The LANE hash function
Extended Abstract

Sebastiaan Indesteege, Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, and Elmar Tischhauser

[1] Department of Electrical Engineering ESAT/SCD-COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
sebastiaan.indesteege@esat.kuleuven.be
[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.

Abstract. In this document, we propose the cryptographic hash function LANE as a candidate for the SHA-3 competition [11] organised by NIST. LANE is an iterated hash function supporting multiple digest sizes. Components of the AES block cipher [3,9] are reused as building blocks. LANE aims to be secure, easy to understand, elegant and flexible in implementation.
We give the specification of LANE, and the rationale behind the important design choices. For a more extended specification, security analysis and a discussion of the implementation aspects, we refer to the LANE submission document [6].

Key words: LANE, SHA-3 candidate, hash function.

1 Introduction

LANE is an iterated cryptographic hash function, supporting digest sizes of 224, 256, 384 and 512 bits. These four variants of LANE are referred to as LANE-224, LANE-256, LANE-384 and LANE-512, respectively. The LANE hash functions reuse components from the AES block cipher [3,9]. These, and other building blocks are described in Sect. 3.

LANE supports the use of a salt value, if this is desirable for the application. A well-known example of such an application is password hashing. If a different salt is used for every stored password, it is no longer possible to attack multiple targets in parallel in a dictionary attack or an exhaustive search. Digital signatures are another application where a salt provides a benefit. This is referred to as *randomised hashing*, after the work of Halevi and Krawczyk [5].

Hashing a message with LANE is performed in three steps. In the first step, which is described in Sect. 4, the message is padded and split into message blocks of equal length. Also, the initial chaining value H_{-1} is set to the initial value $IV_{n,S}$, which depends on the digest size n and the (optional) salt value S.

In the second step, a compression function $f(\cdot,\cdot,\cdot)$ is applied iteratively:

$$H_i = f\left(H_{i-1}, M_i, C_i\right) \quad . \tag{1}$$

Each compression function call uses a message block M_i to update the chaining value H_{i-1} to H_i. A counter C_i, which indicates the number of message bits processed so far, including the message bits in the block M_i which is currently being processed, is also input into the compression function. The compression function of LANE is described in Sect. 5.

The third and final step is the output transformation, described in Sect. 6. In this step, the digest is derived from the final chaining value, using the message length l and the (optional) salt value S as additional inputs. It consists of a single compression function call and, depending on the digest length, a truncation of the result.

The iteration mode used in LANE was designed to be easy to understand and implement. It is based on the well-known Merkle-Damgård construction [4,8]. For this construction, it can be proven that if the compression function is collision resistant, so is the iterated hash function built on it.

2 Conventions

Throughout the specification of LANE, the big-endian convention is used, *i.e.*, the first (or leftmost) bit of a bit string is the most significant bit. Also for sequences of bytes, this convention is used.

As LANE reuses components of the AES block cipher [3,9], it is required to map a sequence of bytes into an *AES state*, a 4×4 array of bytes, and vice versa. This is done in the same way as for the AES, *i.e.*, the sequence of 16 bytes $y_0 \,||\, \cdots \,||\, y_{15}$ is mapped to the AES state

$$\begin{bmatrix} y_0 & y_4 & y_8 & y_{12} \\ y_1 & y_5 & y_9 & y_{13} \\ y_2 & y_6 & y_{10} & y_{14} \\ y_3 & y_7 & y_{11} & y_{15} \end{bmatrix} . \tag{2}$$

A LANE *state* is the state used inside the LANE compression function. In LANE-224 and LANE-256, a state of 256 bits is used, which corresponds to two AES states. In LANE-384 and LANE-512, the state is 512 bits in size, corresponding to four AES states. A sequence of 32 or 64 bytes can be mapped to two or four AES states, depending on the LANE variant, where the first (leftmost) 16 bytes map into the first (leftmost) AES state, and so on.

3 Building blocks

The LANE hash function reuses several components from the AES block cipher [3,9]. In particular, the SubBytes, ShiftRows and MixColumns transformations are also part of LANE. In LANE, however, they are used several times in parallel, due to the larger state size.

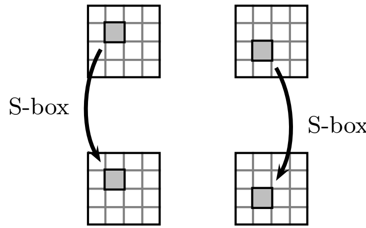

Fig. 1. The SubBytes transformation in Lane-224 and Lane-256.

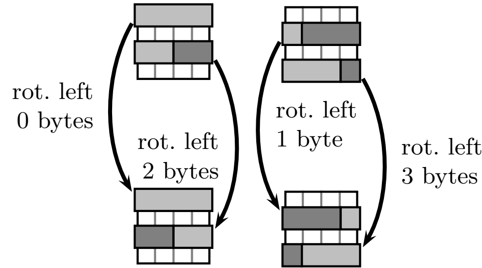

Fig. 2. The ShiftRows transformation in Lane-224 and Lane-256.

3.1 SubBytes

The SubBytes transformation in Lane is identical to the corresponding component of the AES block cipher, except that it operates on a larger state. Figure 1 illustrates this for Lane-224 and Lane-256. The same non-linear substitution (S-box) is applied to each of the state bytes independently. This S-box is the same as the S-box used in the AES block cipher [3,9]. For an exact definition of the S-box, we refer to the Lane specification document [6] or to [3,9].

3.2 ShiftRows

The ShiftRows transformation cyclically shifts the bytes of the rows of each of the AES states that comprise the Lane state. The first, *i.e.*, topmost row is not shifted. The second, third and fourth row are cyclically shifted to the left over one, two and three byte positions, respectively. This is identical to the ShiftRows transformation in the AES block cipher, except that it is applied two or four times in parallel, depending on the Lane variant. Figure 2 illustrates ShiftRows for Lane-224 and Lane-256.

3.3 MixColumns

The MixColumns transformation operates on the columns of the state. Each column is viewed as a polynomial over $\mathrm{GF}(2^8)$, *i.e.*, a polynomial of degree three with coefficients in $\mathrm{GF}(2^8)$. Then, this polynomial is multiplied modulo $Y^4 + 1$

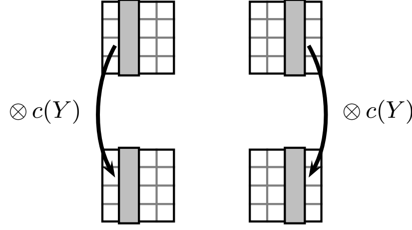

Fig. 3. The MixColumns transformation in LANE-224 and LANE-256.

```
1: k_0 ← 07fc703d_x
2: for i = 1 to 272 (resp. 768 for LANE-384 and LANE-512) do
3:     k_i = k_{i-1} ≫ 1
4:     if k_{i-1} ∧ 00000001_x then
5:         k_i = k_i ⊕ d0000001_x
6:     end if
7: end for
```

Fig. 4. Pseudocode for generating the LANE constants.

with the fixed polynomial $c(Y) = 03Y^3 + 01Y^2 + 01Y + 02$. Equivalently, this operation can be written as a matrix multiplication

$$\begin{bmatrix} y'_0 \\ y'_1 \\ y'_2 \\ y'_3 \end{bmatrix} = \begin{bmatrix} \mathtt{02}_x & \mathtt{03}_x & \mathtt{01}_x & \mathtt{01}_x \\ \mathtt{01}_x & \mathtt{02}_x & \mathtt{03}_x & \mathtt{01}_x \\ \mathtt{01}_x & \mathtt{01}_x & \mathtt{02}_x & \mathtt{03}_x \\ \mathtt{03}_x & \mathtt{01}_x & \mathtt{01}_x & \mathtt{02}_x \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} . \tag{3}$$

Again, this is identical to the MixColumns transformation used in the AES block cipher. Figure 3 illustrates MixColumns for LANE-224 and LANE-256.

3.4 AddConstants

The AddConstants transformation adds a 32-bit constant k_i to each column of the state. These constants k_i are generated using a linear feedback shift register (LFSR), which is described in pseudocode in Figure 4.

Which constants are added to the state depends on the full round number r, which is given as a parameter to AddConstants. For LANE-224 and LANE-256, AddConstants is defined as

$$\text{AddConstants}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_7) = x_0 \oplus k_{8r} \,||\, x_1 \oplus k_{8r+1} \,||\, \cdots \,||\, x_7 \oplus k_{8r+7} \ . \tag{4}$$

For LANE-384 and LANE-512, AddConstants is similarly defined as

$$\text{AddConstants}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{15}) = x_0 \oplus k_{16r} \,||\, x_1 \oplus k_{16r+1} \,||\, \cdots \,||\, x_{15} \oplus k_{16r+15} \ . \tag{5}$$

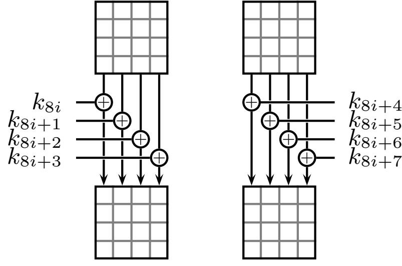

Fig. 5. The AddConstants transformation in LANE-224 and LANE-256.

Figure 5 shows the AddConstants transformation for LANE-224 and LANE-256.

Note that the constants can either be stored in a lookup table, or generated on-the-fly. The latter avoids the need for large tables of constants in implementations where memory is limited. A linear feedback shift register (LFSR) is a natural choice for generating constants. It is simple, and can be implemented using only very limited resources.

3.5 AddCounter

The AddCounter transformation adds part of the counter to the state. The 64-bit counter C is split into two 32-bit words c_0 and c_1, where c_0 is the most significant and c_1 the least significant word, *i.e.*, following the big endian convention.

Depending on the round parameter r, AddCounter adds one of these words to the fourth column of the first AES state. More formally, for LANE-224 and LANE-256 it is given by

$$\text{AddCounter}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \,||\, \cdots \,||\, x_7) = \\ x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \oplus c_{r \bmod 2} \,||\, \cdots \,||\, x_7 \ . \quad (6)$$

Figure 6 shows the AddCounter transformation for LANE-224 and LANE-256. For LANE-384 and LANE-512, AddCounter is defined by

$$\text{AddCounter}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \,||\, \cdots \,||\, x_{15}) = \\ x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \oplus c_{r \bmod 2} \,||\, \cdots \,||\, x_{15} \ . \quad (7)$$

3.6 SwapColumns

The SwapColumns transformation takes a LANE state, and reorders the columns. It ensures that the AES states that comprise the LANE state are mixed among themselves. For LANE-224 and LANE-256 it is given by

$$\text{SwapColumns}\,(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_7) = x_0 \,||\, x_1 \,||\, x_4 \,||\, x_5 \,||\, x_2 \,||\, x_3 \,||\, x_6 \,||\, x_7 \ . \quad (8)$$

Figure 7 shows the SwapColumns transformation for LANE-224 and LANE-256. It can be viewed as a matrix transposition of a 2×2 matrix, where the elements are

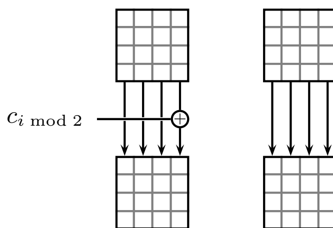

Fig. 6. The AddCounter transformation in LANE-224 and LANE-256.

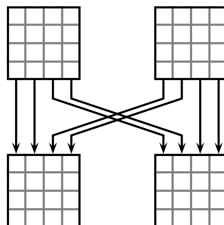

Fig. 7. The SwapColumns transformation in LANE-224 and LANE-256.

formed by pairs of state columns. For LANE-384 and LANE-512, SwapColumns is defined by

$$\begin{aligned} &\text{SwapColumns}\,(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{15}) = \\ &x_0 \,||\, x_4 \,||\, x_8 \,||\, x_{12} \,||\, x_1 \,||\, x_5 \,||\, x_9 \,||\, x_{13} \,||\, x_2 \,||\, x_6 \,||\, x_{10} \,||\, x_{14} \,||\, x_3 \,||\, x_7 \,||\, x_{11} \,||\, x_{15} \enspace . \end{aligned} \tag{9}$$

Figure 8 shows the SwapColumns transformation for LANE-384 and LANE-512. Similar to LANE-256 and LANE-224, SwapColumns can be seen as a matrix transposition, now of a 4×4 matrix, where the elements are the columns of the state.

4 Preprocessing

Before hashing a message using LANE, two preprocessing steps are carried out: message padding, and setting the initial chaining value.

4.1 Message padding

LANE processes a message in blocks of a fixed size, the blocksize. For LANE-224 and LANE-256, the blocksize is 512 bits, and for LANE-384 and LANE-512, the blocksize is 1024 bits. To support any message length up to $2^{64} - 1$ bits (included), zero bits are appended to the message until its length is an integer multiple of the blocksize. This ensures that the padded message can be split into an integer number of blocks of b bits. Note that, if the message length l already is an integer multiple of the blocksize b, no padding bits are added.

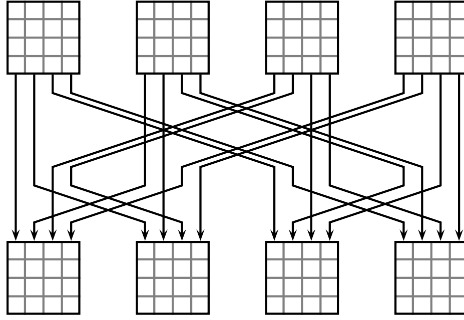

Fig. 8. The SwapColumns transformation in LANE-384 and LANE-512.

Table 1. Parameters of the LANE hash functions.

	LANE-224	LANE-256	LANE-384	LANE-512
Digest length n	224 bits	256 bits	384 bits	512 bits
Blocksize b	512 bits	512 bits	1024 bits	1024 bits
Size of chaining value	256 bits	256 bits	512 bits	512 bits
Salt length $\|S\|$	256 bits	256 bits	512 bits	512 bits

This message padding rule is simpler than the one used in plain Merkle-Damgård, as used in the SHA family of hash functions [10]. As LANE uses an extra compression function call as an output transformation, see Sect. 6, it is natural to include the message length, *i.e.*, the Merkle-Damgård strengthening, in this extra block.

4.2 Setting the initial chaining value

Every digest size supported by LANE uses a different initial value $IV_{n,S}$, which also depends on the (optional) salt. These are defined using the LANE compression function $f(H, M, C)$ itself, which will be defined in detail in Sect. 5.

Let n be the digest size in bits, *i.e.*, n is 224, 256, 384 or 512. Let S be the salt value, or zero if no salt is used. The initial value $IV_{n,S}$ is then given by

$$IV_n = f\left(0, \phi \,||\, \mathrm{bin}_{32}(n) \,||\, 0^\star \,||\, S, 0\right) \quad . \tag{10}$$

Here, $\mathrm{bin}_{32}(n)$ is the digest length n in bits, represented as a 32-bit big-endian integer. The flag byte ϕ indicates whether or not a salt value is used; see Table 2. Note that, if no salt is used, the inital values can be precomputed for each digest size. Note that generalisations of LANE to other digest lengths can be defined in a similar way, if desired.

The purpose of the flag byte ϕ is to provide domain separation. More specifically, the only compression function calls in LANE that use a zero counter C occur in the derivation of the initial chaining value and in the output transformation. Hence, it is impossible to simulate these calls using a normal message block. In order to provide a similar separation for the four cases that do have

Table 2. The flag byte ϕ.

	No salt used	Salt used
Output transformation	$\texttt{00}_x$	$\texttt{01}_x$
Derivation of $IV_{n,S}$	$\texttt{02}_x$	$\texttt{03}_x$

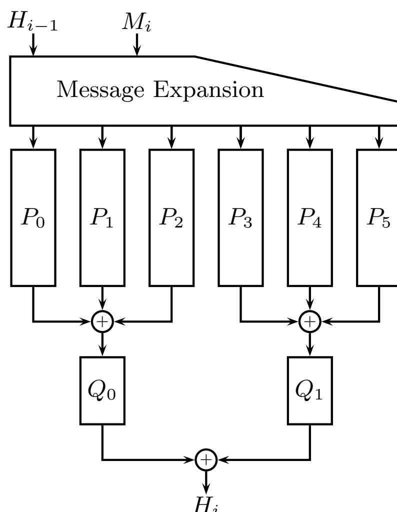

Fig. 9. The LANE compression function.

a zero counter C, *i.e.*, initial value derivation with or without salt, and output transformation with or without salt, the flag byte ϕ is used.

5 The LANE compression function

This section describes the LANE compression function $f(H_{i-1}, M_i, C_i)$. This function takes the following three inputs:

- The input chaining value H_{i-1} is equal to the output of the previous compression function call, or, for the first compression function call, the initial value $IV_{n,S}$.
- The message block M_i holds part of the padded message. Each message block is of a fixed size, the blocksize, which is indicated in Table 1.
- The counter C_i holds the number of message bits hashed so far, including the message bits in the current message block M_i. The counter C_i is represented as a 64-bit unsigned integer in big-endian notation.

The idea of including a bit counter in every compression function call is borrowed from the 'HAsh Iterative FrAmework' (HAIFA) of Biham and Dunkelman [1]. This stops several attacks on the iteration, *e.g.*, the long message second preimage attack by Kelsey and Schneier [7], at only a very modest cost.

The structure of the LANE compression function is shown in Figure 9. It consists of a message expansion, eight permutation *lanes*, arranged in two layers, and three XOR combiners. Section 5.1 describes the message expansion. The permutation *lanes* are discussed in Sect. 5.2. The LANE compression function was designed to be simple to understand and easy to analyse.

The use of permutations ensures that internal collisions can only occur in certain places, *i.e.*, at the XOR combiners. Establishing such an internal collision is equivalent to satisfying a linear condition on the outputs of several permutations. Similarly, the message expansion imposes linear relations on the inputs of the permutations. The rationale is that, while such conditions are very simple, it is hard to maintain or even track them through the rounds of the permutations.

A similar rationale applies to the problem of finding (second) preimages for the compression function. Straightforward inversion attempts fail, as one has to ensure that the linear conditions imposed by the message expansion hold. This is again considered to be very difficult.

As described in detail in [6], having only a single layer of permutations would allow for a class of distinguishers for the compression function, based on limiting the permutation inputs to a small set. The second layer of permutations not only prevents that, but also has a beneficial effect on the resistance to differential cryptanalysis. Indeed, in a collision differential, either the entire second layer must be activated, or an internal collision must be reached simultaneously on both of the XOR combiners after the first layer, *i.e.*, on a value twice the size of the chaining value.

The ample parallelism provided by the LANE compression function allows for flexibility in implementation. In software implementations, LANE offers many opportunities for instruction level parallelism (ILP), which can be used by modern pipelined and superscalar CPU's. Also, as the same operations are carried out on many independent data values in parallel, it is possible to use vector instructions, *i.e.*, Single Instruction Multiple Data (SIMD) instructions. On the other end of the spectrum, it is equally possible to implement LANE in a completely serial way. In such implementations, the memory requirements are kept minimal. Hardware designers implementing LANE are offered an area-speed tradeoff, making LANE suitable for both resource-constrained and very high-speed applications.

5.1 The message expansion

The message expansion of LANE takes the message block M_i and the input chaining value H_{i-1}, and expands them into six expanded message blocks, $W_0, \ldots, W_5$.

In LANE-224 and LANE-256, the six expanded message words, $W_0, \ldots, W_5$, are all 256 bits long. They are computed as follows. Split the 512-bit message block M_i into four 128-bit parts $m_0, \ldots, m_3$, and split the 256-bit input chaining value H_{i-1} into two 128-bit parts, h_0 and h_1:

$$\begin{aligned} m_0 \,||\, m_1 \,||\, m_2 \,||\, m_3 &\leftarrow M_i \\ h_0 \,||\, h_1 &\leftarrow H_{i-1} \end{aligned} \quad . \tag{11}$$

Then, compute the six expanded message words, $W_0, \ldots, W_5$ as

$$\begin{array}{ll} W_0 = h_0 \oplus m_0 \oplus m_1 \oplus m_2 \oplus m_3 & || \; h_1 \oplus m_0 \oplus m_2 \\ W_1 = h_0 \oplus h_1 \oplus m_0 \oplus m_2 \oplus m_3 & || \; h_0 \oplus m_1 \oplus m_2 \\ W_2 = h_0 \oplus h_1 \oplus m_0 \oplus m_1 \oplus m_2 & || \; h_0 \oplus m_0 \oplus m_3 \\ W_3 = h_0 & || \; h_1 \\ W_4 = m_0 & || \; m_1 \\ W_5 = m_2 & || \; m_3 \end{array} . \qquad (12)$$

The message expansion in LANE-384 and LANE-512 is completely analogous. The only difference is that all sizes are doubled.

Even more so than other components of LANE, the message expansion was chosen to be very simple and light. Its main purpose is to introduce dependencies between the inputs of the various permutation lanes, such that they cannot be chosen independently. It also precludes straightforward inversion attempts, as it is conjectured that, however simple the linear conditions imposed by the message expansion, it is not feasible to satisfy them when only having direct control over the permutation outputs.

The message expansion is based on a (6,3,4) linear code over GF(4). The minimum distance property of this code ensures that, in a differential attack, at least four out of the six lanes in the first layer will be *active*, *i.e.*, have a difference at the input as well as output.

Provable resistance is offered against meet-in-the-middle preimage attacks on the compression function. Indeed, it is not possible to construct two independent sets of permutation lanes to use in such an attack. This follows from the minimum distance property of the linear code on which the message expansion is based.

Each output of the message expansion can be computed independently of the others, and read-only access to the current message block suffices. This implies that the message buffer can be shared with another application, eliminating the need for extra memory and costly data copying.

Finally, note that the inputs of the permutation lanes P_4 and P_5 only depend on the message block input, and not on the chaining value. This implies that those lanes can already be computed while the previous chaining value is not yet known, *e.g.*, in parallel with the second layer of the previous compression function call.

5.2 The permutations

The LANE compression function contains eight permutations, arranged in two layers. Each permutation consists of a number of rounds, where the number of rounds is different for the two layers: the permutations in the first layer have twice as many rounds as those in the second layer. In the rest of the document, we use "lane" as a synonym for a single LANE permutation. Table 3 gives the number of rounds in the permutations for each LANE variant. The rationale behind the choice of the number of permutation rounds is to use as few rounds as possible, for performance reasons, but still enough rounds to offer an adequate security margin. We refer to [6] for a more detailed discussion.

Table 3. Number of rounds in the LANE permutations.

	LANE-224	LANE-256	LANE-384	LANE-512
$P_0, \ldots, P_5$	6	6	8	8
Q_0, Q_1	3	3	4	4

```
function Round(r, X)
1: X ← SubBytes(X)
2: X ← ShiftRows(X)
3: X ← MixColumns(X)
4: X ← AddConstants(r, X)
5: X ← AddCounter(r, X)
6: X ← SwapColumns(X)
7: return X
```

```
function LastRound(X)
1: X ← SubBytes(X)
2: X ← ShiftRows(X)
3: X ← MixColumns(X)
4: X ← SwapColumns(X)
5: return X
```

Fig. 10. Pseudocode for the LANE permutation rounds.

The rounds of the permutations use the building blocks described in Sect. 3. More in detail, a full permutation round consists of the following sequence of transformations: SubBytes, ShiftRows, MixColumns, AddConstants, AddCounter and SwapColumns. The last round of each permutation omits AddConstants and AddCounter. Figure 10 gives a pseudocode description of the LANE permutation rounds.

Note that a permutation round can be seen as two, for LANE-224 and LANE-256, or four, for LANE-384 and LANE-512, parallel invocations of a round of the AES block cipher [3,9], where the appropriate constants and counter word are used as a round key, followed by SwapColumns.

A round number r is assigned to each of the full rounds across all permutations, to specify the constants and counter to use in each round. The permutations are taken in the order P_0, P_1, ..., P_5, Q_0, Q_1 and only the full rounds are counted, *i.e.*, the last round of each permutation is ignored. Table 4 lists the round numbers r in each of the permutations.

A pseudocode description of the permutations used in LANE-224 and LANE-256 is given in Figure 11, including an exact expression to compute the full round number r for each round. Figure 12 describes the permutations used in LANE-384 and LANE-512.

The permutations used in LANE are built using components of the AES block cipher [3,9]. One motivation for this choice is that these components and their properties are well studied and hence well understood. This allows to build on existing work on the security of these components to analyse LANE.

Reusing AES components also has several practical benefits. Much effort has already been spent on efficient implementations of the AES on a wide variety of platforms. Since LANE is based on the AES, these techniques can equally be applied to LANE. For example, the new AES-NI instruction set, which was announced by Intel [2] and will be introduced in the next generation of Intel processors as of 2009, can be used to accelerate LANE. Another benefit lies in resource constrained environments, requiring both a hash function and a block

Table 4. The full round number r.

	LANE-224	LANE-256	LANE-384	LANE-512
P_0	0—4	0—4	0—6	0—6
P_1	5—9	5—9	7—13	7—13
P_2	10—14	10—14	14—20	14—20
P_3	15—19	15—19	21—27	21—27
P_4	20—24	20—24	28—34	28—34
P_5	25—29	25—29	35—41	35—41
Q_0	30—31	30—31	42—44	42—44
Q_1	32—33	32—33	45—47	45—47

function $P_j(X)$
1: **for** $i = 0$ to 4 **do**
2: $r \leftarrow 5j + i$
3: $X \leftarrow \text{Round}\,(r, X)$
4: **end for**
5: $X \leftarrow \text{LastRound}(X)$
6: **return** X

function $Q_j(X)$
1: **for** $i = 0$ to 1 **do**
2: $r \leftarrow 30 + 2j + i$
3: $X \leftarrow \text{Round}\,(r, X)$
4: **end for**
5: $X \leftarrow \text{LastRound}(X)$
6: **return** X

Fig. 11. Pseudocode for the permutations in LANE-224 and LANE-256.

function $P_j(X)$
1: **for** $i = 0$ to 6 **do**
2: $r \leftarrow 7j + i$
3: $X \leftarrow \text{Round}\,(r, X)$
4: **end for**
5: $X \leftarrow \text{LastRound}(X)$
6: **return** X

function $Q_j(X)$
1: **for** $i = 0$ to 2 **do**
2: $r \leftarrow 42 + 3j + i$
3: $X \leftarrow \text{Round}\,(r, X)$
4: **end for**
5: $X \leftarrow \text{LastRound}(X)$
6: **return** X

Fig. 12. Pseudocode for the permutations in LANE-384 and LANE-512.

cipher. Using Lane together with the AES allows large parts of the implementation to be shared, yielding a substantial overall improvement.

For simplicity and ease of (parallel) implementation, all permutations in Lane are built in the same way. Different constants are thus required in each permutation lane, to ensure that any attack based on maintaining symmetry across several permutation rounds is avoided. The first constant, used to initialise the LFSR which generates the other constants, was chosen such that no two constant bytes used in the same position of two different lanes are equal.

The permutations are keyed using the bit counter input to the compression function. This is a natural way of including the bit counter, as it is very simple and lightweight, but achieves the goal of making the whole compression function dependent on this counter.

6 The output transformation

The output transformation of Lane takes as input the chaining value after all padded message blocks have been processed, and returns the message digest. It also includes the message length l, and the (optional) salt S, if one was used.

The transformation consists of two parts. First, a single additional compression function call is done. The counter C is set to zero, and the message input is set to

$$\phi \,||\, \mathrm{bin}_{64}(l) \,||\, 0^{\star} \,||\, S \ . \tag{13}$$

Here, $\mathrm{bin}_{64}(l)$ is the (unpadded) message length l in bits, represented as a 64-bit big-endian integer. The flag byte ϕ indicates whether or not a salt value is used; see Table 2. If a salt value is not used, zero bits are used instead of S.

In the second part of the output transformation, a truncation is applied to compute the final message digest. This truncation keeps the n leftmost bits and truncates away the other bits, where n is the digest length. For Lane-256 and Lane-512, no truncation is required. The truncation operation for Lane-224 is given by

$$\mathrm{Trunc}_{224}\,(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_6 \,||\, x_7) = x_0 \,||\, x_1 \,||\, \cdots \,||\, x_6 \ . \tag{14}$$

For Lane-384, the truncation is defined similarly as

$$\mathrm{Trunc}_{384}\,(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{11} \,||\, \cdots \,||\, x_{15}) = x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{11} \ . \tag{15}$$

Note that generalisations of Lane to other digest lengths can be defined using a similar truncation, if desired.

The output transformation is used to offer an additional layer of protection against (first) preimage attacks. For simplicity, this output transformation is constructed based on the Lane compression function, with a message block of a fixed structure. It is straightforward to see that this structure imposed on the message block drastically limits the freedom of an adversary seeking a preimage. The output transformation also serves to protect against length-extension attacks, as it is impossible to simulate the effect of the output transformation using a regular message block. Finally, the output transformation also offers additional protection against distinguishing attacks, as any potential bias in the compression function

Acknowledgements

Elena Andreeva, Sebastiaan Indesteege, Elmar Tischhauser ('Aspirant F.W.O. Vlaanderen') and Christophe De Cannière ('Postdoctoraal Onderzoeker F.W.O. Vlaanderen') are supported by the Fund for Scientific Research Flanders (F.W.O. Vlaanderen). This work was also supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

References

1. Eli Biham and Orr Dunkelman. A framework for iterative hash functions — HAIFA. Presented at the second NIST hash workshop (August 24–25), 2006.
2. Intel Corporation. Advanced encryption standard (AES) instructions set. White paper, July 2008. Available online at `http://softwarecommunity.intel.com/isn/downloads/intelavx/AES-Instructions-Set_WP.pdf`.
3. Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.
4. Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1989.
5. Shai Halevi and Hugo Krawczyk. Strengthening digital signatures via randomized hashing. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2006.
6. Sebastiaan Indesteege, Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, and Elmar Tischhauser. The LANE hash function. Submission to NIST, 2008. Available online at `http://www.cosic.esat.kuleuven.be/lane/`.
7. John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.
8. Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.
9. National Institue of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. Available online at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.
10. National Institue of Standards and Technology. Secure hash standard. Federal Information Processing Standards Publication 180-2, 2002. Available online at `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`.
11. National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, November 2007.