optimal choice for c_{out} is $c_{out} = (20/7)m \approx 2.89m$, and with this choice the first step takes $O(2^{(20/7)m}) = o(2^{3.5m})$ time/memory and uses $(40/7)m < 8m$ bits of r_{out} (the remaining bits of r_{out} are set to zero).

6 Attacks on the Final Compression Function

This section presents collision attacks on the final compression function f_{out} (including the output transform). For a given $r \in \mathbb{Z}_{256}^m$, the attacks produce $s, s' \in \mathbb{Z}_{256}^m$ with $s \neq s'$ such that $f_{out}(r, s) = f_{out}(r, s')$. To motivate these attacks, we note that they can be converted into a 'very long message' collision attack on the full LASH function, similar to the attack in Sect. 4. The two colliding messages will have the same final non-zero message block, and all preceding message blocks will be zero. To generate such a message pair, the attacker chooses a random $(8m-8)$-bit final message block (common to both messages), pads with a 0x80 byte, and applies the internal compression function f (with zero chaining value) to get a value $r \in \mathbb{Z}_{256}^m$. Then using the collision attack on f_{out} the attacker finds two distinct length fields $s, s' \in \mathbb{Z}_{256}^m$ such that $f_{out}(r, s) = f_{out}(r, s')$. Moreover, s, s' must be congruent to $8m-8 \pmod{8m}$ due to the padding scheme. For LASH-160, we can force s, s' to be congruent to $8m - 8 \pmod{64}$ by choosing the six LS bits of the length, so this leaves a $1/5^2$ chance that both inputs will be valid.

The lengths s, s' produced by the attacks in this section are very long (longer than $2^{x/2}$). However, we hope the ideas here can be used for future improved attacks.

6.1 Generalized Birthday Attack on the Final Compression

The authors of [3] describe an application of Wagner's generalized birthday attack [13] to compute a collision for the internal compression function f using $O(2^{2x/3})$ time and memory. Although this 'cubic root' complexity is lower than the generic 'square-root' complexity of the birthday attack on the full compression function, it is still higher than the $O(2^{x/2})$ birthday attack complexity on the *full* function, due to the final transformation outputting only half the bytes. Here we describe a variant of Wagner's attack for finding a collision in the *final* compression including the final transform (so the output bit length is x bits). The asymptotic complexity of our attack is $O\left(x2^{\frac{x}{2(1+\frac{107}{105})}}\right)$ time and memory – slightly better than a 'fourth-root' attack. For simplicity, we can call the running time $O(x2^{x/4})$.

The basic idea of our attack is to use the linear representation of f_{out} from Sect. 3.2 and apply a variant of Wagner's attack [13], modified to carefully deal with additive carries in the final transform. As in Wagner's original attack, we build a binary tree of lists with 8 leaves. At the ith level of the tree, we merge pairs of lists by looking for pairs of entries (one from each list) such that their sums have $7-i$ zero MS bits in selected output bytes, for $i = 0, 1, 2$. This ensures that the list at the root level has 4 zero MS bits on the selected bytes (these 4 MS bits are the output bits), accounting for the effect of carries during the merging process. More precise details are given below.

The attack. The attack uses inputs r, s for which the internal compression function f has a linear representation absorbing the Miyaguchi-Preneel feedforward (see Section 3.2). For such inputs, which may be of length up to $9m$ bit (recall: $m = x/4$), the *final* compression function $f' : \mathbb{Z}_{256}^{9m} \to \mathbb{Z}_{16}^{m}$ has the form

$$f'(r) = MS_4(H' \cdot [\mathsf{Rep}(r)]^T), \tag{10}$$

where $MS_4 : \mathbb{Z}_{256}^{m} \to \mathbb{Z}_{16}^{m}$ keeps only the 4 MS bits of each byte of its input, concatanating the resulting 4 bit strings (note that we use r here to represent the whole input of the linearised compression function f' defined in Section 3.2). Let $Rep(r) = (r[0], r[2], \ldots, r[9m-1]) \in \mathbb{Z}_{256}^{9m}$ with $r[i] \in \{0,1\}$ for $i = 0, \ldots, 9m-1$. Let $\ell \approx \lfloor \frac{4m}{2(1+107/105)} \rfloor$ (notice that $8\ell < 9m$). We refer to each component $r[i]$ of r as an *input bit.* We choose a subset of 8ℓ input bits from r and partition the subset into 8 substrings $r^i \in \mathbb{Z}_{256}^{\ell}$ $(i = 1, \ldots, 8)$ each containing ℓ input bits, i.e.

$r = (r^1, r^2, \ldots, r^8)$. The linearity of (10) gives

$$f'(r) = MS_4(H'_1 \cdot [r^1]^T + \cdots + H'_8 \cdot [r^8]^T),$$

where, for $i = 1, \ldots, 8$, H'_i denotes the $m \times \ell$ submatrix of H' consisting of the ℓ columns indexed $(i-1)\cdot\ell, (i-1)\cdot\ell+1, \ldots, i\cdot\ell-1$ in H'. Following Wagner [13], we build 8 lists $L_1, \ldots, L_8$, where the ith list L_i contains all 2^ℓ possible candidates for the pair (r^i, y^i), where $y^i \overset{\text{def}}{=} H'_i \cdot [r^i]^T$ (note that y^i can be easily computed when needed from r^i and need not be stored). We then use a binary tree algorithm described below to gradually merge these 8 lists into a single list L^3 containing 2^ℓ entries of the form $(r, y = H' \cdot [r]^T)$, where the 4 MS bits in each of the first α bytes of y are zero, for some α, to be defined below. Finally, we search the list L^3 for a pair of entries which match on the values of the 4 MS bits of the last $m - \alpha$ bytes of the y portion of the entries, giving a collision for f' with the output being α zero half-bytes followed by $m - \alpha$ random half-bytes.

The list merging algorithm operates as follows. The algorithm is given the 8 lists $L_1, \ldots, L_8$. Consider a binary tree with $c = 8$ leaf nodes at level 0. For $i = 1, \ldots, 8$, we label the ith leaf node with the list L_i. Then, for each jth internal node n^i_j of the tree at level $i \in \{1, 2, 3\}$, we construct a list L^i_j labelling node n^i_j, which is obtained by merging the lists L^{i-1}_A, L^{i-1}_B at level $i-1$ associated with the two parent nodes of n^i_j. The list L^i_j is constructed so that for $i \in \{1, \ldots, 3\}$, the entries (r', y') of all lists at level i have the following properties:

- $(r', y') = (r'_A || r'_B, y'_A + y'_B)$, where (r'_A, y'_A) is an entry from the left parent list L^{i-1}_A and (r'_B, y'_B) is an entry from the right parent list L^{i-1}_B.
- If $i \geq 1$, the $\lceil \ell/7 \rceil$ bytes of y' at positions $0, \ldots, \lceil \ell/7 \rceil - 1$ each have their $(7-i)$ MS bits all equal to zero.
- If $i \geq 2$, the $\lceil \ell/6 \rceil$ bytes of y' at positions $\lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$ each have their $(7-i)$ MS bits all equal to zero.
- If $i = 3$, the $\lceil \ell/5 \rceil$ bytes of y' at positions $\lceil \ell/7 \rceil + \lceil \ell/6 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil + \lceil \ell/5 \rceil - 1$ each have their $(7-i) = 4$ MS bits all equal to zero.

The above properties guarantee that all entries in the single list at level 3 are of the form $(r, y = H' \cdot [\mathsf{Rep}(r)]^T)$, where the first $\alpha = \lceil \ell/7 \rceil + \lceil \ell/6 \rceil + \lceil \ell/5 \rceil$ bytes of y all have 7-3=4 MS bits equal to zero, as required.

To satisfy the above properties, we use a hash table lookup procedure, which aims, when merging two lists at level i, to fix the $7-i$ MS bits of some of the sum bytes to zero. This procedure runs as follows, given two lists L_A^{i-1}, L_B^{i-1} from level $i-1$ to be merged into a single list L^i at level i:

- Store the first component r'_A of all entries (r'_A, y'_A) of L_A^{i-1} in a hash table T_A, indexed by the hash of:
  - If $i = 1$, the 7 MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of y'_A, i.e. string $(MS_7(y'_A[0]), \ldots, MS_7(y'_A[\lceil \ell/7 \rceil - 1]))$.
  - If $i = 2$, the 6 MS bits of bytes $\lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$ of y'_A, i.e. string $(MS_6(y'_A[\lceil \ell/7 \rceil]), \ldots, MS_6(y'_A[\lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1]))$.
  - If $i = 3$, the 5 MS bits of bytes $\lceil \ell/7 \rceil + \lceil \ell/6 \rceil, \ldots, \alpha - 1$ of y'_A, i.e. string $(MS_5(y'_A[\lceil \ell/7 \rceil + \lceil \ell/6 \rceil]), \ldots, MS_6(y'_A[\alpha - 1]))$.
- For each entry (r'_B, y'_B) of L_B^{i-1}, look in hash table T_A for matching entry (r'_A, y'_A) of L_A^{i-1} such that:
  - If $i = 1$, the 7 MS bits of corresponding bytes in positions $0, \ldots, \lceil \ell/7 \rceil - 1$ add up to zero modulo $2^7 = 128$, i.e. $MS_7(y'_A[j]) \equiv -MS_7(y'_B[j]) \bmod 2^7$ for $j = 0, \ldots, \lceil \ell/7 \rceil - 1$.
  - If $i = 2$, the 6 MS bits of corresponding bytes in positions $\lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$ add up to zero modulo $2^6 = 64$, i.e. $MS_6(y'_A[j]) \equiv -MS_6(y'_B[j]) \bmod 2^6$ for $j = \lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$.
  - If $i = 3$, the 5 MS bits of corresponding bytes in positions $\lceil \ell/7 \rceil + \lceil \ell/6 \rceil, \ldots, \alpha - 1$ add up to zero modulo $2^5 = 32$, i.e. $MS_5(y'_A[j]) \equiv -MS_5(y'_B[j]) \bmod 2^5$ for $j = \lceil \ell/7 \rceil + \lceil \ell/6 \rceil, \ldots, \alpha - 1$.
- For each pair of matching entries $(r'_A, y'_A) \in L_A^{i-1}$ and $(r'_B, y'_B) \in L_B^{i-1}$, add the entry $(r'_A \| r'_B, y'_A + y'_B)$ to list L^i.

Correctness. The correctness of the merging algorithm follows from the following simple fact:

Fact If $x, y \in \mathbb{Z}_{256}$, and the k MS bits of x and y (each regarded as the binary representation of an integer in $\{0, \ldots, 2^k - 1\}$) add up to zero modulo 2^k, then the $(k-1)$ MS bits of the byte $x + y$ (in $\mathbb{Z}_{256}$) are zero.

Thus, if $i = 1$, the merging lookup procedure ensures, by the Fact above, that the $7 - 1 = 6$ MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of $y'_A + y'_B$ are zero, whereas for $i \geq 2$, we have as an induction hypothesis that the $7 - (i-1)$ MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of both y'_A and y'_B are zero, so again by the Fact above, we conclude that the $7 - i$ MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of $y'_A + y'_B$ are zero, which proves inductively the desired property for bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ for all $i \geq 1$. A similar argument proves the desired property for all bytes in positions $0, \ldots, \alpha - 1$. Consequently, at the end of the merging process at level $i = 3$, we have that all entries (r, y) of list L^3 have the $7 - 3 = 4$ MS bits of bytes $0, \ldots, \alpha - 1$ being zero, as required.

Asymptotic Complexity. The lists at level $i = 0$ have $|L^0| = 2^\ell$ entries. To estimate the expected size $|L^1|$ of the lists at level $i = 1$, we model the entries (r^0, y^0) of level 0 lists as having uniformly random and independent y^0 components. Hence for any pair of entries $(r^0_A, y^0_A) \in L^0_A$ and $(r^0_B, y^0_B) \in L^0_B$ from lists L^0_A L^0_B to be merged, the probability that the 7 MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of y^0_A and y^0_B are negatives of each other modulo 2^7 is $\frac{1}{2^{\lceil \ell/7 \rceil \times 7}}$. Thus, the total expected number of matching pairs (and hence entries in the merged list L^1) is

$$|L^1| = \frac{|L^0_A| \times |L^0_B|}{2^{\lceil \ell/7 \rceil \times 7}} = \frac{2^{2\ell}}{2^{\lceil \ell/7 \rceil \times 7}} = 2^{\ell + O(1)}.$$

Similarly, for level $i = 2$, we model bytes $\lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$ as uniformly random and independent bytes, and with the expected sizes $|L^1| = 2^{\ell+O(1)}$ of the lists from level 1, we estimate the expected size $|L^2|$ of the level 2 lists as:

$$|L^2| = \frac{|L^1_A| \times |L^1_B|}{2^{\lceil \ell/6 \rceil \times 6}} = 2^{\ell + O(1)},$$

and a similar argument gives also $|L^3| = 2^{\ell+O(1)}$ for the expected size of the final list. The entries (r, y) of L^3 have zeros in the 4 MS bits of bytes $0, \ldots, \alpha - 1$, and random values in the remaining $m - \alpha$ bytes. The final stage of the attack searches $|L^3|$ for two entries with a identical values for the 4 MS bits of each of these remaining $m - \alpha$ bytes. Modelling those bytes as uniformly random and independent we have by a birthday paradox argument that a collision will be found with high constant probability as long as the condition $|L^3| \geq \sqrt{2^{4(m-\alpha)}}$ holds. Using $|L^3| = 2^{\ell+O(1)}$ and recalling that $\alpha = \lceil \ell/7 \rceil + \lceil \ell/6 \rceil + \lceil \ell/5 \rceil = (1/7 + 1/6 + 1/5)\ell + O(1) = \frac{107}{210}\ell + O(1)$, we obtain the attack success requirement

$$\ell \geq \frac{4m}{2(1 + \frac{107}{105})} + O(1) \approx \frac{x}{4} + O(1).$$

Hence, asymptotically, using $\ell \approx \lfloor \frac{x}{2(1+107/105)} \rfloor$, the asymptotic memory complexity of our attack is $O(x 2^{\frac{x}{2(1+\frac{107}{105})}}) \approx O(x 2^{x/4})$ bit, and the total running time is also $O(x 2^{\frac{x}{2(1+\frac{107}{105})}}) \approx O(x 2^{x/4})$ bit operations. So asymptotically, we have a 'fourth-root' collision finding attack on the final compression function.

Concrete Example. For LASH-160, we expect a complexity in the order of 2^{40}. In practice, the $O(1)$ terms increase this a little. Table 2 summarises the requirements at each level of the merging tree for the attack with $\ell = 42$ (note that at level 2 we keep only 2^{41} of the 2^{42} number of expected list entries to reduce memory storage relative to the algorithm described above). It is not difficult to see that the merging tree algorithm can be implemented such that at most 4 lists are kept in memory at any one time. Hence, we may approximate the total attack memory requirement by 4 times the size of the largest list constructed in the attack, i.e. $2^{48.4}$ bytes of memory. The total attack time complexity is approximated by $\sum_{i=0}^{3} |L^i| \approx 2^{43.3}$ evaluations of the linearised LASH compression function f', plus $\sum_{i=0}^{3} 2^{3-i}|L^i| \approx 2^{46}$ hash table lookups. The resulting attack success probability (of finding a collision on the 72 random output bits among the 2^{37} entries of list L^3) is estimated to be about $1 - e^{-0.5 \cdot 2^{37}(2^{37}-1)/2^{160-88}} \approx 0.86$. The total number of input bits used to form the collision is $8\ell = 336$ bit, which is less than the number $9m = 360$ bit available with the linear representation for the LASH compression function.

Table 2. Concrete Parameters of an attack on final compression function of LASH-160. For each level i, $|L^i|$ denotes the expected number of entries in the lists at level i, 'Forced Bytes' is the number of bytes whose $7-i$ MS bits are forced to zero by the hash table lookup process at this level, 'Zero bits' is four times the total number of output bytes whose 4 MS bits are guaranteed to be zero in list entries at this level, 'Mem/Item' is the memory requirement (in bit) per list item at this level, 'log(Mem)/List' is the base 2 logarithm of the total memory requirement (in bytes) for each list at this level (assuming that our hash table address space is twice the expected number of list items).

Level (i)	$log(\|L^i\|)$	**Forced Bytes**	**Zero bits**	**Mem/Item**, bit	log(`Mem`)**/List**, Byte
0	42	6	0	42	45.4
1	42	7	24	84	46.4
2	41	9	52	168	46.4
3	37		88	336	43.4

6.2 Heuristic Lattice-Based Attacks on the Final Compression

We investigated the performance of two heuristic lattice-based methods for finding collisions in truncated versions of the final compression function of LASH. The first reduces finding collisions to a lattice Shortest Vector Problem (SVP). The second uses the SVP as a preprocessing stage and applies a cycling attack with a lattice Closest Vector Problem (CVP) solved at each iteration.

First Method: SVP-Based Attack We assume that the r input to the final compression function is known and use the 'affine' representation (4) in Sect. 3.2 of the internal compression function, i.e. $g(s) = f(r, s) = H' \cdot s + \boldsymbol{b}$, with $m \times n$ matrix H' and $m \times 1$ vector $\boldsymbol{b}$. To find collisions in the final compression function truncated to $m' \leq m$ half-bytes using a subset of $n' \leq n$ input bits, we choose a $m' \times n'$ submatrix $\bar{H}$ of H' (we let $\boldsymbol{b}'$ denote the corresponding $m' \times 1$ subvector of $\boldsymbol{b}$) and set up a lattice $\mathcal{L}_{\bar{H}}$ spanned by the rows of the following $(n'+m') \times (n'+m')$ basis matrix:

$$M = \begin{pmatrix} B_1 \cdot I_{n'} & \bar{H}^T \\ 0 & 256 \cdot I_{m'} \end{pmatrix} .$$

Here, $B_1 \in \mathbb{Z}$ is a parameter with a typical value between 12 and 16, and $I_{n'}, I_{m'}$ denote identity matrices of size n' and m', respectively. We now run an SVP approximation algorithm (such as LLL or its variants) on M to find a short vector

$$\boldsymbol{v} = (v_0, \ldots, v_{n'-1}, v_{n'}, \ldots, v_{n'+m'-1})$$

in lattice $\mathcal{L}_{\bar{H}}$. Notice that by construction of $\mathcal{L}_{\bar{H}}$, for any lattice vector $\boldsymbol{v} \in \mathcal{L}_{\bar{H}}$ we have the relation

$$\sum_{i=0}^{n'-1} (v_i/B_1) \cdot \boldsymbol{h}_i \equiv (v_{n'}, \ldots, v_{n'+m'-1})^T \pmod{256} \ , \qquad (11)$$

where $\boldsymbol{h}_i \in \mathbb{Z}_{256}^{m'}$ denotes the ith column of $\bar{H}$ for $i = 0, \ldots, n'-1$.

We hope that $\boldsymbol{v}$ is 'good', i.e. has the following properties:

1 $v_i/B_1 \in \{-1, 0, 1\}$ for all $i = 0, \ldots, n'-1$.
2 $|v_i| < 16$ for all $i = n', \ldots, n'+m'-1$.

We choose n' to guarantee that such 'good' lattice vectors exist. Namely, suppose that we model the last m' coordinates of a lattice vector $\boldsymbol{v}$ as an independent uniformly random vector in $\mathbb{Z}_{256}^{m'}$, for each choice of the first n' coordinates of $\boldsymbol{v} \in \{-B_1, 0, B_1\}$. Then we expect that one of the resulting $3^{n'}$ lattice vector has $|\boldsymbol{v}[i]| < 16$ for $i = n', \ldots, n'+m'-1$ as long as $3^{n'}(31/256)^{m'} \geq 1$, which leads to the condition $n' \geq (\log(256/31)/\log(3)) \cdot m' \approx 1.92m'$ (we remark that a rigorous argument using Minkowski's Theorem shows that a 'good' lattice vector is guaranteed to exist if $8 < B_1 < 16$ and $n' > m'/(1 - \log(B_1)/4)$).

If $\boldsymbol{v}$ is 'good', then rearranging (11) yields the following relation in $\mathbb{Z}_{256}^{m'}$:

$$\sum_{i:v_i>0} \boldsymbol{h}_i = \sum_{i:v_i<0} \boldsymbol{h}_i + (v_{n'}, \ldots, v_{n'+m'-1})^T \ .$$

Let $\boldsymbol{t}_1 = \sum_{i:v_i>0} \boldsymbol{h}_i \in \mathbb{Z}_{256}^{m'} + \boldsymbol{b}'$, $\boldsymbol{t}_2 = \sum_{i:v_i<0} \boldsymbol{h}_i \in \mathbb{Z}_{256}^{m'} + \boldsymbol{b}'$, and $\boldsymbol{e} = (v_{n'}, \ldots,\ v_{n'+m'-1})^T \in \{-15, \ldots, +15\}^{m'}$. To obtain a collision for the final compression function, we need that the 4 MS bits of the bytes in $\boldsymbol{t}_1$ match the 4 MS bits in the corresponding bytes of $\boldsymbol{t}_2$, i.e. we need that the addition of the error vector $\boldsymbol{e}$ to $\boldsymbol{t}_2$ doesn't affect the 4 MS bits of the bytes of $\boldsymbol{t}_2$. This happens if and only if for each (jth) byte $\boldsymbol{t}_2[j]$ of $\boldsymbol{t}_2$, we have

$$LS_4(\boldsymbol{t}_2[j]) \in \begin{cases} \{0, \ldots, 15 - \boldsymbol{e}[j]\} & \text{if } \boldsymbol{e}[j] \geq 0 \ , \\ \{|\boldsymbol{e}[j]|, \ldots, 15\} & \text{if } \boldsymbol{e}[j] < 0 \ . \end{cases} \tag{12}$$

Here $LS_4(\boldsymbol{t}_2[j])$ denotes the 4 LS bits of byte $\boldsymbol{t}_2[j]$. Hence, for each j, there are $(16 - |\boldsymbol{e}[j]|)$ 'good' values for $LS_4(\boldsymbol{t}_2[j])$ which lead to a collision on the 4 MS bits of that output byte. Modelling the bytes of $\boldsymbol{t}_2$ as uniformly random and independent, we thus expect that all m' bytes of $\boldsymbol{t}_2$ are good (and hence we get an m'-byte collision for the final compression function) with probability $p_{good} = \prod_{j=0}^{m'-1} \frac{16-|\boldsymbol{e}[j]|}{16}$.

Rather than running the costly SVP algorithm about $k \overset{\text{def}}{=} 1/p_{good}$ times using different subsets of n' input bits, we suggest a much faster alternative. We run the SVP algorithm just once to get a single $(\boldsymbol{t}_1, \boldsymbol{t}_2)$ pair with additive difference vector $\boldsymbol{e} = \boldsymbol{t}_1 - \boldsymbol{t}_2$, and then generate about k additional pairs $\boldsymbol{t}_1^i, \boldsymbol{t}_2^i$ with the same additive difference vector $\boldsymbol{e}$, by adding k common shift vectors $\boldsymbol{\delta}^i$ to both $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$, i.e. $\boldsymbol{t}_1^i = \boldsymbol{t}_1 + \boldsymbol{\delta}^i$, $\boldsymbol{t}_2^i = \boldsymbol{t}_2 + \boldsymbol{\delta}^i$ for $i = 1, \ldots, k$. The common shift vectors $\boldsymbol{\delta}^i$ are generated as all 0-1 linear combinations of about $\log(k)$ unused columns of H' (i.e. columns of H indexed by input bits which are not in the subset of n' bits used in the submatrix $\bar{H}$). Modelling these k shift vectors $\boldsymbol{\delta}^i$ as independent uniformly random vectors, we expect to obtain a good $\boldsymbol{t}_2^i = \boldsymbol{t}_2 + \boldsymbol{\delta}^i$ among those candidates, investing at most $\log(k)$ vector additions per trial (or even one vector addition/subtraction per trial if we use a Gray code sequence of 0-1 combinations for the input bits used for generating the shift vectors).

Experimental Results. The largest partial collision we obtained for the final compression function with this attack was with $n' = 85$, $m' = 30$ (120 colliding bits out of 160) using reduction time 9639 sec plus a post computation time of 22611 sec on a 1.6GHz PC (a good shift vector was found after about $2^{35.5}$ trials, close to the expected number $k \approx 2^{36.3}$).

This is much lower than the 2^{60} hash computations needed to do this via a birthday paradox approach. The partial collision is shown in Fig. 5.

First Input	**Second Input**
$r\|\|s$ (first 20 bytes):	$r\|\|s$ (first 20 bytes):
30 22 44 e2 f0 04 21 74 30 00 c2 de 57 e1 73 80 00 00 00 00	80 00 2a 08 02 00 80 09 05 20 02 de 57 e1 73 80 00 00 00 00
hash:	hash:
4f 04 45 2f 29 a5 95 ab ec 52 a0 17 8e 62 80 85 62 9f b3 64	4f 04 45 2f 29 a5 95 ab ec 52 a0 17 8e 62 80 e0 44 f7 50 89

Fig. 5. Two final compression function inputs that match on the top 4 MS bits of 30 bytes of the output (all input bits which are not shown are zero).

This attack generates long colliding inputs of bit length $n' + \log(k)$. However, with better lattice reduction the value of n' might be shortened (heuristically $n' \geq 1.92m'$ should suffice, hence even $n' \approx 58$ for $m' = 30$ may work). Furthermore, we can reduce the number $\log(1/p_{good})$ of additional input bits for generating the 'postprocessing' shift vectors by instead flipping the values of input bits which have the same values among the n' bits used in the lattice reduction.

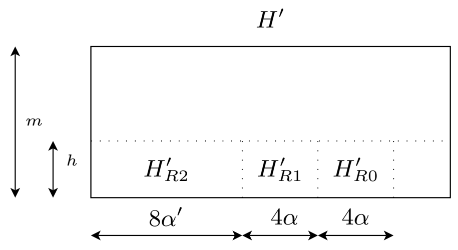
Second Approach: CVP-Based Attack Like the attack in Section 4, the idea of this approach is to run a Pollard rho cycle attack on the final compression function, and force some of the output bytes to zero in each iteration to reduce the size S of the output space. The attack in Section 4 used a table lookup approach to force c output bits to zero at the expense of 2^c table storage and computation. Here, we aim to force c bits to zero at each iteration using lattice techniques *without* the expense of 2^c storage, thus achieving similar run-time but without the necessity of large storage.

The Attack. As in the previous attack, we assume that the r input is known and use the 'affine' representation (4) of the final compression function output in terms of s, i.e. $g(s) = f(r, s) = H' \cdot s + \boldsymbol{b}$, with $m \times n$ matrix H' and $m \times 1$ vector $\boldsymbol{b}$. Fix attack parameters $h \leq m$ (the number of output half-bytes we attempt to force to zero at each Pollard iteration) and $\alpha \geq h/2$.

We define a Pollard iteration map $g : \mathbb{Z}_{256}^{\alpha'} \to \mathbb{Z}_{256}^{\alpha'}$ with $\alpha' \overset{\text{def}}{=} \frac{m}{2} - \frac{3}{8}h$ as follows.

Referring to Fig. 6, let $H'_R = [H'_{R2} H'_{R1} H'_{R0}]$ denote the $h \times 8 \cdot (\alpha' + \alpha)$ submatrix of H' consisting of the intersection of the h bottom rows and

Fig. 6. Submatrices denoted as H'_{R2}, H'_{R1}, H'_{R0} are taken from the bottom left part of the matrix H_R. They correspond to the first α', $\alpha/2$ and $\alpha/2$ bytes of the vector s.



$8 \cdot (\alpha' + \alpha)$ leftmost columns of H'. Let t' denote the bottom h bytes of the compression function output (before truncating 4 LS bits per byte), and $s' = [s'_2 s'_1 s'_0]^T$ denote the top $\alpha' + \alpha$ bytes of s, where $s'_2 \in \mathbb{Z}_{256}^{\alpha'}$ and $s'_1, s'_0 \in \mathbb{Z}_{256}^{\alpha/2}$. From Fig. 2 we have (assuming $\alpha + \alpha' \leq m$), that

$$t' = H'_{R2} \cdot \mathsf{Rep}(s'_2) + H'_{R1} \cdot \mathsf{Rep}(s'_1) + H'_{R0} \cdot \mathsf{Rep}(s'_0). \qquad (13)$$

On input $\bar{s} \in \mathbb{Z}_{256}^{\alpha'}$, the Pollard function g sets $s'_2 = \bar{s}$, and deterministically computes values for s'_1 and s'_0 to attempt to set the 4 MS bits of each byte of t' to zero. Namely, if $lsb(s'_2) = 0$ ('Case 0'), g sets $s'_1 = 0$ and finds a value for $\mathsf{Rep}(s'_0) \in \{-1, 0, 1\}^{4\alpha}$. Otherwise, if $lsb(s'_2) = 1$ ('Case 1'), g sets $s'_0 = 0$ and finds a value for $\mathsf{Rep}(s'_1) \in \{-1, 0, 1\}^{4\alpha}$. Consider first 'Case 0'. Referring to (13), let $\boldsymbol{y} = -H'_{R2} \cdot \mathsf{Rep}(s'_2) \in \mathbb{Z}_{256}^{h}$. Then g computes $\mathsf{Rep}(s'_0) \in \{-1, 0, 1\}^{4\alpha}$ such that $H'_{R0} \cdot \mathsf{Rep}(s'_0) \approx \boldsymbol{y}$. To do so, g sets up lattice $\mathcal{L}_0$ spanned by the rows of the following $(4\alpha + h) \times (4\alpha + h)$ basis matrix:

$$M_0 = \begin{pmatrix} B_1 \cdot I_{4\alpha} & [H'_{R0}]^T \\ 0 & 256 \cdot I_h \end{pmatrix}.$$

Note that this lattice is of the same form as the one used in Sec 6.2 (with B_1 an integer value between 12 and 16). Now g runs a Closest Vector Problem (CVP) approximation algorithm (such as the Babai algorithm [1] and its variants) on M_0 to find a lattice vector

$$\boldsymbol{v} = (v_0, \ldots, v_{4\alpha-1}, v_{4\alpha}, \ldots, v_{4\alpha+h-1}) \in \mathbb{Z}^{4\alpha+h}$$

which is 'close' to the target vector

$$\boldsymbol{y}' = (0, \ldots, 0, \boldsymbol{y}) \in \mathbb{Z}^{4\alpha+h}.$$

We set $\mathsf{Rep}(s'_0)[i] = \boldsymbol{v}[i]/B_1$ for $i = 0, \dots, 4\alpha - 1$. Note that at this point we hope that $\boldsymbol{v}$ is sufficiently close to $\boldsymbol{y}'$ so that

$$\mathsf{Rep}(s'_0) \in \{-1, 0, 1\}^{4\alpha} \text{ and } |\boldsymbol{v}[i] - \boldsymbol{y}'[i]| < 16 \text{ for } i = 4\alpha, \dots, 4\alpha + h - 1, \tag{14}$$

although it suffices if this happens for a noticeable fraction of inputs to g (see analysis later). If (14) is satisfied then $t' = H'_{R2} \cdot \mathsf{Rep}(s'_2) + H'_{R0} \cdot \mathsf{Rep}(s'_0) \equiv \boldsymbol{\delta} \pmod{256}$ for some $\boldsymbol{\delta} \in \{-15, \dots, 15\}^h$, and hence $MS_4(t'[i]) \in \{0, 15\}$ for $i = 0, \dots, h - 1$ (i.e. the 4 MS bits of the output bytes are 'approximately' zero in the sense that there are only two possible values for these 4 MS bits). In 'Case 1', g performs a similar CVP computation finding $\mathsf{Rep}(s'_1)$ as the computation of $\mathsf{Rep}(s'_0)$ in 'Case 0', where the submatrix H'_{R0} above is replaced by the submatrix H'_{R1}, yielding a lattice basis matrix M_1.

Finally, the Pollard iteration output $g(\bar{s}) \in \mathbb{Z}_{256}^{\alpha'}$ is defined as the concatenation of two strings derived from t' computed from (13):

- The h bit string $\boldsymbol{d} \in \{0,1\}^h$, where $\boldsymbol{d}[i] = 0$ iff $MS_4(t'[i]) = 0$.
- The $4 \cdot (m - h)$ bit string consisting of the top $m - h$ half bytes of $H' \cdot [s'_2 s'_1 s'_0 0^{m-(\alpha+\alpha')}]$.

Note that the byte length of $g(\bar{s})$ is $(h + 4 \cdot (m - h))/8 = m/2 - 3/8h \overset{\text{def}}{=} \alpha'$, as required. This completes the description of g.

Crucial Remark. The Babai CVP approximation algorithm can be separated into two steps. The first (more computationally intensive) 'preprocessing step' does not depend on the target vector, and involves computing a reduced basis for the lattice and the associated Gram-Schmidt orthogonalization of the reduced basis. The second (faster) 'online step' involves projecting the target vector on the Gram-Schmidt basis and rounding the resulting projection coefficients to construct the close lattice vector. In our Pollard iteration function g, we only have two fixed basis matrices (M_0 for 'Case 0' and an analogous basis M_1 for 'Case 1'). Hence we need only run the time consuming preprocessing step twice, and then in each Pollard rho iteration g only runs the fast 'online step' using the appropriate precomputed bases.

The attack iterates the Pollard rho iteration function g on a random initial value $\bar{s} \in \mathbb{Z}_{256}^{\alpha'}$. After a sufficient number of iterations (in the order of $2^{8\alpha'/2}$), we expect to find a collision in g, which gives us two compression function *ternary* inputs $s' = [s'_2 s'_1 s'_0]^T$ and $\bar{s}' = [\bar{s}'_2 \bar{s}'_1 \bar{s}'_0]^T$ for which the corresponding compression function outputs $\boldsymbol{t}, \bar{\boldsymbol{t}} \in \mathbb{Z}_{256}^m$ match on the 4 MS bits of all m bytes. Moreover, we hope that $lsb(s'_2) \neq lsb(\bar{s}'_2)$. Suppose, without loss of generality, that $lsb(s'_2) = 0$ and $lsb(\bar{s}'_2) = 1$. We therefore have:

$$\boldsymbol{t} = \sum_{i=0}^{8\alpha'-1} \mathsf{Rep}(s'_2)[i] \cdot \boldsymbol{h}_R^i + \sum_{i=8\alpha'+4\alpha+1}^{8\alpha'+8\alpha-1} \mathsf{Rep}(s'_0)[i-(8\alpha'+4\alpha)] \cdot \boldsymbol{h}_R^i,$$

and

$$\bar{\boldsymbol{t}} = \sum_{i=0}^{8\alpha'-1} \mathsf{Rep}(\bar{s}'_2)[i] \cdot \boldsymbol{h}_R^i + \sum_{i=8\alpha'}^{8\alpha'+4\alpha-1} \mathsf{Rep}(\bar{s}'_1)[i-8\alpha'] \cdot \boldsymbol{h}_R^i,$$

where $\boldsymbol{h}_R^i$ denotes the ith column of H'. From the equality of the 4 MS bits of all m bytes of $\boldsymbol{t}$ and $\bar{\boldsymbol{t}}$ we have

$$\bar{\boldsymbol{t}} = \boldsymbol{t} + \boldsymbol{e} \;\; ,$$

where $\boldsymbol{e} \in \{-15, \ldots, +15\}^m$. Therefore, rearranging this relation to have only 0-1 linear combination coefficients on each side (by moving vectors with -1 coefficients to the other side), we get a relation of the form:

$$\sum_{i=0}^{8\alpha'-1} \mathsf{Rep}(\bar{s}'_2)[i] \cdot \boldsymbol{h}_R^i + \sum_{i:\mathsf{Rep}(\bar{s}'_1)[i-8\alpha']=1} \boldsymbol{h}_R^i + \sum_{i:\mathsf{Rep}(s'_0)[i-(8\alpha'+4\alpha)]=-1} \boldsymbol{h}_R^i$$
$$= \sum_{i=0}^{8\alpha'-1} \mathsf{Rep}(s'_2)[i] \cdot \boldsymbol{h}_R^i + \sum_{i:\mathsf{Rep}(\bar{s}'_1)[i-8\alpha']=-1} \boldsymbol{h}_R^i + \sum_{i:\mathsf{Rep}(s'_0)[i-(8\alpha'+4\alpha)]=1} \boldsymbol{h}_R^i + \boldsymbol{e}.$$

Hence, we are now back to the situation encountered in the SVP-based attack above, where we have two 0-1 inputs to the compression function, such that the corresponding output vectors differ by the vector $\boldsymbol{e} \in \{-15, \ldots, +15\}^m$, and hence match on the 4 MS bits of all m bytes with probability $p_{good} = \prod_{i=0}^{m-1} \frac{16-|\boldsymbol{e}[i]|}{16}$, and we apply the the same 'post-processing' technique (adding about $1/p_{good}$ shift vectors generated by all 0-1 combinations of $\log(1/p_{good})$ unused input columns) until we get a collision on the 4 MS bits of all m output bytes.

Heuristic Complexity Analysis. The memory complexity for this attack is very small. The time complexity T is the sum of three components: (1) The preprocessing time T_{pre} for the CVP algorithm, (2) The time T_ρ for the Pollard rho attack to produce a collision with $\{-1, 0, 1\}$ coefficients, and (3) The postprocessing time T_{post} for transforming the $\{-1, 0, 1\}$ coefficient collision into a $\{0, 1\}$ coefficient collision.

The preprocessing time T_{pre} is dominated by the time to reduce the lattice bases M_0 and M_1. Using the 'block size' and 'pruning' parameters of the NTL BKZ lattice reduction routines [12] we can trade off quality of the reduction (which reduces the expected run-time T_{rho} of the Pollard rho step (see Table 3 below) at the expense of an increased preprocessing time T_{pre}.

The Pollard rho step run-time T_ρ is of the form $N_\rho \cdot T_{itr}$, where N_ρ is the expected number of Pollard rho iterations required to obtain a 'good' collision in the Pollard iteration function g, and T_{itr} is the time per iteration, which is dominated by the 'online step' of the CVP algorithm.

Let $S = 2^{4m-3h}$ denote the size of the space in which g is iterated. Let p_g denote the probability (over a random target vector) that the CVP algorithm returns a 'good' vector, i.e. vector $\boldsymbol{v}$ with $\boldsymbol{v}[i]/B_1 \in \{-1, 0, 1\}$ for $i = 0, \ldots, 4\alpha - 1$ and $|\boldsymbol{v}[i]| < 16$ for $i \geq 4\alpha$. Out of N_ρ iterations, we expect $N_\rho \cdot p_g$ iterations to produce 'good' vectors. Hence by a birthday argument we expect to get a collision with high constant probability if $N_\rho \cdot p_g \cdot \frac{1}{2} \geq \sqrt{S}$, where the factor of $\frac{1}{2}$ accounts also for the probability that the collision is 'good' also in the sense that $lsb(s_2') \neq lsb(\bar{s}_2')$. Using $S = 2^{4m-3h} = 2^{4m-3c/4}$ we get

$$N_\rho \approx 2^{1+2m-3h/2}/p_g. \tag{15}$$

The probability p_g can be determined experimentally for a given reduced basis. It seems to be difficult to estimate by theoretical arguments. However, we note that the parameter choice $\alpha \geq h/2$ is made to ensure (heuristically) that a 'good' vector $\boldsymbol{v}$ above will exist. Namely, suppose that we heuristically model the last h coordinates of a lattice vector $\boldsymbol{v} \in \mathcal{L}_0$ as an independent uniformly random vector in $\mathbb{Z}_{256}^h$, for each choice for the first 4α coordinates of $\boldsymbol{v} \in \{-B_1, 0, B_1\}$. Then we expect that one of the resulting $3^{4\alpha}$ lattice vector has $|\boldsymbol{v}[i] - \boldsymbol{y}'[i]| < 16$ for $i = 4\alpha, \ldots, 4\alpha + h - 1$ as long as $3^{4\alpha}(31/256)^h \geq 1$, which leads to the condition $\alpha \geq (\log(256/31)/(4\log 3)) \cdot h \approx h/2$.

The postprocessing time T_{post} is estimated by $1/p_{good}$ shift vector additions, where $p_{good} = \prod_{i=0}^{m-1} \frac{16-|\boldsymbol{e}[i]|}{16}$ is the probability that a random shift vector yields a collision on all output half bytes. Modelling the error vector elements $\boldsymbol{e}[i]$ as uniformly random in $\{-15, \ldots, +15\}$ and independent, the expected value of $\frac{16}{16-|\boldsymbol{e}[i]|}$ is 3.46, so the expected value of $1/p_{good} = \prod_{i=0}^{m-1} \frac{16}{16-|\boldsymbol{e}[i]|}$ is $3.46^m \approx 2^{0.448x}$. Hence $T_{post} \approx 2^{0.448x} \cdot T_{add}$, where T_{add} is the time to add/subtract an m-byte vector (assuming we use a Gray code sequence for enumerating the input bit combinations producing the tested shift vectors). We note that this may be a pessimistic estimate for T_{post} since the error coordinates $\boldsymbol{e}[i]$ are likely to be biased towards small absolute values, rather than being uniformly random in $\{-15, \ldots, +15\}$. To get a better estimate one can compute the average value of $1/p_{good}$ for the outputs produced by the CVP algorithm.

Concrete Estimates for LASH-160. Table 3 summarises our experimental results for estimating the complexity of this attack on LASH-160.

Table 3. Experimental results for CVP attack on LASH-160. Refer to text for explanation of table headings.

h	4α	b	p	$\log(T_{pre})$	$\log(1/p_g)$	$\log(N_\rho)$	$\log(T_{itr})$	$\log(T_\rho)$	n_i
20	70	55	12	**23.2**	7.7	58.7	9.2	**68.0**	224

In Table 3, the unit of time used is one LASH-160 compression function evaluation, which is taken to be $392.83 \times 40 \approx 15713$ Pentium cycles, as reported in implementation results in [3]. The two most important parameters $\log(T_{pre})$ (measured preprocessing step time) and $\log(T_\rho)$ (estimated Pollard rho step time) are shown in bold. For all the tabulated cases, the postprocessing step time T_{post} is $T_{post} \approx 2^{0.448 \times 160} \cdot T_{add} \approx 2^{64}$ compression function evaluations, using the estimate $T_{add} \approx 80$ cycles.

Additional remarks on Table 3. The parameters b and p denote block size and prune parameters, respectively, used for the NTL BKZ lattice reduction algorithm [12] in the preprocessing step. Time T_{itr} is the measured time for the 'online' CVP step, approximating the time for one evaluation of the Pollard iteration function g. The probability of a 'good' vector p_g was estimated by running the 'online' step of the CVP algorithm 1000 times, each time with a new and uniformly random target vector $\boldsymbol{y} \in \mathbb{Z}_{256}^h$, counting the number n_{nb} of runs for which $\boldsymbol{v}[i]/B_1 \in \{-1, 0, 1\}$ for $i = 0, \ldots, 4\alpha - 1$, the number n_{nm} for which $|\boldsymbol{v}[i] - \boldsymbol{y}'[i]| < 16$ for $i = 4\alpha, \ldots, 4\alpha + h - 1$, and estimating $p_g \approx \frac{n_{nb}}{1000} \times \frac{n_{nm}}{1000}$. The parameter n_{in} shows the bit length of each of the colliding inputs to the final compression functions produced by the attack.

From the results in the table, we therefore estimate that with the right choice of parameters, this attack can find collisions in the final compression of LASH-160 using about 2^{68} total run-time and very little memory.

References

1. L. Babai. On Lovasz' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

2. M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology – CRYPTO '96*, volume 1109 of *LNCS*, pages 1–15. Springer, 1996.
3. K. Bentahar, D. Page, M.-J. O. Saarinen, J. H. Silverman, and N. Smart. LASH. Second Cryptographic Hash Workshop, August, 24–25 2006.
4. D. J. Bernstein. Circuits for integer factorization: A proposal. Web page, http://cr.yp.to/papers/nfscircuit.pdf.
5. D. J. Bernstein. What output size resists collisions in a xor of independent expansions? ECRYPT Hash Workshop, May 2007.
6. S. Contini, R. Steinfeld, J. Pieprzyk, and K. Matusiewicz. A critical look at cryptographic hash function literature. ECRYPT Hash Workshop, May 2007.
7. O. Goldreich, S. Goldwasser, and S. Halevi. Collision-free hashing from lattice problems. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(042), 1996.
8. S. Lucks. Failure-friendly design principle for hash functions. In *Advances in Cryptology – ASIACRYPT '05*, volume 3788 of *LNCS*, pages 474–494. Springer, 2005.
9. V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. Provably Secure FFT Hashing (+ comments on "probably secure" hash functions). Second Cryptographic Hash Workshop, August, 24–25 2006.
10. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
11. C. Peikert. Private Communication, August 2007.
12. V. Shoup. NTL: A library for doing number theory. http://www.shoup.net/ntl/.
13. D. Wagner. A generalized birthday problem. In *Advances in Cryptology – CRYPTO '02*, volume 2442 of *LNCS*, pages 288–303. Springer, 2002.
14. M. J. Wiener. The full cost of cryptanalytic attacks. *J. Cryptol.*, 17(2):105–124, 2004.