LASH

K. Bentahar, D. Page, J.H. Silverman, M.-J. O. Saarinen and N.P. Smart

Abstract—**We present a practical cryptographic hash function based on the Miyaguchi–Preneel construction, which instead of using a block cipher as the main component uses a modular matrix multiplication. Thus as the core component it uses a compression function which is closely related to the theoretical lattice based hash function considered by Goldreich, Goldwasser and Halevi. We show that by suitable parameter choices we can produce a hash function which is comparable in performance to existing deployed hash functions such as SHA-1 and SHA-2.**

I. Introduction

In the last few years a number of weaknesses have been found in standardised hash functions such as MD4, MD5, RIPEMD and SHA-1 [4], [26], [27]. All of these hash functions are essentially derived from the same design and are constructed using somewhat ad-hoc techniques. In contrast, other areas of cryptography have replaced ad-hoc construction with well defined sets of design principles. Examples include the wide-trail design strategy of AES [8, Chapter 9], or the rigorous application of reductionist provable security techniques as in the context of RSA-OEAP [3], [11]. While the SHA-2 family of hash functions is not yet known to succumb to the recent attack techniques, its design principles are so similar to SHA-1 that we have no guarantee an attack will not appear in the near future.

Much is known theoretically about how to construct hash functions from one-way functions, yet these theoretical results do not aid one in designing efficient and practical realisations. One problem with previous attempts to design hash functions based on hard computational problems, for example the MASH-1 algorithm [1], has been that the result is not competitive in terms of performance. Even so, interest in hash functions which are provably reducible to hard computation problems has recently been reawakened, see for example the hash function VSH [6] which is based on the difficulty of factoring. VSH is faster than MASH-1, but still significantly slower than standard hash functions. The output block length is fixed to the size of an RSA-modulus, although of course this may be truncated in an actual application, and its design criteria mentions nothing about pre-image resistance. Additionally, VSH raises the question as to who actually generates the hard problem on which the hash functions security is based, i.e. the prime factorisation of the RSA-modulus.

In 1996, Goldreich, Goldwasser and Halevi [14] presented a hash function whose collision resistance could be related to the worst case of the problem of finding small vectors in lattices. In the tradition of provable security, it was shown that any algorithm which found collisions for such a function could be used to solve the problem of finding short vectors in lattices. The reduction to the worst case of this latter problem was made using ideas of Ajtai [2].

The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT. The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Clearly, when using the construction of Goldreich et. al. in a practical hash function one would use their method as a compression function and then extend the domain to an arbitrary length using a construction like that of Merkle and Damgård [20], [9]. This construction provides a provably secure collision resistant hash function, under the assumption that the compression function is itself collision resistant. When combined with the technique of Goldreich et. al. one obtains a collision resistant hash function which can take arbitrary length inputs. Recent work showing that the MerkleĐam gård construction is weak in certain circumstances [16], [17] can be resolved with minor alterations, see for example [7], [18].

The problem with the construction of a compression function using the ideas of Goldreich et. al. is that, with the parameters needed so as to reduce the underlying lattice problem to the worst case scenario, the resulting hash function is not very efficient. In addition it appears hard to directly develop a hash function which meets a specific security gaurantee required by the practical community, for example if the output hash size is n bits in length then it should require $2^{n/2}$ operations to find a collision. One can show (see later) that collisions can be found in the construction of Goldreich et. al. using $2^{n/3}$ operations, or $2^{n/4}$ operations if one is using the GGH construction with the MerkleĐamg ård construction to extend the input domain.

However, one can take the idea behind the construction of Goldreich et al. and try to obtain an efficient hash function whose security is related to finding short vectors in a particular fixed lattice. One would then need to study whether this lattice behaved as a random lattice, and that the underlying hard problem was actually secure. In this paper, we take this latter approach and present an efficient (supposedly) collision resistant hash function whose performance is comparable to that of SHA-2. The design has been motivated by implementation quality, including issues such as speed and memory footprint, and the ability to fully utilise processor features available in current computer architectures. We present this proposal for a hash function simply to stimulate the community into considering hash function whose components are easy to analyse mathematically.

This paper is organized as follows. In Section II we present the LASH algorithm, then in Section III we present the properties behind each component we have used and we justify the design principles we have used. In Section IV we present our preliminary overview of possible security weaknesses. In Section V we give an overview of the performance of the algorithm. In the Appendix we present a more detailed mathematical analysis of the linear component of our scheme, which is essentially the compression function of Goldreich et. al.

We end this introduction by commenting on the name LASH. LASH stand for a number of possible acronyms.

- **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.

II. LASH

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 that $m = n/16$.

A. Pseudorandom Sequence

Consider the following pseudorandom sequence. Start with $y_0 = 54321$ and iterate the following recurrance, based on the Pollard generator,

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

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

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

B. Compression Function

We define a compression function f that takes in two byte sequences $r_0, r_1, \ldots, r_{m-1}$ and $s_0, s_1, \ldots, s_{m-1}$ and produces a new byte sequence $t_0, t_1, \ldots, t_{m-1}$. Algorithm 1 gives the LASH compression function, where we use $\oplus$ to denote the exclusive-or operator.

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 $a_0, a_1, \ldots$, above, 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 combination helps defeat the attacks on simply using the Goldreich et. al. construction on its own.

Algorithm 1 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
    $x \leftarrow \lfloor 2^{-(7-(i \bmod 8))} r_{\lfloor i/8 \rfloor} \rfloor \bmod 2$
  else
    $x \leftarrow \lfloor 2^{-(7-(i \bmod 8))} s_{(\lfloor i/8 \rfloor - m)} \rfloor \bmod 2$
  end if
  if $x = 1$ then
    for $j = 0, 1, \ldots, m-1$ do
      $t_j \leftarrow t_j + a_{((n+j-i) \bmod n)} \bmod 256$
    end for
  end if
end for
return $\mathbf{t}$
```

C. Hashing the message

Let l be the length of the original message in bits. The individual message bytes are $v_0, v_1, v_2, \ldots$. The message is padded with a single 1 bit (in case of byte-aligned data, a single byte with hexadecimal value `0x80`). The rest of the v_i values are taken to be be zeros.

The message is cut into $k = \lceil l/8m \rceil$ blocks of m bytes and fed to the compression function, and then a final transform is performed, which involves applying the compression function to the chaining variable and an encoding of l, to produce a message digest. Algorithm 2 describes the overall hash function.

Algorithm 2 LASH

```
for $i = 0, 1, \ldots, m-1$ do
  $r_i = 0$
end for
for $i = 0, 1, \ldots, \lceil l/8m \rceil - 1$ do
  for $j = 0, 1, \ldots, m-1$ do
    $s_i = v_{m \times i + j}$
  end for
  $\mathbf{r} \leftarrow f(\mathbf{r}, \mathbf{s})$
end for
for $i = 0, 1, \ldots, m-1$ do
  $s_i \leftarrow \lfloor l/2^{8i} \rfloor \bmod 256$
end for
$\mathbf{r} \leftarrow f(\mathbf{r}, \mathbf{s})$
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 $\mathbf{t}$
```

III. Design Overview

In this section we go into more detail over the precise design choices we have made. The goals of the design have been as follows:

- To adopt the large-pipe strategy of Lucks [18] to avoid problems with the Merkle–Damgård construction. The final hash value being produced from the large-pipe by taking the upper bits of each byte, these being the bits which depend in the most non-linear manner on the input values.
- To combine two forms of mathematical operation in the compression function, arithmetic modulo 256 and bitwise exclusive-or. Thus the compression functions consists of two parts, a linear function (motivated by the lattice based hash function of Goldreich et. al. [14]) and a xoring of the chaining variable and the next message block (motivated by the construction of Miyaguchi–Preneel [21], [23]).
- To be able to reason about the ability of the linear function to resist preimages and collisions.
- To be as simple and efficient as possible, particularly aiming for application on as wide a range of platforms as possible. Thus the hash function is byte oriented and built out of components found on all processors and which are easy to implement in hardware.
- To enable as much parallelism as possible, thus allowing the hash function to exploit performance enhancing features in modern instruction sets.
- The hash function should be patent free, as such none of the designers have taken out patents on its design.

A. Linear Function

In this subsection we consider the function considered by Goldreich et. al. [14] as a basis for their theoretical lattice based hash function.

Let L denote an n-dimensional lattice in $\mathbb{R}^n$ generated by an integral basis matrix B, with discriminant $\Delta(L)$. We denote by $\lambda(L)$ the length of the shortest non-zero vector in L and by $\lambda(L, \mathbf{b})$ the length of the closest lattice vector to the arbitrary vector $\mathbf{b} \in \mathbb{R}^n$. A binary (resp. ternary) vector in the lattice L is defined to be a vector in L whose coordinates are restricted to come from the set $\{0, 1\}$ (resp. $\{-1, 0, 1\}$). The set of all binary (resp. ternary) vectors in $\mathbb{R}^n$ will be denoted by $\mathcal{B}_n$ (resp. $\mathcal{T}_n$).

Let H denote an integral $m \times n$ matrix, and let q denote some fixed integer. Note, in what follows one should not think of q as being prime. We define a map f_H by

$$f_H : \begin{array}{ccc} \{0,1\}^n & \longrightarrow & (\mathbb{Z}/q\mathbb{Z})^m \\ \mathbf{b} & \longmapsto & H \cdot \mathbf{b} \pmod q \end{array} \tag{1}$$

and a lattice L_H by

$$L_H = \{\mathbf{x} \in \mathbb{Z}^n : H\mathbf{x} = 0 \pmod q\}. \tag{2}$$

Since $q\mathbb{Z}^n \subset L_H \subset \mathbb{Z}^n$, it is clear that $\dim(L_H) = n$. If the map f_H is surjective, then there is an exact sequence

$$0 \longrightarrow L_H \longrightarrow \mathbb{Z}^n \xrightarrow{H} (\mathbb{Z}/q\mathbb{Z})^m \longrightarrow 0.$$

This allows us to compute the discriminant of the matrix L_H,

$$\Delta(L_H) = [\mathbb{Z}^n : L_H] = \#(\mathbb{Z}/q\mathbb{Z})^m = q^m.$$

A basis matrix for the lattice H can be derived as follows. First, form the Smith Normal Form (SNF) of H as

$$S_H = UHV.$$

If we let r denote the rank of H, then the lattice L_H is spanned by the first r rows of V^t. When the corresponding diagonal entry $s_{i,i}$ of S is not equal to one, we multiply the corresponding row of V^t by $q/s_{i,i} \pmod q$. This $r \times n$ matrix is then augmented with the rows of the $n \times n$ matrix qI_n. A basis from this spanning set can then be obtained in the standard manner. We define B_H to be the row-oriented basis matrix obtained in this way.

Alternatively if f_H is already known to be surjective then a basis for the lattice can be derived by taking the $n \times (n - m)$-kernel matrix K_H of H over the integers. For convenience we write this as

$$K = \begin{pmatrix} K^* \\ I_{n-m} \end{pmatrix},$$

for some $m \times (n - m)$ matrix K^*. A basis for our lattice L_H can then be obtained from the rows of the matrix

$$\begin{pmatrix} (K^*)^t & I_{n-m} \\ qI_m & 0 \end{pmatrix}.$$

Goldreich, Goldwasser and Halevi [14] show that, for a suitably chosen matrix $H \in M_{m,n}(\mathbb{F}_q)$, if the map f_H defined by (1) is collision resistant, then it is hard to find small non-zero ternary vectors in the lattice L_H. More precisely, they show that if m, n, and q satisfy

$$m \log q < n < \frac{q}{2m^4}, \tag{3}$$

with $q = O(n^c)$ for some constant $c > 0$, then the difficulty of finding collisions for f_H is equivalent to the *worst* case of the approximate shortest vector problem apprSVP in a lattice of dimension m. Their proof builds on the work of Ajtai [2], who proves the average case/worst case equivalence of certain lattice problems.

Goldreich et. al. propose that the function f_H is suitable as a cryptographic hash function. However, in practice things are not so clear cut. As m and n go to infinity, constants and even log factors may not be of great theoretical importance. However, in practise a cryptographic system is likely to employ lattices of dimension a few hundred, or maybe a few thousand. In those cases, the constants and log factors are significant. For example, an algorithm that finds collisions in dimension $n = 500$ can be turned into an algorithm to solve apprSVP in dimension m, but only with $m \leq 11$. Similarly, finding collisions in dimension $n = 1000$ gives an apprSVP algorithm in dimension at most $m = 20$; and even dimension $n = 10000$ gives an apprSVP algorithm in dimension at most $m = 150$.

Given the efficiency of LLL-type algorithms in low dimension, it thus appears that the practical security of hash functions based directly on the compression function f_H must depend on the average case difficulty of solving Ajtai's problem itself in high dimension, rather than on the derived difficulty of solving worst case APPRSVP in much lower dimension.

If using the output of the linear function f_H as the hash value one does not achieve the concrete security level one would want in practice. The output hash size is q^m, and so one expects in practice that the best method for finding collisions will take time at least $q^{m/2}$ operations. In Sections C that one can find collisions in the function f_H in time significantly shorter than this, and in Section D we show an improved attack assume the function f_H is used as the compression function in a MerkleĐamg årdconstruction.

Despite not being able to rely on the asymptotic worst case/average case analysis of [14], it is not hard to relate the security of the function f_H to the hardness of certain problems in L_H.

Proposition 1: (a) Inversion of f_H is equivalent to finding, for a given vector $\mathbf{a} \in \mathbb{R}^n$,a vector that differs from $\mathbf{a}$ by a binary vector, that is, finding a vector $\mathbf{x}$ satisfying

$$\mathbf{x} \in L_H \qquad \text{and} \qquad \mathbf{x} - \mathbf{a} \in \mathcal{B}_n.$$

In particular, such a vector $\mathbf{x}$ always satisfies $\|\mathbf{x} - \mathbf{a}\| \leq \sqrt{n}$, and on average it will satisfy $\|\mathbf{x} - \mathbf{a}\| \approx \sqrt{n/2}$.

(b) Finding a collision for f_H is equivalent to finding a nonzero ternary vector in L_H, that is, finding a vector in the intersection

$$\mathbf{x} \in \mathcal{T}_n \cap L_H \qquad \text{with } \mathbf{x} \neq 0.$$

In particular, such a collision-producing vector always satisfies $\|\mathbf{x}\| \leq \sqrt{n}$, and on average a collision gives a vector $\mathbf{x} \in L_H$ satisfying $\|\mathbf{x}\| \approx \sqrt{n/2}$.

This result appears in [10] and [14]. For completeness of the current paper, we include the elementary proof in Appendix A. We have made the conservative assumption that solving APPRSVP for the lattice L_H yields a collision for f_H, but this is actually only true if the solution is a ternary vector. A detailed analysis using standard assumptions, e.g., assuming that the collection of lattices $\{L_H\}$ satisfies the Gaussian heuristic (cf. [15], [19]), yields a more precise statement. One finds that for the suggested LASH parameters, solving APPRSVP in L_H to within a factor of approximately 2.5 is likely to yield a ternary vector, and hence a collision of f_H. In the opposite direction, solving APPRSVP in L_H to within a factor of (say) 1.8 is unlikely to yield a collision, since almost all vectors of this size in L_H are not ternary vectors. See Appendix C for details.

We now turn to issues as to how we selected the precise function f_H used in our construction, we therefore need to select m, n, q and the matrix entries of H.

We first look at the values (m, n, q):

- Due to the fact that finding collisions in f_H is easier than the naive $q^{m/2}$, we take m to be larger than one needs in our final hash function output. This is also useful to defeat various other generic attacks on hash functions and is consistent with the advice of Lucks [18].
- It turns out to be convenient in our chaining algorithm to select $n = 2m \log_2 q$.
- Whilst a value of $q = 2^{32}$ is more likely to place us in the range of the inequality (3), we have found via various experiments that since the output size of the hash function is fixed (and so m is limited), a harder lattice problem is produced if q is smaller. Hence, we select $q = 2^8$.

All that remains is to define the particular linear function f_H that we shall use, i.e., we need to describe the $m \times n$ matrix H. We take H to be the m-by-n circulant matrix associated to the sequence $a_0, \dots, a_n$ generated by the earlier PRNG,

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

The reasons for this choice are as follows. Firstly, the use of a circulant matrix allows more efficient implementations of our function f_H, and deriving the entries via a pseudorandom number generator allows us to reduce the memory requirements of our hash function. The choice of p in the Pollard generator is made to enable a sequence with period greater than the largest value of n and so $\sqrt{p}$ should be greater than the largest value of n chosen. In addition we selected a p for which modular reduction can be performed efficiently. The non-linearity of the generator is crucial in creating a matrix for which the associated lattice problem that is hard to solve. For example, we have found that using a linear-congruential PRNG instead of the Pollard PRNG results in a compression function that is easy to break.

B. Compression Function

Recall, the compression function for LASH is then defined from the m-byte chaining variable r and the next m-byte block s, via

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

The compression function is highly motivated by the construction of MiyaguchiP reneel [21], [23], which is of the form

$$f(r, s) = (r \oplus s) \oplus E_{g(r)}(s),$$

for a block cipher $E_k(m)$ and a function g which takes inputs the size of the chaining variable and outputs keys for the block cipher.

Thus we are treating the function f_H as equivalent to a block cipher with key r and message s. We are not claiming that the function f_H can be used as a block cipher.

Hence, the "proof of security" of the Miyaguchi–Preneel construction [5] does not apply in this situation.

However, the function f_H does have some interesting properties which it shares with a block cipher, naively

- Given an output f_H it is hard to invert, as shown in Proposition 1.
- It is hard to find collisions in the function f_H, again as shown in Proposition 1.

C. Final Transformation

In the final transform we need to compress the $8m$ bit chaining variable down to the output hash value of $4m$ bits. Recall that each byte of the chaining variable has been obtained by performing a lot of additions modulo $q = 256$, which have been dependent on the message bits.

To compute the final hash value we select the upper four bits of each byte of the chaining variable and concaternate them together. This produces an output of the correct size. The reason for taking the upper four bits, is that due to the nature of addition modulo q these are going to be the bits which are affected in the most non-linear manner due to the effect of carry propogation in the addition operations.

IV. Additional Security Considerations

The general structure of LASH, having only linear components, easily leads one to suspect that it is directly vulnerable to differential and linear cryptanalysis. LASH has gone through several evolutionary stages after the idea of a lattice-based hash function was first considered. The current version is a result of combining the traditions of provable complexity-theoretic security with symmetric cryptanalysis.

In determining the security of LASH against these attacks, we note that as a fully parameterisable hash function (message block size, state size, and hash result size can all be flexibly chosen), simulation of attacks against LASH is straightforward and meaningful. If an attack can be successfully mounted and simulated on reduced variants of LASH, and the asymptotic behavior of the security as a function of various parameters established, concrete evidence about the security of full-size variants is obtained. This flexibility also makes it easy to create larger versions of LASH if weaknesses are found in the current versions. This is a clear advantage of LASH over many hash function designs with a more rigid, block cipher - like structures.

A. Differential Cryptanalysis

A small input difference (in either the chaining variable and/or the message block) will result in a very large difference in the hash function state. Differential trails are very wide. The propagation of differentials is further amplified in final iteration (which does not use message bits), making all output bits differentially dependant on all input bits.

We conjecture that the simple and understandable structure of LASH will make it difficult to find differential anomalies such as the so-called necessary conditions exploited by Wang et al in their attacks on MD5, SHA-1, and other hash functions [26], [27], [28], [29].

B. Linear Cryptanalysis

All components of the LASH compression function are, in some sense, linear. Furthermore, if we consider a matrix H' that contains the least significant bits of H, then the product function $H' \cdot \mathbf{b}$ is a linear equation in $\mathbb{F}_2$ and indeed H' is invertible with a significant probability. This can be exploited in some attacks, as is shown in Section D. These attacks are difficult to extend to the full version of LASH, however.

It is unlikely that classical linear cryptanalysis (involving the parity of subsets of bits) can be applied on LASH.

C. Generalized Birthday Attack

Wagner's method for solving the generalized birthday problem [25] can be applied to the GGH construction. We will give a brief description of the algorithm and its limitations, a more detailed discussion is given in Appendix B. Using the GGH function f_H on its own implies we can find collisions in $O(q^{m/3})$ operations as opposed to the $O(q^{m/2})$ operations one would want in practice from a hash function.

Although improvements to this basic version of the attack can be made, this attack does not seem to be applicable to the internal f_H function used in LASH, due to the ratio between the message block size and the size of the internal state. This motivates our choice of a large chaining variable and our output transformation. Our use of the Miyaguchi–Preneel construction, as opposed to using the function f_H directly also helps defeat this attack.

D. A Hybrid Attack

We will outline a hybrid attack that combines cycle-based collision finding techniques with linear algebra and a time-memory trade-off against the GGH function applied directly to multi-block messages using the Merkle–Damgård constuction, i.e. Lash with a different compression function, i.e. the function f_H as the compression function, and no output transform.

The general strategy of the attack is to try to select two-block messages in a way that forces a cycle-based collision finding algorithm such as [22] into a smaller cycle, thus producing collisions faster. If the outputs belong to a subset S of possible outputs, collision search will have $O(\sqrt{\mid S \mid})$ complexity, assuming that the message selection process is $O(1)$.

The messages are chosen as follows. The first block of the message contains the output of the previous iteration in the collision finding algorithm. The message bits in the second block are chosen in a way that causes a number of bits in the internal state of the hash function be to zero, hence forcing the final output to a smaller subset of possible outputs. The algorithm for selecting the second message block requires $O(1)$ time. The message selection algorithm is as follows:

1. Since carry propagation in addition is from least significant bits towards higher bits, $H \cdot \mathbf{b} \pmod 2$ is in fact a system of linear equations in $\mathbb{F}_2$, independent

of the 7 higher bits in each byte of H. Using simple linear algebra operations in $\mathbb{F}_2$, bit 0 in each of the m state bytes can be forced to zero by selecting m message bits appropriately. This is an $O(1)$ step.
2. A precomputed lookup table is used to force further c bits to zero. The table has 2^c entries and uses $m+c$ message bits (since the table entries must also have least significant bits as zeros). Each lookup requires $O(1)$ time. The precomputation phase requires $O(2^c)$ time.

Thus, by selecting $2m+c$ message bits in the second block in a certain way, $m+c$ bits in the $8m$-bit internal state are forced to zero. The offline complexity of the attack is $O(2^c)$ and the collision search algorithm is expected to find a collision in $O(2^{\frac{1}{2}(7m-c)})$ steps.

First consider the hypothetical case where LASH would have the standard Merkle-Damgård structure. In this case the internal state would have the same size as the final output, i.e. $8m$ bits. If we choose $c = \frac{7}{3}m \approx 2.33m$, the overall complexity of the algorithm will be $O(2^{2.33m})$, which is significantly less than $O(2^{4m})$ expected by direct application of the birthday paradox. However, since the internal state of LASH is twice as wide as the final output, the security goal of LASH is $O(2^{2m})$. This is the rationale behind the final transformation of LASH.

We note that it is possible to also force bit 1 of each byte to zero if the message block is large enough so that additional m^2 message bits can be selected. This is why a relatively short message block size is being used (larger message blocks would have resulted in greater hashing speed).

V. Implementation

A. Storage of Pseudorandom Data

We have several options as regards storage of the pseudorandom matrix. A compromise seems the most attractive option, that is to store only part of the matrix. Due the circulant nature, there is no real benefit in storing the whole matrix since each row is essentially a rotation of the first. Therefore, we can simply store one row and be able to access all the required elements by shifting a window from right to left; at each of n steps, the window contains the elements for the corresponding column.

The circulant nature of the matrix has an additional property in that neighbouring columns differ only in one element. Therefore, one can imagine storing only a single column of the matrix and updating it by computing a new entry at each step. This creates a computational overhead in that we need to generate a total of n matrix entries, but offers a saving in storage overhead since there are far less rows than columns in the matrix.

B. Parallelism in Compression Function

The basic algorithm for executing the compression function offers parallelism in two directions. Firstly, since they are essentially unrelated, one can operate on different columns of the matrix at once, summing the partial sums to form the final result. Secondly, one can add different elements of a given column into the state in parallel. These two method combine to offer a high degree of scalability. This is easy to exploit in hardware or where a dedicated SIMD instruction set is available.

We can manually apply a similar technique on processors which do not have SIMD instruction sets but do have a native word size greater than 8-bits. For example, on a 32-bit processor we can pack four 8-bit sub-words into one 32-bit value. We cannot add packed values using native 32-bit addition since carries from one sub-word may overflow into another. However, we can construct a suitable method for addition by masking the top bits of the packed bytes to prevent carries before using 32-bit addition and patching up the result. The resulting packed addition of x and y to produce the result r can be described as

$$\begin{array}{rcl} x' & \leftarrow & x \wedge \texttt{0x7F7F7F7F} \\ y' & \leftarrow & y \wedge \texttt{0x7F7F7F7F} \\ r' & \leftarrow & x' + y' \\ r & \leftarrow & ((x \oplus y) \wedge \texttt{0x80808080}) \oplus r' \end{array}$$

with a similar construction possible for other word sizes.

C. Specialisation of Compression Function

Considering how the compression function is used to process arbitrary length messages, the first and last invocations can be considered special. In the first invocation the chaining variable is zero; in the last invocation the message block is mostly zero with only a few bytes representing the message length. In both cases, only a small portion of the compression function input is relevant and in the first case the initial mixing stage is redundant since $t_i = r_i \oplus s_i = s_i$ for all i.

The saving afforded from capitalising on these features by using specialised versions of the compression function is amortised over all invocations. For short messages, the saved computation can be significant since the first and last invocations of the compression function comprise the majority of the total.

D. Results

We recompiled and tested publicly available source code for the SHA1 and SHA2 hash functions [24], [12], [13], as well as preliminary implementations of LASH, on our experimental platform. This platform housed a 2.8 GHz Pentium 4 processor running the 2.4.21 Linux kernel. All source code was written in C, making use of GCC 4.0.1 and the intrinsics feature to access the SIMD functionality of the processor. Measurement of the number of cycles elapsed during execution was performed using the `rdtsc` instruction in the normal way.

Table I shows the results of the experiment and compares SHA1 and SHA2 with equivalent parameterisations of LASH. The results were averaged over a large number of random inputs; it is vital to note that LASH performance is variable depending on the input. Also note that the storage requirement is intended to detail only the amount of pre-computed material rather than the total memory footprint. The results show an encouraging ratio between the

TABLE I

A table comparing the performance of LASH with standardised hash functions with comparable parameterisations.

Name	Implementation	Storage	Cycles/byte
SHA1-160	without SIMD [24]	0 bytes	26.29
SHA1-160	with SIMD [12]	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
SHA2-256	without SIMD [24]	256 bytes	55.16
SHA2-256	without SIMD [13]	288 bytes	31.34
SHA2-256	with SIMD [12]	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
SHA2-384	without SIMD [24]	640 bytes	124.57
SHA2-384	without SIMD [13]	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
SHA2-512	without SIMD [24]	640 bytes	124.98
SHA2-512	without SIMD [13]	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

fastest implementations of LASH versus SHA1 and SHA2. In particular, LASH is potentially only 30 times slower than SHA1 with the ratio improving significantly for SHA2 with LASH being only 10 to 20 times slower. This is comparable at the lower security levels with an implementation of VSH, although this clearly depends on how large one takes the modulus in ones VSH implementation.

VI. Test vectors

We provide test vectors for each variant of LASH. The vectors are computed over two test messages. Message A consists of three lower-case ASCII characters "abc", whose corresponding hexadecimal bytes are `61 62 63`. The message length is 24 bits. Message B consists of 100000 repetitions of the ten ASCII characters "0123456789", whose corresponding hexadecimal bytes are `30 31 32 33 34 35 36 37 38 39`. The message length is 8 million bits.

```
LASH-160(A) =
        67 58 25 ec  f3 ba f5 c9  4f fe 38 a1  5b c0 ab 40
        77 9b 96 4d

LASH-160(B) =
        43 68 df 33  4f ce b9 e7  99 d2 77 22  12 fc 44 f2
        ce ec 04 1e

LASH-256(A) =
        39 ff b7 84  0b 6b 3b 71  89 fc 5e dc  9e 24 33 9e
        77 8c f4 be  bf 94 df 00  c3 53 d0 bf  37 30 b3 2f

LASH-256(B) =
        e9 57 75 d4  53 d6 36 1e  3c 9c 88 8c  dc eb 3c 8a
        ab 49 cd ad  43 56 b5 ba  97 98 38 6b  b6 dc 95 e9

LASH-384(A) =
        11 d0 9c 55  cb ba 6f 31  10 bf 87 7f  ab cf b6 30
        10 52 0c 30  76 e1 dc d2  7b af dc a8  38 5e 25 0e
        4e fa 42 97  a1 6c 69 23  b9 a1 33 3d  8d ca 1d a7

LASH-384(B) =
        41 7e cb d6  dd 54 2f 82  e4 29 e4 ec  93 e6 c0 78
        3d 81 7c 5e  38 4d d2 e4  97 61 6c b1  0f 32 6e b6
        10 5c ef 9e  32 ba 2f 97  9b 5e 94 8b  31 e7 8c 75

LASH-512(A) =
        c5 bb 7c f4  c1 ca c6 38  43 94 66 65  7c 8d ed 14
        bb ab f8 28  e4 b3 69 99  86 11 64 b9  79 2d 88 fd
        48 eb 0f aa  aa f4 e0 33  19 fc bd 4d  4e 5c 2c 06
        82 5a 85 97  35 98 69 dd  1e 84 0b 12  15 96 19 c8

LASH-512(B) =
        07 02 25 1f  85 b4 5a a7  78 0d f4 9d  69 b2 de b0
        20 12 c5 e3  20 46 7e 3b  04 a3 4f fa  75 a0 19 0d
        c8 f5 41 20  c2 33 a5 08  38 26 a8 e6  47 68 2c 5b
        59 c0 9e d2  52 c7 1e 81  66 f6 2e 59  ef fb 24 57
```

References

[1] ISO/IEC 10118-4. Information technology – Security techniques – Hash-functions – Part 4: Hash-functions using modular arithmetic. Draft, 1996.

[2] M. Ajtai. Generating hard instances of lattice problems. In *28th ACM Symposium on Theory of Computing*, 99-108, 1996.

[3] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology – EUROCRYPT '94*, Springer-Verlag LNCS 950, 92–111, 1994.

[4] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet and W. Jalby. Collisions of SHA-0 and reduced SHA-1. In *Advances in Cryptology – EUROCRYPT 2005*, Springer-Verlag LNCS 3494, 36–57, 2005.

[5] J. Black, P. Rogaway and T. Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In *Advances in Cryptology – CRYPTO 2002*, Springer-Verlag LNCS 2442, 320–335, 2002.

[6] S. Contini, A.K. Lenstra and R. Steinfeld. VSH, an efficient and provable collision resistant hash function. APR e-print 2005/193, 2005.

[7] J.-S. Coron, Y. Dodis, C. Malinaud and P. Puniya. Merkle–Damgård Revisted: How to construct a hash function. In *Advances in Cryptology – CRYPTO 2005*, Springer-Verlag LNCS 3621, 430–448, 2005.

[8] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag, 2002.

[9] I.B. Damgård. Collision free hash functions and public key signature schemes. In *Advances in Cryptology – EUROCRYPT 1987*, Springer-Verlag LNCS 304, 203–216, 1988.

[10] C. Dwork. Positive applications of lattices to cryptography. In *22nd International Symposium on Mathematical Foundations of Computer Science*, Springer-Verlag LNCS 1295, 44–51, 1997.

[11] E. Fujisaki, T. Okamoto, D. Pointcheval and J. Stern. RSA-OAEP is secure under the RSA assumption. In *Advances in Cryptology – CRYPTO 2001*, Springer-Verlag LNCS 2139, 260–274, 2001.

[12] D. Gaudet. SHA1 and SHA256 using SSE2. Available at: `http://www.arctic.org/~dean/crypto/`.

[13] O. Gay. SHA-224, SHA-256, SHA-384 and SHA-512. Available at: `http://www.ouah.org/ogay/sha2/`.

[14] O. Goldreich, S. Goldwasser and S. Halevi. Collision-free hashing from lattice problems. In *Electronic Colloquium on Computational Complexity TR96-042*, 1996.

[15] J. Hoffstein, J. Pipher and J.H. Silverman. NTRU: A new high speed public key cryptosystem, In *Algorithmic Number Theory – ANTS III*, Springer-Verlag LNCS 1423, 267–288, 1998.

[16] A. Joux. Multicollisions in iterated hash functions. Application to cascaded constructions In *Advances in Cryptology – CRYPTO 2004*, Springer-Verlag LNCS 3152, 306–316, 2004.

[17] J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *Advances in Cryptology – EUROCRYPT 2005*, Springer-Verlag LNCS 3495, 474–490, 2005.

[18] S. Lucks. Design principles for iterated hash functions. Cryptology ePrint Archive, 2004/253, 2004.

[19] A. May and J.H. Silverman. Dimension reduction methods for convolution modular lattices. In *Cryptography and Lattices Conference – CaLC 2001* Springer-Verlag LNCS 2146, 110–125, 2001.

[20] R.C. Merkle. A fast software one-way hash function. *Journal of Cryptology*, **3**, 43–58, 1990.

[21] S. Miyaguchi, K. Ohta and M. Iwata. 128-bit hash function (N-hash). *NTT Review*, **2**, 117–127, 1990.

[22] P. van Oorschot and M. Wiener. Parallel collision search with cryptanalytic applications. Journal of Cryptology, 12 (1999), p 1-28.

[23] B. Preneel. *Analysis and design of cryptographic hash functions.* PhD Thesis, KU Leuven, 1993.

[24] T. St Denis. LibTomCrypt: A Portable ISO C Cryptographic Toolkit. Available at: `http://libtomcrypt.org/`.

[25] D. Wagner. A generalized birthday problem. In *Advances in Cryptology – CRYPTO 2002*, Springer-Verlag LNCS 2442, 288–303, 2002.

[26] X. Wang, X. Lai, D. Feng, H. Chen and X. Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In *Advances in Cryptology – EUROCRYPT 2005*, Springer-Verlag LNCS 3494, 1–18, 2005.

[27] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Advances in Cryptology – EUROCRYPT 2005*, Springer-Verlag LNCS 3494, 19–35, 2005.

[28] X. Wang, H. Yu, and Y. L. Yin. Efficient Collision Search Attacks on SHA-0. In *Advances in Cryptology – CRYPTO 2005*, Springer-Verlag LNCS

[29] X. Wang, Y. Yin, H. Yu. Finding Collisions in the Full SHA-1. In *Advances in Cryptology – CRYPTO 2005*, Springer-Verlag LNCS

Appendices

A. Proof of Proposition 1

For (a), suppose that we are given $\mathbf{b} \in (\mathbb{Z}/q\mathbb{Z})^m$ and want to solve $f_H(\mathbf{y}) = \mathbf{b}$. We begin by finding any vector $\mathbf{a} \in \mathbb{Z}^n$ satisfying $H\mathbf{a} \equiv -\mathbf{b} \pmod{q}$. This is easy to do, since the congruence $H\mathbf{a} \equiv -\mathbf{b}$ has more variables than equations. Of course, we are assuming that there is at least one solution. Now the following problems are equivalent:

- Solve $f_H(\mathbf{y}) = \mathbf{b}$.
- Find $\mathbf{y} \in \mathcal{B}_n$ satisfying $H\mathbf{y} = \mathbf{b}$. (Since the domain of f_H is the set of binary vectors.)
- Find $\mathbf{y} \in \mathcal{B}_n$ satisfying $H(\mathbf{y} + \mathbf{a}) = 0$. (Since $\mathbf{b} = -H\mathbf{a}$.)
- Finding $\mathbf{x} \in L_H$ satisfying $\mathbf{x} - \mathbf{a} \in \mathcal{B}_n$ (Letting $\mathbf{x} = \mathbf{y} + \mathbf{a}$.)

This completes the proof of (a).

For (b), we first observe that if $f_H(\mathbf{x}) = f_H(\mathbf{y})$, then $\mathbf{x} - \mathbf{y} \in L_H$ and clearly $\mathbf{x} - \mathbf{y}$ is ternary. Conversely, suppose that $\mathbf{z} \in L_H$ is a ternary vector. Then $\mathbf{z}$ can be written as a difference $\mathbf{z} = \mathbf{x} - \mathbf{y}$ of binary vectors, so $f_H(\mathbf{x}) = f_H(\mathbf{y})$ and we have produced a collision.

Binary and ternary vectors of dimension n have length at most $\sqrt{n}$, and the average length of a binary vector is $\sqrt{n/2}$. The average length of a ternary vector is $\sqrt{2n/3}$, but the average length of the difference of two binary vectors (which is how the ternary vectors are being produced) is $\sqrt{n/2}$.

B. Finding collisions in the Goldreich et. al. construction in less than $q^{m/2}$ operations

In this section we describe in detail the attack outlined in Section C. In particular we show that for fixed parameter sizes one does not achieve the security one would hope for from the Goldreich et. al. construction.

The attack, pointed out to us by an anonymous referee for an earlier version of this manuscript which does not use the MiyaguchiP reneel scheme or the post processing step, finds a binary vector in the lattice associated to f_H in time $q^{m/3}$ and thus can be used to break the collision resistance of a hash function based soley on the Goldreich et. al. construction.

The attack works as follows: We assume f_H is surjective and write down the basis of the associated lattice as the rows of the matrix

$$\begin{matrix} (K_H^*)^T & I_{n-m} \\ qI_m & 0 \end{matrix} .$$

We now consider only vectors of the form $\mathbf{x} = (\mathbf{y}, 0)$ where $\mathbf{y} \in \mathcal{B}_{n-m}$. The vector $\mathbf{x}$ produces a lattice vector of the form $(\mathbf{x}(K_H^*)^t, \mathbf{x})$. If try to solve $\mathbf{x}(K_H^*)^t \pmod{q} = 0$ then the resulting lattice vector will be a binary vector in the lattice.

However, solving $\mathbf{x}(K_H^*)^t \pmod{q} = 0$ has been studied by Wagner [25] in terms of a k-sum generalisation of the birthday paradox. This is done as follows: We divide the $n - m$ row vectors of $(K_H^*)^t$ into four lists and place form $q^{m/3}$ combinations of the row vectors in each list. Then we use the technique of Wagner to find a subset sum equal to zero modulo q. We expect such a subset sum to exist since the values of the top m components of $(K_H^*)^t$ are essentially random elements modulo q. Thus the running time is $q^{m/3}$, which is the time to produce the lists and the time to run Wagner's algorithm.

One can extend this method by constructing a list of 2^d partial matrix-vector products by using d message bits in a message block and running through all combinations (i.e. subset sums of rows of $(K_H^*)^t$). By choosing another d message bits, another list of equal size can be produced. It is possible to merge these distinct lists in essentially $O(2^d)$ time to produce a third list of equal size that has the property of having d selected bits as zero. The process can be recursively applied in a tree-like fashion to produce a collision in kd bits of the internal state with the selection of $2^k d$ message bits and $O(2^{k+d})$ effort in optimal conditions.

C. Ternary Vectors in Lattices

In this section we develop the tools needed to analyze whether solutions to an approximate shortest vector problem in a lattice $L \subset \mathbb{Z}^n$ are likely or unlikely to be ternary vectors. This section aims to present an analysis on how hard it is to either invert or find collisions in the internal function f_H via lattice basis reduction. Before commencing we reiterate that finding collisions or inverting f_H is not sufficient to break LASH due to the use of the Miyaguchi–Preneel construction.

A. Which Balls Contain Many Ternary Lattice Points?

Let $\mathcal{T}_n$ be the set of ternary vectors of dimension n as usual, and let $B_n(R)$ be the ball of radius R centered at 0 in $\mathbb{R}^n$. If R is small, than most of the integral lattice points in $B_n(R)$ will be ternary vectors, while if R is large, then few of them will be ternary. We would like to determine a critical value R_n at which the ternary vectors cease to predominate. This should be roughly the value R such that the number of ternary vectors of norm at most R is equal to the volume of the ball of radius R, i.e., R_n solves the equation

$$\mathrm{Vol}(B_n(R)) = \#\bigl(\mathcal{T}_n \cap B_n(R)\bigr).$$

Using the formula for the volume of a ball and the counting formula for ternary vectors, we see that R_n solves

$$\frac{\pi^{n/2}}{\Gamma(n/2+1)}R^n = \sum_{d=0}^{\lfloor R^2 \rfloor} \binom{n}{d} 2^d. \tag{4}$$

The sum on the righthand side of (4) is a step function, so the equation (4) tends to have several solutions. For example, if $n = 100$, then (4) has 14 solutions ranging from 4.992 to 6.087. Although this does not give an exact solution, it tells us that a ball of radius 5 in $\mathbb{R}^{100}$ contains mostly ternary lattice points, while a ball of radius (say) 10 contains proportionally very few ternary lattice points. Table II gives the largest, smallest, and average solutions to (4) for a range of dimensions.

TABLE II
SOLUTIONS TO $\mathrm{Vol}(B_n(R)) = \#(\mathcal{T}_n \cap B_n(R))$

n	$R_n^{\min}$	R_n^{mean}	$R_n^{\max}$
50	3.15042	3.90777	4.58992
100	4.99171	5.55618	6.08738
150	6.32237	6.81316	7.28238
200	7.48077	7.90118	8.30731
250	8.48252	8.83002	9.16873
300	9.37782	9.69343	10.0022
350	10.1947	10.4858	10.7715
400	10.9082	11.2014	11.4894
450	11.6179	11.8743	12.1269
500	12.2867	12.5294	12.7689

It is clear from Table II that R_n^{mean} does not grow linearly with n. For our data, the regression line of $\log(R_n^{\text{mean}})$ versus $\log(n)$ is

$$\log(R_n^{\text{mean}}) \approx 0.50634 \log(n) - 0.6173 \tag{5}$$

with correlation coefficient 0.999996.

This suggests that $R_n \approx c\sqrt{n}$. We next relate the sum on the righthand side of (4) to a binomial distribution and use a normal approximation to prove the validity of this guess and find an asymptotic value for c.

Proposition 2: For large values of n, the equation

$$\frac{\pi^{n/2}}{\Gamma(n/2+1)}R^n = \sum_{0\le d\le R^2} \binom{n}{d} 2^d \tag{6}$$

has a solution R satisfying $R \approx 0.4332\sqrt{n}$. (This may be compared with the experimental value $R \approx 0.54 \cdot n^{0.506}$ given by (5).)

Proof: For any $r > 0$,

$$\sum_{0\le d\le r} \binom{n}{d} 2^d = 3^n \sum_{d=0}^{r} \binom{n}{d} \left(\frac{2}{3}\right)^d \left(\frac{1}{3}\right)^{n-d}$$

is 3^n times the probability that a binomial distribution (with probabilities 1/3 and 2/3) is smaller than r. If n is large, we can approximate this probability using the normal distribution

$$\begin{aligned}\Phi(x) &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-t^2/2}\,dt \\ &= \sqrt{\frac{2}{\pi}} \cdot \frac{1}{|x|} e^{-x^2/2} \left(1 + O(1/x^2)\right) \qquad \text{for } x < 0.\end{aligned}$$

Thus

$$\begin{aligned}\frac{1}{3^n}\sum_{0\le d\le r} \binom{n}{d} 2^d &= \sum_{d=0}^{r} \binom{n}{d} \left(\frac{2}{3}\right)^d \left(\frac{1}{3}\right)^{n-d} \\ &\sim \Phi\left(\frac{r - 2n/3}{\sqrt{2n/9}}\right) \qquad \text{as } n \to \infty.\end{aligned}$$

To ease notation, we let $r = \alpha n$ and set $\beta = (3\alpha - 2)/\sqrt{2}$, so the above quantity is $\Phi(\beta\sqrt{n})$.

Using the elementary asymptotic expansion for $\Phi(x)$ (valid for $x < 0$) and Sterling's formula to approximate $\Gamma(x)$, the equation (6) that we are trying to solve (with $R = \sqrt{r} = \sqrt{\alpha n}$) becomes

$$\begin{aligned}(2\pi e r/n)^{n/2} &\approx 3^n \Phi(\beta\sqrt{n}) \\ (2\pi e \alpha)^{n/2} &\approx 3^n \cdot \sqrt{\frac{2}{\pi}} \cdot \frac{1}{|\beta|\sqrt{n}} \cdot e^{-\beta^2 n/2}\end{aligned}$$

Taking n^{th} roots and letting n go to infinity gives the equation

$$\sqrt{2\pi e \alpha} = 3e^{-\beta^2/2}$$

to be solved for α, where recall that $\beta = (3\alpha - 2)/\sqrt{2}$. The numerical solution is $\alpha \approx 0.18762$, so we find that the solutions R to (6) are given approximately by $R = \sqrt{\alpha n} \approx 0.4332\sqrt{n}$. ■

B. Which General Lattice Problems Have Many Ternary Solutions?

Let $L \subset \mathbb{Z}^n$ be a lattice of dimension n and let $\lambda(L)$ denote the length of a shortest nonzero vector in L. Proposition 2 suggests that if $\lambda(L)$ is significantly smaller than $R_n \approx 0.4332\sqrt{n}$, then most solutions to APPRSVP will be ternary vectors, but if $\lambda(L)$ is significantly larger than R_n, then only a small proportion of the solutions to APPRSVP will be ternary vectors. Combining this observation with the value of $\lambda(L)$ given by the Gaussian heuristic yields the following result.

Proposition 3: Let $\mathcal{L}$ be a class of lattices for which the Gaussian heuristic is valid and fix $\epsilon > 0$. Then for $L \in \mathcal{L}_n$, solutions $\mathbf{v} \in L$ of APPRSVP satisfying

$$\|\mathbf{v}\| < (1-\epsilon) \cdot \frac{1.79}{\mathrm{Disc}(L)^{1/n}} \cdot \lambda(L)$$

are quite likely to be ternary vectors, while solutions $\mathbf{v} \in L$ of APPRSVP satisfying

$$\|\mathbf{v}\| > (1+\epsilon) \cdot \frac{1.79}{\mathrm{Disc}(L)^{1/n}} \cdot \lambda(L)$$

are unlikely to be ternary vectors.

In particular, if $\mathrm{Disc}(L)$ is significantly larger than 1.79^n, then even a shortest vector in L (i.e., a solution to SVP) is unlikely to be a ternary vector.

Proof: The Gaussian estimate says that the shortest nonzero vector in a "typical lattice" has length

$$\lambda(L) \approx \sqrt{n/2\pi e}\,\mathrm{Disc}(L)^{1/n}.$$

(See, e.g., [15], [19].) Solving apprSVP in L yields a vector of length $C\lambda(L)$ for some $C \geq 1$. Proposition 2 says that this vector is quite likely to be a ternary vector if $C\lambda(L) < 0.4332(1-\epsilon)\sqrt{n}$ and that it is not very likely to be a ternary vector if $C\lambda(L) > 0.4332(1+\epsilon)\sqrt{n}$. Thus the critical value for C is

$$\begin{aligned} C &= \frac{0.4332\sqrt{n}}{\lambda(L)} \approx 0.4332 \cdot \sqrt{2\pi e} \cdot \mathrm{Disc}(L)^{-1/n} \\ &\approx 1.79 \cdot \mathrm{Disc}(L)^{-1/n}. \end{aligned}$$

■

C. Which Lattice Problems Arising From f_H Have Many (or Mostly) Ternary Solutions?

If we are to base a hash function upon the linear function f_H, then we would want the difficulty of finding binary (resp. ternary vectors) in L_H to be at least as hard as inversion (resp. finding collisions) of f_H via generic methods. An interesting aspect of the lattices we shall use is that for a fixed output size of the linear function, the value $\Delta^{1/n}$ of the associated lattice tends to one as we increase the dimension of the lattice, i.e. the input block size of the linear function.

As indicated by Proposition 1, the ability of finding collisions in f_H depends on the difficulty of finding special sorts of short vectors in the circulant lattice L_H. The NTRU cryptosystem [15] is also based on the difficulty of finding short vectors in certain lattices (called convolution modular lattices in [19]) that are built up out of circulant matrices. However, the matrices (and lattices) underlying LASH are rather different from those underlying NTRU, so the associated lattice problems are also different.

We now apply the results of the previous section to the lattices L_H used by LASH. Recall that $\dim(L_H) = n$ and $\mathrm{Disc}(L_A) = q^m$. Notice that if we make the assumption that $q^m < 2^n$, which is required if f_H is to be a compression function, then $1 < \mathrm{Disc}(L_H)^{1/n} < 2$.

Proposition 4: Assume that the Gaussian heuristic holds for the LASH lattices (2).

(a) If $q^m > 1.8^n$, then solving apprSVP in L_H is unlikely to give a ternary vector.

(b) If $q^m < 1.78^n$, then solving apprSVP in L_H to within a factor of $1.79/q^{m/n}$ is quite likely to give a ternary vector.

Proof: This is immediate from Proposition 3 using the values $\dim(L_H) = n$ and $\mathrm{Disc}(L_A) = q^m$. ■

Finally, we apply Proposition 4 to the specific LASH parameters. In all cases we find that the LASH lattice is likely to contain many ternary vectors. The crucial quantity is the approximation factor $1.79/q^{m/n}$, which tells us how closely we need to solve apprSVP in order to (probably) find a ternary vector. The conclusion is that in order to find a collision in the linear function for the suggested parameters, it is probably necessary to find a vector in L_H that is no more than about 2.5 times as long as the shortest nonzero vector.

However, we note that finding collisions in the linear function f_H is not sufficient to find collisions in LASH itself.