The LANE hash function

Department of Electrical Engineering ESAT/SCD-COSIC,
Katholieke Universiteit Leuven.
Kasteelpark Arenberg 10-2446, B-3001 Heverlee, Belgium.
{sebastiaan.indesteege,bart.preneel}@esat.kuleuven.be

Designer: Sebastiaan Indesteege

Submitters: Sebastiaan Indesteege, Bart Preneel

Contributors: Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, Elmar Tischhauser

SHA-3 Proposal

Acknowledgements

I would like to thank Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, Vincent Rijmen and Elmar Tischhauser for many interesting discussions concerning the design of Lane and its predecessors, and for their continued effort on the cryptanalysis of both older and the final version of Lane. Their findings, comments and suggestions for improvements were invaluable in the design process.

I extend my gratitude to Antoon Bosselaers, Emilia Käsper, Miroslav Knežević, Nicky Mouha and Vesselin Velichkov for their work on several implementations of Lane, and for giving useful feedback from the implementor's point of view.

Additional thanks go to all of the people mentioned above, for their contributions to the writing and proofreading of this document. Finally, thanks to everyone in the COSIC research group for their support.

Sebastiaan Indesteege
October 2008

Sebastiaan Indesteege is supported by the Fund for Scientific Research Flanders (Aspirant 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 Interdisciplinary Institute for BroadBand Technology (IBBT), and in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

Contents

1 Introduction **1**

2 Specification **3**
2.1 Introduction . . . 3
2.2 Preliminaries . . . 4
2.2.1 Bit strings, bytes and states . . . 4
2.2.2 The finite field GF(2^8) . . . 4
2.3 Building blocks . . . 6
2.3.1 SubBytes . . . 6
2.3.2 ShiftRows . . . 6
2.3.3 MixColumns . . . 8
2.3.4 AddConstants . . . 8
2.3.5 AddCounter . . . 10
2.3.6 SwapColumns . . . 10
2.4 Preprocessing . . . 12
2.4.1 Message padding . . . 12
2.4.2 Setting the initial chaining value . . . 12
2.5 The Lane compression function . . . 14
2.5.1 The message expansion . . . 15
2.5.2 The permutations . . . 15
2.6 The output transformation . . . 16

3 Design rationale **19**
3.1 The iteration mode . . . 19
3.1.1 The message padding . . . 19
3.1.2 The use of a counter . . . 20
3.1.3 The output transformation . . . 20
3.1.4 The use of a salt value . . . 20
3.1.5 A parallel iteration mode . . . 21
3.2 The compression function . . . 22
3.2.1 The message expansion . . . 23
3.2.2 The permutations . . . 23
3.2.3 The constants . . . 25
3.3 Advantages and limitations of Lane . . . 25
3.3.1 Advantages . . . 25
3.3.2 Limitations . . . 26

4 Security analysis **27**
4.1 Reduced versions of Lane for cryptanalysis 27
4.2 Standard differential cryptanalysis 28
4.2.1 Active lanes in the first layer 28
4.2.2 Active S-boxes per lane 30
4.2.3 Breaking reduced versions 30
4.2.4 Maximum probability of a trail 32
4.3 Truncated differential cryptanalysis 32
4.3.1 Truncated differentials 32
4.3.2 Identifying the optimal truncated differential 33
4.3.3 Using truncated differentials for collision searching 35
4.4 Higher order differential cryptanalysis 36
4.4.1 A fourth order differential distinguisher 36
4.4.2 Square attacks on the compression function 37
4.4.3 Multiset distinguishers 38
4.5 Cryptanalysis of wide-block Rijndael 38
4.6 Algebraic attacks 39
4.7 Attacks based on reduced query complexity 39
4.7.1 General comments 39
4.7.2 Results on Lane 40
4.7.3 Bounds for query complexity 41
4.8 Wagner's generalised birthday attack 43
4.9 Meet-in-the-middle attacks 43
4.10 Long message second-preimage attacks 44
4.11 Length-extension attacks 45
4.12 Multicollision attacks 45
4.13 On the mode of operation 46
4.14 Expected strength of Lane 47

5 Implementation aspects **49**
5.1 General purpose CPU's 49
5.1.1 Bitsliced implementation 50
5.1.2 Intel AES-NI instruction set 52
5.2 Embedded systems with an 8-bit CPU 53
5.3 Hardware implementation 54

A The constants used in Lane **63**

List of Figures

2.1 The SubBytes transformation in Lane-224 and Lane-256. 7
2.2 The ShiftRows transformation in Lane-224 and Lane-256. . . . 9
2.3 The MixColumns transformation in Lane-224 and Lane-256. . . 9
2.4 Pseudocode for generating the Lane constants. 9
2.5 The AddConstants transformation in Lane-224 and Lane-256. . 11
2.6 The AddCounter transformation in Lane-224 and Lane-256. . . 11
2.7 The SwapColumns transformation in Lane-224 and Lane-256. . 11
2.8 The SwapColumns transformation in Lane-384 and Lane-512. . 13
2.9 The Lane compression function. 14
2.10 Pseudocode for the Lane permutation rounds. 17
2.11 Pseudocode for the permutations in Lane-224 and Lane-256. . . 17
2.12 Pseudocode for the permutations in Lane-384 and Lane-512. . . 17

4.1 A collision differential for Lane. 31
4.2 Truncated differentials in one lane of Lane-256 33
4.3 Truncated differentials in one lane of Lane-512 34

List of Tables

2.1 Parameters of the LANE hash functions. 5
2.2 The notation used in the specification of LANE. 5
2.3 The AES S-box, in hexadecimal format. 7
2.4 The flag byte ϕ. 13
2.5 The LANE initial values IV_n, if no salt is used. 13
2.6 Number of rounds in the LANE permutations. 17
2.7 The full round number r. 17

4.1 Lower bounds on the number of active S-boxes 30
4.2 Expected strength of LANE against cryptanalytic attacks. 48

5.1 Test platform for the software implementations of LANE. 50
5.2 Performance measurement results of our LANE implementations. 51
5.3 Number of XMM instructions in one LANE round. 52
5.4 Hardware evaluation of the LANE hash function. 54

A.1 The constants used in LANE 63

Chapter 1

Introduction

In this document, we propose the cryptographic hash function Lane as a candidate for the SHA-3 competition organised by NIST [50]. Lane is an iterated hash function supporting multiple digest sizes. Components of the AES block cipher [48, 21] are reused as building blocks. Lane aims to be secure, easy to understand, elegant and flexible in implementation.

The structure of this document is as follows. In Chapter 2, we give a full specification of the Lane hash function. The design rationale, including motivations for all important design choices, is discussed in Chapter 3. Chapter 4 contains an extensive security analysis, investigating the resistance of Lane against a variety of attacks. Finally, implementation aspects of Lane form the subject of Chapter 5.

Chapter 2

Specification

2.1 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 [48, 21]. After introducing some preliminaries and conventions in Sect. 2.2, the building blocks are described in Sect. 2.3.

Optionally, a salt value S, can be used while computing the digest. When used, the size of this salt is 256 bits for LANE-224 and LANE-256, and 512 bits for LANE-384 and LANE-512. Refer to Table 2.1 for a comparison of the parameters of the various LANE variants.

Hashing a message is performed in three steps. In the first step, which is described in Sect. 2.4, 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:

$$H_i = f\left(H_{i-1}, M_i, C_i\right) \quad . \tag{2.1}$$

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. 2.5.

The third and final step is the output transformation, described in Sect. 2.6. 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.

Note that LANE supports hashing in 'one-pass' streaming mode. There is no need to buffer the entire message, and one can start hashing as soon as the first complete message block has been received. This property is similar to the hash functions of the SHA-family [49].

2.2 Preliminaries

This section introduces the preliminaries and conventions that will be used in this specification. For reference, the notations used in this chapter are summarised in Table 2.2.

2.2.1 Bit strings, bytes and states

Definition 1. A *bit string* is an ordered sequence of binary digits of arbitrary length. A bit string is written from left to right, *i.e.*, the leftmost bit is the first bit of the sequence.

Definition 2. A *byte* is a bit string consisting of eight bits. A byte can represent an integer in the range from 0 to $2^8 - 1$. The big-endian convention is used, *i.e.*, the first (leftmost) bit of a byte is the most significant bit.

Definition 3. An *AES state* is a 4×4 array of bytes, corresponding to an internal state of the AES block cipher [48, 21]. A sequence of 16 bytes can be mapped to an AES state, and vice versa. The sequence of 16 bytes $y_0 \,||\, \cdots \,||\, y_{15}$ is mapped to the AES state

$$\begin{bmatrix} y_0 & y_4 & y_8 & y_{12} \\ y_1 & y_5 & y_9 & y_{13} \\ y_2 & y_6 & y_{10} & y_{14} \\ y_3 & y_7 & y_{11} & y_{15} \end{bmatrix} . \tag{2.2}$$

Definition 4. 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. The sequence is split into 16-byte parts, each of which is mapped to an AES state as described above. The AES states are ordered in the same way as the 16-byte parts in the original byte sequence, *i.e.*, the leftmost AES state contains the first 16 bytes of the sequence.

2.2.2 The finite field GF(2^8)

As LANE is based on components of the AES block cipher, it also uses arithmetic operations in the finite field GF(2^8). Elements of the finite field GF(2^8) can be represented in several ways, but all representations are isomorphic, *i.e.*, they are simply different ways of representing the same finite field with 2^8 elements [40]. In this document, we adopt the same representation as commonly used to describe the AES block cipher [48, 21].

A *byte* is used to represent an element of the finite field GF(2^8). It is useful to view the byte, consisting of bits $b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7$ as a polynomial with coefficients 0 or 1:

$$b_0 X^7 + b_1 X^6 + b_2 X^5 + b_3 X^4 + b_4 X^3 + b_5 X^2 + b_6 X + b_7 \ . \tag{2.3}$$

The *addition* of two elements of GF(2^8), represented as polynomials, is defined as component-wise addition modulo two. On the byte level, this corresponds to exclusive or (XOR). The neutral element with respect to addition is the byte $\texttt{00}_x$, and every element is its own additive inverse.

Table 2.1: 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

Table 2.2: The notation used in the specification of LANE.

$0^\star$	A number of zero bits, required to pad a bit string to a given length.
$\oplus$	Exclusive or (XOR).
$\|\|$	Concatenation of bit strings.
$x \gg i$	Bitwise right-shift of the word x over i bits.
$\text{bin}_{32}(\cdot)$ or $\text{bin}_{64}(\cdot)$	Big-endian representation of a number in 32 or 64 bits, respectively.
$\cdots x$	A number in hexadecimal notation.
ϕ	The flag byte used in the output transformation and the derivation of $IV_{n,S}$.
b	The blocksize.
C_i	Counter indicating the number of message bits (excluding padding) in message blocks 0 up to and including i.
$f(H_{i-1}, M_i, C_i)$	The LANE compression function.
H_i	Chaining value after processing message block i.
$IV_{n,S}$ or IV_n	The initial value for digest length n and salt S (if applicable).
l	Message length in bits.
k_i	A 32-bit LANE constant.
M	A message.
M_i	Padded message block i.
n	Digest length, *i.e.*, 224, 256, 384 or 512 bits.
P_i, Q_i	The permutations (lanes) used in LANE in the first and second layer, respectively.
r	The full round number.
S	Salt value.
W_0,...,W_5	Expanded message blocks.
x_i	A column of an AES state, consisting of four bytes.

The *multiplication* of two elements of GF(2^8) is defined as the the multiplication of polynomials, reduced modulo an irreducible polynomial $m(X)$,

$$m(X) = X^8 + X^4 + X^3 + X + 1 \ . \tag{2.4}$$

The byte $\texttt{01}_x$ is the neutral element with respect to multiplication. Every nonzero byte has a multiplicative inverse, which can be computed using the extended Euclidean algorithm.

2.3 Building blocks

The LANE hash functions reuse several components from the AES block cipher [48, 21]. 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.

2.3.1 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 2.1 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 substitution consists of the composition of the following operations:

1. The inverse operation in the finite field GF(2^8), defined by the irreducible polynomial $m(X)$, given in (2.4). The zero element is mapped to itself.
2. An affine mapping over GF(2), defined by

$$\begin{bmatrix} b'_7 \\ b'_6 \\ b'_5 \\ b'_4 \\ b'_3 \\ b'_2 \\ b'_1 \\ b'_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \ . \tag{2.5}$$

Here, $b_0,\ldots,b_7$ denote the bits of a byte representing an element of GF(2^8), where b_0 is the most significant bit. This is the same S-box as the one used in the AES block cipher [48, 21]. It is given in Table 2.3.

2.3.2 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 2.2 illustrates ShiftRows for LANE-224 and LANE-256.

	␣0	␣1	␣2	␣3	␣4	␣5	␣6	␣7	␣8	␣9	␣a	␣b	␣c	␣d	␣e	␣f
0␣	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1␣	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2␣	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3␣	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4␣	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5␣	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6␣	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7␣	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8␣	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9␣	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a␣	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b␣	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c␣	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d␣	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e␣	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f␣	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table 2.3: The AES S-box, in hexadecimal format.

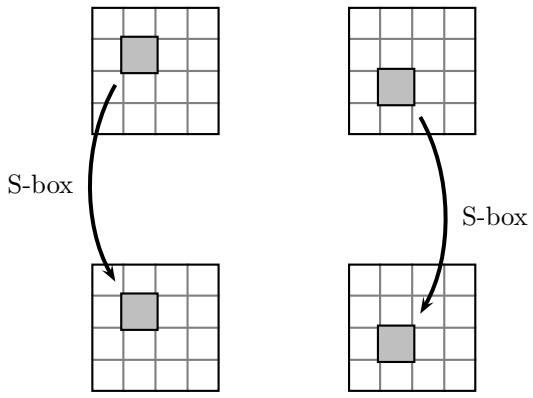

Figure 2.1: The SubBytes transformation in LANE-224 and LANE-256.

2.3.3 MixColumns

The MixColumns transformation operates on the columns of the state. Each column is viewed as a polynomial over GF(2^8), *i.e.*, a polynomial of degree three with coefficients in GF(2^8):

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \quad \leftrightarrow \quad y_3 \cdot Y^3 + y_2 \cdot Y^2 + y_1 \cdot Y + y_0 \ . \tag{2.6}$$

Then, this polynomial is multiplied modulo $Y^4 + 1$ with the fixed polynomial $c(Y)$,

$$c(Y) = 03Y^3 + 01Y^2 + 01Y + 02 \ . \tag{2.7}$$

Even though Y^4+1 is not an irreducible polynomial over GF(2^8), implying that multiplication with a fixed polynomial is not necessarily invertible, the polynomial $c(Y)$ is such that MixColumns is an invertible operation. Equivalently, this operation can be written as a matrix multiplication

$$\begin{bmatrix} y'_0 \\ y'_1 \\ y'_2 \\ y'_3 \end{bmatrix} = \begin{bmatrix} \mathtt{02}_x & \mathtt{03}_x & \mathtt{01}_x & \mathtt{01}_x \\ \mathtt{01}_x & \mathtt{02}_x & \mathtt{03}_x & \mathtt{01}_x \\ \mathtt{01}_x & \mathtt{01}_x & \mathtt{02}_x & \mathtt{03}_x \\ \mathtt{03}_x & \mathtt{01}_x & \mathtt{01}_x & \mathtt{02}_x \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \ . \tag{2.8}$$

Again, this is identical to the MixColumns transformation used in the AES block cipher. Figure 2.3 illustrates MixColumns for Lane-224 and Lane-256.

2.3.4 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 2.4. Table A.1 in Appendix A contains the values of the constants used in Lane, found using this algorithm.

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

$$\begin{aligned} &\text{AddConstants}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_7) = \\ &\qquad\qquad x_0 \oplus k_{8r} \,||\, x_1 \oplus k_{8r+1} \,||\, \cdots \,||\, x_7 \oplus k_{8r+7} \ . \end{aligned} \tag{2.9}$$

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

$$\begin{aligned} &\text{AddConstants}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{15}) = \\ &\qquad\qquad x_0 \oplus k_{16r} \,||\, x_1 \oplus k_{16r+1} \,||\, \cdots \,||\, x_{15} \oplus k_{16r+15} \ . \end{aligned} \tag{2.10}$$

Figure 2.5 shows the AddConstants transformation for Lane-224 and Lane-256.

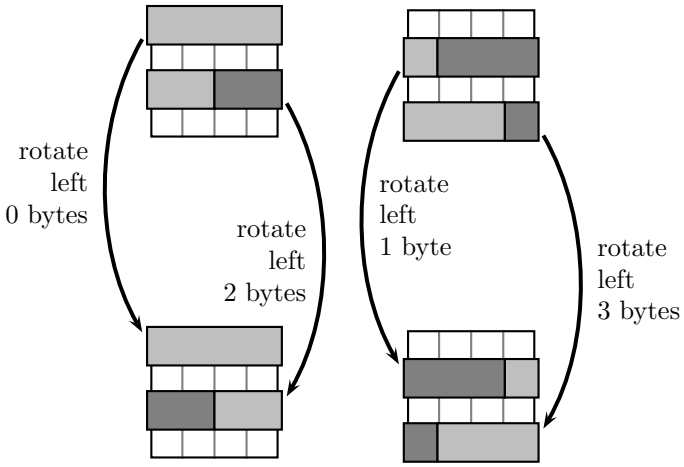

Figure 2.2: The ShiftRows transformation in LANE-224 and LANE-256.

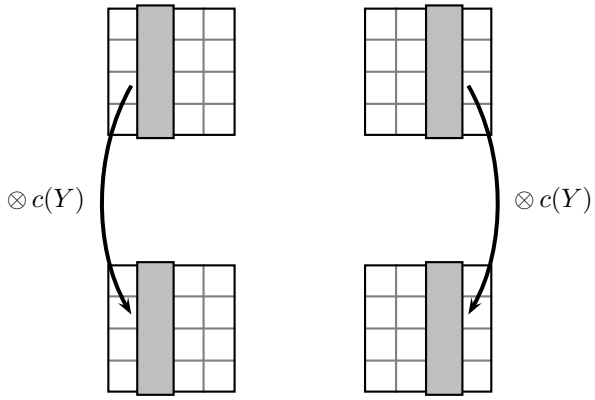

Figure 2.3: The MixColumns transformation in LANE-224 and LANE-256.

1: $k_0 \leftarrow \texttt{07fc703d}_x$
2: **for** $i = 1$ to 272 (resp. 768 for LANE-384 and LANE-512) **do**
3: $\quad k_i = k_{i-1} \gg 1$
4: $\quad$ **if** $k_{i-1} \wedge \texttt{00000001}_x$ **then**
5: $\quad\quad k_i = k_i \oplus \texttt{d0000001}_x$
6: $\quad$ **end if**
7: **end for**

Figure 2.4: Pseudocode for generating the LANE constants.

2.3.5 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

$$\begin{aligned}\text{AddCounter}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \,||\, \cdots \,||\, x_7) = \\ x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \oplus c_{r \bmod 2} \,||\, \cdots \,||\, x_7 \ . \end{aligned} \quad (2.11)$$

Figure 2.6 shows the AddCounter transformation for LANE-224 and LANE-256. For LANE-384 and LANE-512, AddCounter is defined by

$$\begin{aligned}\text{AddCounter}\,(r, x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \,||\, \cdots \,||\, x_{15}) = \\ x_0 \,||\, x_1 \,||\, \cdots \,||\, x_3 \oplus c_{r \bmod 2} \,||\, \cdots \,||\, x_{15} \ . \end{aligned} \quad (2.12)$$

2.3.6 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

$$\begin{aligned}\text{SwapColumns}\,(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_7) = \\ x_0 \,||\, x_1 \,||\, x_4 \,||\, x_5 \,||\, x_2 \,||\, x_3 \,||\, x_6 \,||\, x_7 \ . \end{aligned} \quad (2.13)$$

Figure 2.7 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 formed by pairs of state columns. For LANE-384 and LANE-512, SwapColumns is defined by

$$\begin{aligned}&\text{SwapColumns}\,(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{15}) = \\ &x_0 \,||\, x_4 \,||\, x_8 \,||\, x_{12} \,||\, x_1 \,||\, x_5 \,||\, x_9 \,||\, x_{13} \,||\, x_2 \,||\, x_6 \,||\, x_{10} \,||\, x_{14} \,||\, x_3 \,||\, x_7 \,||\, x_{11} \,||\, x_{15} \ . \end{aligned} \quad (2.14)$$

Figure 2.8 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.

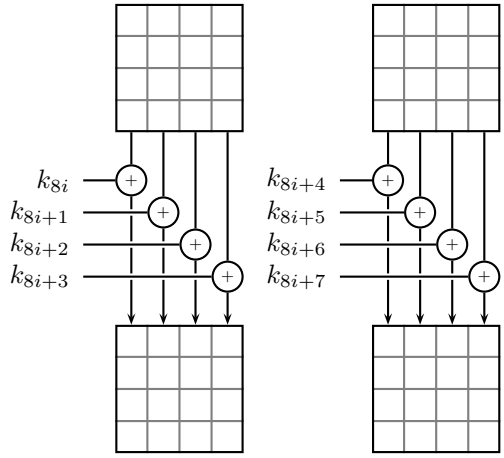

Figure 2.5: The AddConstants transformation in Lane-224 and Lane-256.

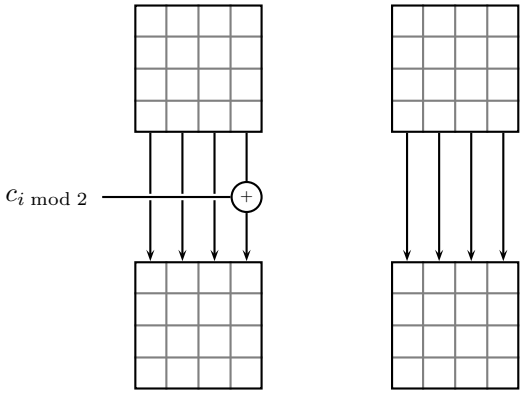

Figure 2.6: The AddCounter transformation in Lane-224 and Lane-256.

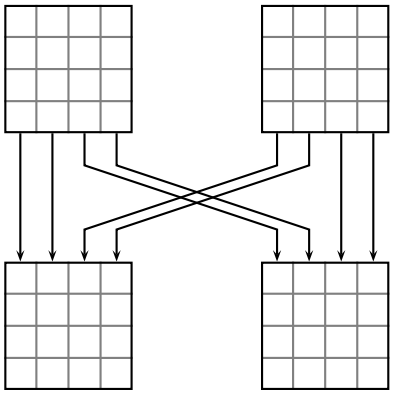

Figure 2.7: The SwapColumns transformation in Lane-224 and Lane-256.

2.4 Preprocessing

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

2.4.1 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.

More formally, a message M of length l is padded as follows. Let b be the blocksize and let κ be the smallest positive integer, $0 \leq \kappa < b$, such that

$$l + \kappa \equiv 0 \pmod{b} \ . \tag{2.15}$$

Now, the padded message is computed as

$$\mathrm{pad}\,(M) = M \,||\, 0^{\kappa} \ . \tag{2.16}$$

This padding rule 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. The empty string is a particular example of this; no padding bits are added to it. Thus, when hashing the empty string, no message blocks are to be processed, and one proceeds immediately with the output transformation.

2.4.2 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. 2.5.

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 the output of the following compression function call using a zero input chaining value and a zero counter value:

$$IV_n = f\,(0, \phi \,||\, \mathrm{bin}_{32}(n) \,||\, 0^{\star} \,||\, S, 0) \ . \tag{2.17}$$

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 2.4. If a salt value is not used, $\phi = \mathtt{02}_x$ and the salt S is filled with zero bits. If a salt value is used, $\phi = \mathtt{03}_x$. The size of the salt S is 256 bits for LANE-224 and LANE-256, and 512 bits for LANE-384 and LANE-512, as indicated in Table 2.1. Note that generalisations of LANE to other digest lengths can be defined in a similar way, if desired.

If no salt is being used, the initial values can also be precomputed for each digest size. Table 2.5 lists the initial values for supported digest sizes.

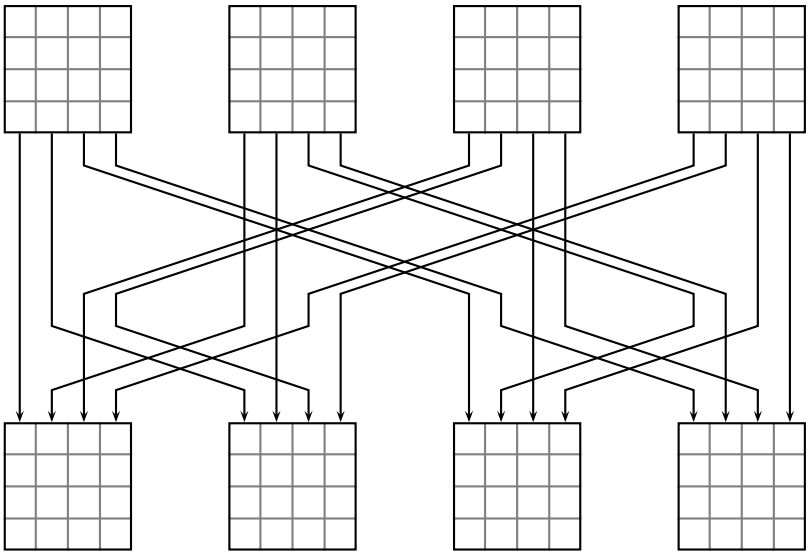

Figure 2.8: The SwapColumns transformation in LANE-384 and LANE-512.

Table 2.4: The flag byte ϕ.

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

Table 2.5: The LANE initial values IV_n, in big-endian notation, if no salt is used.

LANE-224	$c8245a868d733102314ddcb9f60a7ef4_x$ $57b8c917eefeaec2ff4fc3be87c4728e_x$
LANE-256	$be292e17bb541ff2fe54b6f730b1c96a_x$ $7b2592688539bdf397c4bdd649763fb8_x$
LANE-384	$148922ce548c300176978bc8266e008c_x$ $3dc60765d85b09d94cb1c8d8e2cab952_x$ $db72be8e685f0783fa436c3d4b9acb90_x$ $5088dd47932f55a9a0c415c6db6dd795_x$
LANE-512	$9b6034811d5a931b69c4e6e0975e2681_x$ $b863ba538d1be11b77340080d42c48a5_x$ $3a3a1d611cf3a1c4f0a303477e56a44a_x$ $9530ee60dadb05b63ae3ac7cd732ac6a_x$

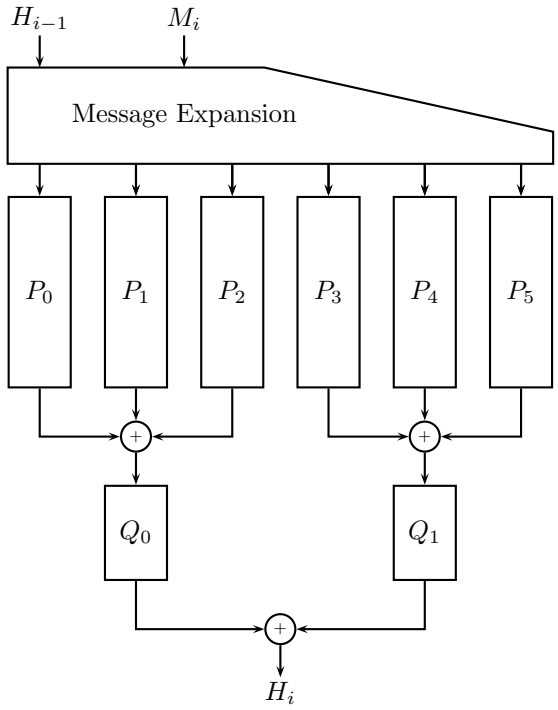

Figure 2.9: The LANE compression function.

2.5 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 $IV_{n,S}$. In LANE-224 and LANE-256, the size of the chaining value is 256 bits. In LANE-384 and LANE-512, a 512-bit chaining value is used.

- The message block M_i holds part of the padded message. Each message block is of a fixed size, the blocksize, which is also indicated in Table 2.1. In LANE-224 and LANE-256, the blocksize is 512 bits. In LANE-384 and LANE-512, the blocksize is 1024 bits.

- 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 structure of the LANE compression function is shown in Figure 2.9. It consists of a message expansion, eight permutation *lanes*, arranged in two layers, and three XOR combiners. Section 2.5.1 describes the message expansion. The permutation *lanes* are discussed in Sect. 2.5.2.

2.5.1 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, $m_0, \ldots, m_3$:

$$m_0 \,||\, m_1 \,||\, m_2 \,||\, m_3 \leftarrow M_i \ . \tag{2.18}$$

Similarly, split the 256-bit input chaining value H_{i-1} into two 128-bit parts, h_0 and h_1:

$$h_0 \,||\, h_1 \leftarrow H_{i-1} \ . \tag{2.19}$$

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

$$\begin{array}{lcll} W_0 & = & h_0 \oplus m_0 \oplus m_1 \oplus m_2 \oplus m_3 & || \quad h_1 \oplus m_0 \oplus m_2 \\ W_1 & = & h_0 \oplus h_1 \oplus m_0 \oplus m_2 \oplus m_3 & || \quad h_0 \oplus m_1 \oplus m_2 \\ W_2 & = & h_0 \oplus h_1 \oplus m_0 \oplus m_1 \oplus m_2 & || \quad h_0 \oplus m_0 \oplus m_3 \\ W_3 & = & h_0 & || \quad h_1 \\ W_4 & = & m_0 & || \quad m_1 \\ W_5 & = & m_2 & || \quad m_3 \end{array} \ . \tag{2.20}$$

The message expansion in LANE-384 and LANE-512 is completely analogous. The only difference is that all sizes are doubled, *i.e.*, the 1024-bit message block M_i is split into four 256-bit parts, $m_0, \ldots, m_3$ and the 512-bit input chaining value H_{i-1} is split into two 256-bit parts. Then, (2.20) is used to compute the six 512-bit expanded message words $W_0, \ldots, W_5$.

2.5.2 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.

The rounds of the permutations use the building blocks described in Sect. 2.3. 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 2.10 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 [48, 21], where the appropriate constants and counter word are used as a round key, followed by SwapColumns.

In LANE-224 and LANE-256, the first layer permutations, P_0 until P_5, consist of six rounds each. The second layer permutations, Q_0 and Q_1, have three rounds each. In LANE-384 and LANE-512, the number of rounds in increased to eight rounds for the P_i's and four rounds for the Q_i's. Table 2.6 summarises the number of rounds in the permutations.

A round number r is assigned to each of the full rounds across all permutations, with the purpose to specify the constants and counter use of 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 2.7 lists the round numbers r in each of the permutations. Each full round is given its round number r as an extra parameter, as indicated in Figure 2.10. This parameter is then passed on to the AddConstants and AddCounter transformations, described in Sect. 2.3.4 and Sect. 2.3.5, respectively.

A pseudocode description of the permutations used in LANE-224 and LANE-256 is given in Figure 2.11, including an exact expression to compute the full round number r for each round. Figure 2.12 describes the permutations used in LANE-384 and LANE-512.

2.6 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 n, 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

$$\phi \,||\, \mathrm{bin}_{64}(l) \,||\, 0^\star \,||\, S \ . \tag{2.21}$$

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 2.4. If a salt value is not used, $\phi = \mathtt{00}_x$ and the salt S is filled with zero bits. If a salt value is used, $\phi = \mathtt{01}_x$. The size of the salt S is 256 bits for LANE-224 and LANE-256, and 512 bits for LANE-384 and LANE-512, as indicated in Table 2.1.

In the second part of the output transformation, a truncation is applied to compute the final message digest. No output truncation is required for LANE-256 and LANE-512, as the size of the chaining value is equal to the required digest size. In LANE-224 and LANE-384, however, this is not the case. The digest of LANE-224 is found by taking only the first, *i.e.*, leftmost 224 bits of the last 256-bit chaining value. Similarly, in LANE-384, only the first 384 bits of the last 512-bit chaining value are used.

More formally, the truncation operation for LANE-224 is given by

$$\mathrm{Trunc}_{224}\left(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_6 \,||\, x_7\right) = x_0 \,||\, x_1 \,||\, \cdots \,||\, x_6 \ . \tag{2.22}$$

For LANE-384, the truncation is defined similarly as

$$\mathrm{Trunc}_{384}\left(x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{11} \,||\, \cdots \,||\, x_{15}\right) = x_0 \,||\, x_1 \,||\, \cdots \,||\, x_{11} \ . \tag{2.23}$$

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

function Round(r, X)
1: $X \leftarrow \text{SubBytes}(X)$
2: $X \leftarrow \text{ShiftRows}(X)$
3: $X \leftarrow \text{MixColumns}(X)$
4: $X \leftarrow \text{AddConstants}(r, X)$
5: $X \leftarrow \text{AddCounter}(r, X)$
6: $X \leftarrow \text{SwapColumns}(X)$
7: **return** X

function LastRound(X)
1: $X \leftarrow \text{SubBytes}(X)$
2: $X \leftarrow \text{ShiftRows}(X)$
3: $X \leftarrow \text{MixColumns}(X)$
4: $X \leftarrow \text{SwapColumns}(X)$
5: **return** X

Figure 2.10: Pseudocode for the LANE permutation rounds.

Table 2.6: 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

Table 2.7: 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

Figure 2.11: 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

Figure 2.12: Pseudocode for the permutations in LANE-384 and LANE-512.

Chapter 3

Design rationale

This chapter discusses the rationale behind the design of Lane. All of the important design decisions are explained. The discussion of the rationale is structured by components of the Lane hash function. The advantages and disadvantages of Lane are also discussed in this chapter.

3.1 The iteration mode

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 [22, 46]. For this construction, it can be proven that if the compression function is collision resistant, so is the iterated hash function built on it.

The same iteration mode supports multiple digest lengths. One simply needs to compute the initial chaining value, which depends on the digest length, and after the iteration apply a suitable truncation to the message digest. The derivation of the initial chaining value is based on the Lane compression function, for ease of implementation.

3.1.1 The message padding

The message padding is simplified when compared to plain Merkle-Damgård, as used in the SHA family of hash functions [49]. As Lane uses an output transformation, which is simply an additional compression function call, it is natural to include the message length, *i.e.*, the Merkle-Damgård strengthening, in this extra block.

Because of this extra block, one can now simply pad a message with zero bits until the next block boundary. It no longer depends on the exact message length whether or not an extra padding block has to be introduced. This greatly simplifies implementation, and still results in an efficient iteration mode. If no salt is used, the initial chaining value can be precomputed, and the total number of compression function calls, including the output transformation, is

$$\#\text{calls to } f = \left\lceil \frac{l}{b} \right\rceil + 1 \ . \tag{3.1}$$

For plain Merkle-Damgård, assuming that the representation of the message length in the padding uses 64 bits, this is

$$\#\text{calls to } f = \left\lceil \frac{l + 65}{b} \right\rceil \quad . \tag{3.2}$$

This means that the LANE iteration mode uses at most one additional compression function call compared to plain Merkle-Damgård, but has the advantage that there is always one extra compression function call, *i.e.*, the LANE output transformation, whose 'message' input is not under the control of the adversary, except very limited influence via choosing the message length. In plain Merkle-Damgård, an adversary could choose the message length such that almost all of the padded message bits in the last block can be chosen freely.

3.1.2 The use of a counter

Additionally, the LANE mode of iteration borrows the idea of including a bit counter in every compression function call from the 'HAsh Iterative FrAmework' (HAIFA) of Biham and Dunkelman [10]. This stops several attacks on the iteration, at only a very modest cost. If no counter is used, a fixed point of the compression function, if found, can be concatenated to itself to form an *expandable message* [23, 35], *i.e.*, a set of message patterns of different lengths, all leading to the same internal hash state. Such an expandable message can for instance be used to construct efficient second preimage attacks on long messages [35]. Due to the use of a bit counter, however, it is no longer possible to concatenate a fixed point to itself, as it will only be a fixed point for a specific counter value [10].

3.1.3 The output transformation

An 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 used in the output transformation 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. Indeed, the output transformation takes a zero counter as input, which according to the specification is not possible in a normal message block.

The output transformation also offers additional protection against distinguishing attacks, as any potential bias in the compression function is expected to be destroyed by the output transformation.

3.1.4 The use of a salt value

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 [31]. Consider the scenario where an attacker constructs two colliding messages, and asks the victim to sign the first message. Because the second message has the same message digest as the first, the signature is also valid for the second message. If the victim chooses a random salt before signing the message, however, the collision that was carefully crafted by the attacker is destroyed with an overwhelming probability.

When a salt is used in LANE, this salt value is included in the derivation of the initial chaining value as well as in the output transformation. Both of these operations are simply LANE compression function calls with a specific message block and a zero counter input. Apart from the salt value, this message block also includes a flag byte ϕ. The purpose of this byte is to provide domain separation. More specifically, the only compression function calls in LANE that use a zero counter C occur exactly 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 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.

3.1.5 A parallel iteration mode

The iteration mode used by LANE is inherently sequential. Hence, it is not possible to benefit from having multiple CPU cores to accelerate the hashing of a single, long message. There is small-scale parallelism available inside the compression function, which can be used by a single CPU, as will be explained in Sect. 3.2. But this parallelism is too fine-grained to offset the synchronisation overhead required when using multiple independent CPU cores.

In many high-performance applications, this is not a problem. Indeed, often there are many smaller messages that need to be hashed. Consider for example a web server using TLS with HMAC-LANE for data authentication. The server needs to hash every packet it sends to and receives from the network. A machine with multiple CPU cores can process all of these independent messages in parallel.

For applications that do require parallelisable hashing of a single, long message, it is beneficial to use a separate parallel iteration mode. We propose a simple and easy to implement parallel mode that is built on top of the normal, sequential LANE. This mode is based on the seminal work of Damgård [22].

Let T be the desired level of parallelism, *i.e.*, up to T CPU cores can be utilised. Let b_{int} be the interleave factor, which defines the size of the blocks in which the message will be split. It is logical to choose b_{int} to be a multiple of the blocksize b of the underlying hash function, although this is not strictly required. Parse the message M into blocks of b_{int} bits:

$$m_0 \,||\, m_1 \,||\, \ldots \leftarrow M \ . \tag{3.3}$$

Then assign the blocks in turn to T streams $\mathcal{M}_0,\ldots,\mathcal{M}_{T-1}$:

$$\begin{aligned} \mathcal{M}_0 &= m_0 \,||\, m_T \,||\, m_{2\cdot T} \,||\, \ldots \\ &\vdots \\ \mathcal{M}_i &= m_i \,||\, m_{T+i} \,||\, m_{2\cdot T+i} \,||\, \ldots \\ &\vdots \\ \mathcal{M}_{T-1} &= m_{T-1} \,||\, m_{T+T-1} \,||\, m_{2\cdot T+T-1} \,||\, \ldots \end{aligned} \quad . \tag{3.4}$$

Finally, compute the digest of the message M as

$$\textsc{Lane}\Big(\textsc{Lane}\,(\mathcal{M}_0) \;||\, \textsc{Lane}\,(\mathcal{M}_1) \;||\; \cdots \;||\, \textsc{Lane}\,(\mathcal{M}_{T-1})\Big) \quad . \tag{3.5}$$

There are T inner hash functions, all of which are independent and can thus be evaluated in parallel. When the message M is long, the cost of the final hash function, which combines the results from the T streams, is negligible.

Note that this mode is not interoperable with the 'normal' mode of LANE, as the computed message digest is different. Also, different values for the interleave factor b_{int} and the parallelisation degree T result in a different message digest.

3.2 The compression function

The LANE compression function was designed to be simple to understand and easy to analyse. This aim for simplicity can be found in virtually every aspect of the design.

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 Sect. 4.4.1, 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.

3.2.1 The message expansion

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.

A similar structure, with four parallel branches, is found in the Rumba20 compression function [8]. In Rumba20, constants are used at the input to prevent finding preimages by inverting individual branches. The (linear) relations between the inputs of the various lanes of the first layer serve a similar purpose in LANE.

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. This property is described in more detail in Sect. 4.2.1.

Provable resistance is offered against meet-in-the-middle preimage attacks, as detailed in Sect. 4.9. In short, 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.

Also for implementors, the message expansion has several interesting properties. 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. This implementation approach is described in more detail in Sect. 5.1.1. If two (or more) CPU cores are available, it is also possible to let one CPU core precompute $P_4\left(M^h\right) \oplus P_5\left(M^l\right)$, while the second CPU core takes care of the rest of the lanes. In this setting the synchronisation overhead between the CPU cores is manageable.

3.2.2 The permutations

The permutations used in LANE are built using components of the AES block cipher [48, 21]. 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. Another benefit lies in resource constrained environments, requiring both a hash function and a block 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 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. Even though scenarios where the compression function is attacked by introducing differences via the bit counter are of no immediate concern, the method by which the counter is included provides resistance against such attacks. The rationale is that the bit counter can only influence a small part of the state in each round, and those influences cannot be cancelled out immediately in the next round, but instead diffuse to affect the whole state. The fact that the same counter value is used many times in the compression function serves to further complicate such cryptanalytic attempts.

The number of rounds in the permutations in the first layer was chosen to be six rounds for LANE-224 and LANE-256, and eight rounds for LANE-384 and LANE-512. The rationale behind this choice is to use as few rounds as possible, for performance reasons, but still enough rounds to offer an adequate security margin. We refer to the discussion of truncated differential analysis in Sect. 4.3 for a more detailed analysis concerning the required number of rounds in the first layer.

Concerning the number of rounds in the second layer of permutations, recall that the main purpose of the second layer is to preclude higher order differential distinguishers, such as the distinguishers described in Sect. 4.4. Such distinguishers are based on detecting the balancedness of the intermediate values after the first layer, which is a very fragile property. Almost any non-invertible and non-linear second layer would suffice to this end, but it is reasonable to ensure that every input bit influences every output bit, *i.e.*, to achieve full diffusion. It is also a logical choice to use the same type of permutations as in the first layer. Achieving full diffusion requires a minimum of three rounds. Hence, the second layer permutations are defined to have half as many rounds as the first layer permutations.

Unlike in the AES block cipher, the linear diffusion layer is not omitted in the last rounds of the permutations, even though its impact on the security of LANE is limited. Doing these extra operations simply makes many implementations faster and easier, both in high-performance software and hardware. Namely, we avoid handling a special case which would otherwise require multiplexers on the critical path in hardware, or extra tables or masking instructions in software. Only in applications where the MixColumns operation has to be computed explicitly, for instance in embedded software implementations, would omitting

MixColumns offer a performance benefit. In the AES, another reason to omit these operations, besides a performance gain in embedded implementations, is to achieve a similar structure for the inverse cipher. But as the permutations in LANE are only ever evaluated in the forward direction, this argument does not apply to LANE.

3.2.3 The constants

The constants serve to diversify the permutations, in order to avoid any attack based on the similarity of the parallel permutation lanes. An important design goal is that it should be possible to generate the constants on-the-fly in an inexpensive way. This 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. Even though its output stream does not possess any strength in the cryptographic sense, the statistical properties are sufficiently good for the purposes of LANE. A 32-bit LFSR was chosen to match the size of the columns. The feedback polynomial is a primitive polynomial, ensuring a cycle length of $2^{32} - 1$.

The only security-related requirement on the constants is that the constants used in different permutation lanes should be different. The first constant, used to initialise the LFSR, was chosen such that no two constant bytes used in the same position of two different lanes are equal. Additionally, the number of times that two constant bytes, used in the same position in a different lane, are the one's complement of each other was minimised. An exhaustive search resulted in the conclusion that this complement property cannot be avoided. There exist ten starting states for which this happens in only a single byte. Of these ten, we picked the starting state with the lowest numerical value. The source code for this search is included in the submission package.

3.3 Advantages and limitations of LANE

3.3.1 Advantages

- LANE design is simple. This makes LANE easy to understand and implement. Also, simplicity is an important advantage for cryptanalysis. Complex designs are often hard, or even impossible to analyse in a structured way. The design of LANE, on the other hand, allows for a relatively easy analysis of its security.

- LANE incorporates several features that can greatly improve its security, at only a modest cost in performance. In particular, LANE offers the possibility of using a salt value, uses a counter and has an output transformation.

- Components from the AES block cipher [48, 21] are reused as building blocks in LANE. As discussed above, this allows existing cryptanalytic results on the AES to be used in the security analysis of LANE. Also,

implementations of LANE can benefit from existing work on the implementation of the AES on a wide variety of platforms. In particular, LANE can benefit from dedicated hardware support intended to accelerate the AES, like for instance the Intel AES-NI instruction set [17].

- One of the design goals of LANE was to provide a high degree of parallelism in the compression function. At the same time, care was taken to keep the memory requirements modest for a serialised implementation. Thus, LANE is flexible in implementation and scales well across a wide range of platforms and applications.
- LANE can easily be extended to support any digest length up to 512 bits. One simply needs to derive the initial chaining value for the desired digest length, and apply a suitable truncation at the end.
- There is a clear and detailed rationale, which was presented in this chapter, supporting every design decision.

3.3.2 Limitations

- The iteration mode is not parallelisable. For most applications, this is not a problem. For applications where a parallelisable iteration mode is important, we suggest to use the parallel mode described in Sect. 3.1.5.
- Because the size of the intermediate chaining values was chosen to be equal to the digest length, Joux' multicollision attack [32] can be applied to LANE. Refer to Sect. 4.12 for a more in-depth treatment of multicollisions.

Chapter 4

Security analysis

In this chapter, we discuss the security of the LANE construction in general, as well as of the hash functions LANE-256 and LANE-512 in particular. We list known bounds on security and present attacks on reduced versions of the hash functions.

In Sect. 4.1, we suggest ways in which LANE could be reduced, to perform cryptanalysis. Sections 4.2, 4.3 and 4.4 address differential attacks and their applicability to weakened versions of LANE. As the LANE compression function shares a certain similarity with wide-block Rijndael, Sect. 4.5 summarises cryptanalytic results on Rijndael, and their relevance to LANE. Sect. 4.6 is dedicated to algebraic attacks.

Sections 4.8-4.12 discuss the resistance of LANE to various generic attacks. Sect. 4.13 summarises security arguments on the mode of operation; a technical report detailing these results is included as a separate document [1]. Sect. 4.14 concludes with a statement on the expected security of the LANE hash functions.

4.1 Reduced versions of LANE for cryptanalysis

This section discusses various ways in which LANE could be reduced, in order to construct weakened variants. Such variants can be useful in cryptanalysis, as they allow one to understand the margin offered by the full LANE against a particular type of attack.

A first, obvious way to reduce LANE is to vary the number of rounds used in the permutations. For example, lowering the number of rounds increases the probability of (truncated) differentials. The number of rounds in the first layer (P_i) and second layer (Q_i) can be varied independently.

Another option is to reduce the number of lanes in the first layer. A variant where a single lane is removed from the first layer, for instance, would be based on a (5,3,3) linear code, which is simply a shortening of the original code. The number of lanes can be reduced further, but then the property that in a differential attack, always more than half of the lanes are active, is lost.

Finally, LANE could be reduced by omitting the entire second layer, *i.e.*, the Q_i permutations. The compression function output would then be found as the XOR of all six lanes, P_0 to P_5.

4.2 Standard differential cryptanalysis

Differential cryptanalysis was originally introduced by Biham and Shamir [11] as a means to cryptanalyse symmetric encryption primitives (block ciphers). It has also been used with success to break hash functions, *e.g.* [14, 24].

In a differential attack, collisions are determined by considering *pairs* of messages, which have a fixed difference, *i.e.*, the input difference of the *characteristic*. This condition strongly reduces the search space in which the attacker looks for collisions. It is hoped that the fraction of the collisions that lies within this reduced search space is larger and/or easier to find than in the unrestricted search space.

An important difference between differential cryptanalysis of hash functions and differential cryptanalysis of block ciphers is stated in the following fact.

Fact 1. *The absence of secret keys in hash functions can be exploited by the cryptanalyst in order to reduce the complexity of a differential attack.*

For instance, instead of choosing inputs ('plaintexts'), the cryptanalyst can choose any intermediate state, and compute backwards to determine the inputs. The effect is that a number of active S-boxes can be 'bypassed', resulting in a decrease of the complexity of a differential attack.

We consider differential attacks with nonzero differences in the message only. Since each individual lane of LANE implements a permutation in the space of n-bit message inputs, collisions can be obtained only in the XOR combiners. In order to obtain a collision at the output of the compression function, either a collision must be obtained in both of the XOR combiners at the input of the second layer, or both of the lanes in the second layer will be active.

We discuss only the resistance against differential attacks provided by the first layer of lanes. If the lanes of the second layer are active, then they will increase the security against differential attacks.

4.2.1 Active lanes in the first layer

This section describes a property of the LANE message expansion which ensures that, in a differential attack, at least four lanes in the first layer will be *active*, *i.e.*, have a difference. To this end, we first introduce an alternative description of the LANE message expansion, based on a linear code over GF(4).

We adopt the standard polynomial representation of GF(4) using X^2+X+1 as a primitive polynomial to define the multiplication of field elements. A string of two bits b_0b_1, where b_0 is the most significant bit, can now be mapped to an element in GF(4), and vice versa

$$b_0b_1 \quad \leftrightarrow \quad b_0 \cdot X + b_1 \ . \tag{4.1}$$

4.2.1.1 An alternative description of the LANE message expansion

The message expansion of LANE is based on a linear (6,3,4)-code over GF(4) which is known as the *hexacode*. Its generator matrix is given by

$$G = \begin{bmatrix} 1 & X & X & 1 & 0 & 0 \\ X & 1 & X & 0 & 1 & 0 \\ X & X & 1 & 0 & 0 & 1 \end{bmatrix} \ . \tag{4.2}$$

This code has length 6, dimension 3 and minimum distance 4. The minimum distance property can be easily verified by exhaustively listing all 64 codewords.

We now describe the construction of the LANE message expansion. The input chaining value H and the message block $M = M^h \,||\, M^l$ are mapped to elements of GF(4) as follows, where $0 \leq i < n/2$:

$$\begin{array}{lcl} \eta_0^i & \leftrightarrow & (H)_i \cdot X + (H)_{i+n/2} \\ \eta_1^i & \leftrightarrow & (M^h)_i \cdot X + (M^h)_{i+n/2} \\ \eta_2^i & \leftrightarrow & (M^l)_i \cdot X + (M^l)_{i+n/2} \end{array} \quad . \tag{4.3}$$

Here, $(H)_i$, $(M^h)_i$ and $(M^l)_i$ denote the i-th bit of H, M^h and M^l, respectively, where bit 0 is the most significant bit. Now, for each i, encode $\left[\begin{array}{ccc} \eta_0^i & \eta_1^i & \eta_2^i \end{array}\right]$ using the linear code described in (4.2):

$$\left[\begin{array}{cccccc} \mu_0^i & \mu_1^i & \mu_2^i & \mu_3^i & \mu_4^i & \mu_5^i \end{array}\right] = \left[\begin{array}{ccc} \eta_0^i & \eta_1^i & \eta_2^i \end{array}\right] \cdot G \ . \tag{4.4}$$

Finally, the elements of GF(4) $\mu_0^i,\ldots,\mu_5^i$ are mapped back to the n-bit expanded message words $W_0,\ldots,W_5$ in the same way:

$$\begin{array}{lcl} \mu_0^i & \leftrightarrow & (W_0)_i \cdot X + (W_0)_{i+n/2} \\ & \vdots & \\ \mu_5^i & \leftrightarrow & (W_5)_i \cdot X + (W_5)_{i+n/2} \end{array} \quad . \tag{4.5}$$

This entire procedure can be written as a simple partitioned matrix multiplication over GF(2), where I denotes the $n/2 \times n/2$ unity matrix:

$$[W_0 \,||\, W_1 \,||\, \ldots \,||\, W_5] = \left[H \,||\, M^h \,||\, M^l\right] \cdot \left[\begin{array}{cccccc} I & 0 & I & I & I & I \\ 0 & I & I & 0 & I & 0 \\ I & I & I & 0 & I & I \\ I & 0 & 0 & I & I & 0 \\ I & I & I & I & 0 & 0 \\ I & 0 & I & 0 & 0 & I \end{array}\right] \quad . \tag{4.6}$$

It is easy to see that this is equivalent to (2.20), which was used to define the message expansion in Sect. 2.5.1.

4.2.1.2 Minimum distance and active lanes

The LANE message expansion can thus be seen as a parallel application of a linear (6,3,4)-code over GF(4) to $n/2$ 'slices'. The values in each such slice must form a valid codeword. Note that the six elements of GF(4) that comprise a codeword are each input to a different first-layer lane.

Consider two different inputs to the message expansion, which yield two different sets of expanded message words, $\langle W_0, \cdots, W_5 \rangle$ and $\langle W'_0, \cdots, W'_5 \rangle$. In differential cryptanalysis terminology, we say that there is at least one *active* 'slice', *i.e.*, at least one 'slice' has a difference.

As the minimum distance of the message expansion code is four, the Hamming distance between the two codewords in any *active* 'slice' must be at least four. This implies that at least four expanded words must have a difference. Hence, in a differential attack, there are always at least four *active* lanes. This property always holds, even when the difference is only in the chaining value.

Table 4.1: Lower bounds on the number of active S-boxes in one lane for LANE-256 and LANE-512.

Rounds	LANE-256	LANE-512
1	1	1
2	5	5
3	9	9
4	25	25
5	34	41
6	45	$\geq$57
7	(52)	$\geq$58
8	(65)	$\geq$62

4.2.2 Active S-boxes per lane

Next we determine the minimum number of active S-boxes in an active lane. Each active S-box decreases the probability by a factor of at least 2^6. The input to one lane can be seen as two AES states for LANE-256, or four for LANE-512. If these states were processed independently by six, resp. eight rounds of the AES, then the minimum number of active S-boxes in one lane would be 30 resp. 50. This is the minimum for six resp. eight rounds of the AES block cipher, and could thus be achieved by only activating a single AES state.

But as the SwapColumns operation, see Sect. 2.3.6, mixes the AES states together, the minimum number of active S-boxes is in fact higher. There appears to be no elegant formula to compute the minimum number of active S-boxes. A computer search for LANE-256 and LANE-512 resulted in the lower bounds given in Table 4.1. The figures for 7 and 8 rounds in LANE-256 are of theoretical interest only, since we use 6 rounds only. The bounds for 6, and definitely for 7 and 8 rounds in LANE-512 might not be tight.

4.2.3 Breaking reduced versions

We describe a structure for a hypothetical differential collision attack that targets a collision at the XOR combiners after the first layer and hence will apply to LANE up to a certain number of rounds per first-layer permutations P_i, but with an arbitrary number of rounds per second-layer permutations Q_j. It is intended to demonstrate Fact. 1, *i.e.*, that the absence of a secret key in a hash function, allows an attacker to reduce the complexity of a differential attack.

Let $\Delta = (\Delta_0 \,||\, \Delta_1)$ be any n-bit difference, where n is the digest length. Introducing the difference $(0, \Delta, \Delta)$ at the inputs (H, M^h, M^l) of the LANE compression function yields the differences $(0, \Delta', \Delta', 0, \Delta, \Delta)$ at the inputs to the six lanes, where $\Delta' = (\Delta_0 \oplus \Delta_1) \,||\, \Delta_0$. Given two suitable single-lane characteristics C_0, C_1 transforming Δ' into Δ'_o with probability p and Δ into Δ_o with probability q, the differences at the two XOR's after the first layer cancel, causing the output difference of the compression function to be zero, see Figure 4.1. Not that this can be extend to a larger number of compatible characteristics, *i.e.*, characteristics that have the same input and output difference, but differ in the intermediate differences. In this more general case, the probabilities of all such characteristics should be added together. For the sake of simplicity, we

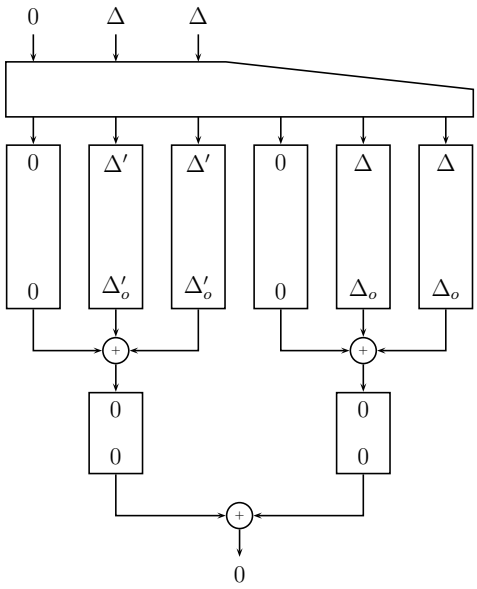

Figure 4.1: A collision differential for LANE.

will describe the case where only a single characteristic is used.

A randomly chosen pair of compression function inputs simultaneously follows all four active parallel branches of such a differential with probability $p^2 \cdot q^2$. Assume for now that this probability is large enough for a right pair to exist amongst the set of all possible inputs (see Sect. 4.2.4 for further discussion on the satisfiability of differential characteristics in LANE). Fact 1 then suggests that the complexity $p^{-2}q^{-2}$ of finding the right pair could be reduced by imposing control over the intermediate state of some lanes.

More specifically, any choice of inputs to some three lanes determines a valid input to the message expansion. For instance, the attacker can fix an intermediate state in lanes P_4 and P_5 and, calculating backwards and forwards, find M^h and M^l in such a way that the layer of S-boxes at the round where the attacker fixes the intermediate state is passed with probability 1. If the round with the greatest number of active S-boxes within the characteristic is chosen as the starting point, this can considerably reduce the complexity of finding a right pair. Also note that the attacker can deal with P_4 and P_5 independently. For fixed M^h, M^l, P_2 becomes invertible with respect to the chaining value H, allowing exactly the same approach for finding a right pair for this lane. If this procedure for P_2, P_4, P_5 is repeated k times, and k right pairs are found for each of these three lanes, the attacker can generate k^3 input pairs by forming all combinations of the independently obtained values for H, M^h and M^l. As soon as k^3 exceeds p^{-1}, one can expect to find a right pair for P_1, which is then simultaneously a right pair for all four active lanes by construction.

As outlined in section 4.2.2, the probability of a single-lane characteristic is upper bounded by 2^{-6a}, where a is the minimum number of active S-boxes for the number of rounds per lane. During the process of finding a right pair for P_4, P_5 and P_2, a certain number of active S-boxes can be disregarded. However,

the attacker has no further control over the input to the fourth active lane P_1, implying that the complexity of this attack is lower bounded by the expected complexity 2^{6a} of finding a right pair for P_1 amongst the set of $k^3 \geq 2^{6a}$ available inputs.

For the six rounds employed in LANE-256, the complexity of such an attack is at least $2^{6 \cdot 45} = 2^{270}$; for LANE-512 with eight rounds, the work factor is at least $2^{6 \cdot 62} = 2^{372}$. Both values are well above the respective birthday bounds of 2^{128} and 2^{256}.

This attack, however, breaks LANE variants faster than a standard birthday approach in case of up to 3 rounds per P-lane for the digest size $n = 256$ bits and up to 5 rounds per P-lane for the 512-bit digest version.

4.2.4 Maximum probability of a trail

In Section 4.2.3, we described an efficient method to determine right pairs for characteristics over reduced versions of LANE. This method works, *provided that such right pairs exist*. In this section, we show that for the full versions of LANE, the overwhelming majority of the characteristics does not exhibit a right pair.

We adopt the usual hypotheses of independence and stochastic to bound the probability of characteristics equivalence [39]. The message expansion ensures that there are always at least 4 active lanes, see Sect. 4.2.1. Each active lane has at least 45 active S-boxes for LANE-256, or at least 62 active S-boxes for LANE-512. Each active S-box has probability at most 2^{-6}. This results in the following bounds for the probability of a characteristic Q:

$$\text{LANE-256:} \qquad \Pr(Q) \leq \left(2^{-6}\right)^{4 \cdot 45} = 2^{-1080} \ . \tag{4.7}$$

$$\text{LANE-512:} \qquad \Pr(Q) \leq \left(2^{-6}\right)^{4 \cdot 62} = 2^{-1488} \ . \tag{4.8}$$

This gives a conservative security margin against this class of attacks, even in the presence of highly effective message modification techniques. This suggests that for the full versions of LANE-256 and LANE-512, even with the best possible message modification techniques, it is not feasible to find a right pair, simply because with very high probability, there exists no right pair.

4.3 Truncated differential cryptanalysis

In the previous section, we have analysed the probability that a pair of message blocks with a fixed difference follows a single predefined characteristic. However, due to the fact that the operations used in LANE are all byte-oriented, we are likely to find a very large number of different characteristics with a comparable probability and the same or similar input and output differences. In a typical attack, each of these characteristics will be equally useful, and hence it makes sense to analyse the probability that a message pair satisfies any of them. Truncated differential cryptanalysis [36] is a technique which does exactly that.

4.3.1 Truncated differentials

Instead of imposing specific differences in every round, a truncated differential only specifies where these differences should be zero. An example of a truncated

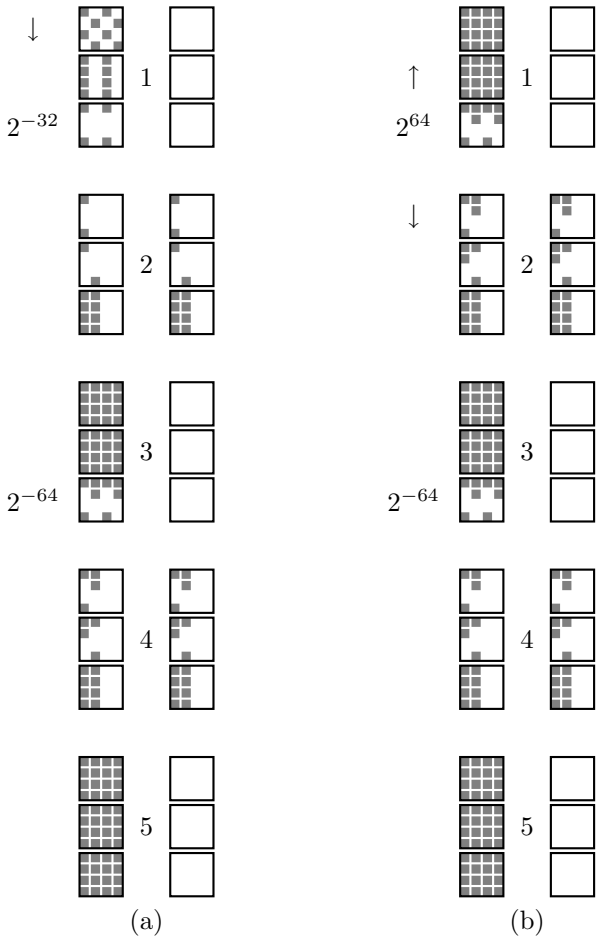

Figure 4.2: Truncated differentials in one lane of Lane-256

differential for a single lane of Lane-256 is shown in Fig. 4.2(a). For each round, except the last one, the figure depicts the differences in the AES states before the ShiftRows transformation, after the ShiftRows transformation, and after the MixColumns transformation. Byte positions where differences are allowed are marked in grey. Since byte-equalities are preserved by all operations, except for the MixColumns transformation, this is the only stage where a reduction in probability can take place. In our example, we end up with a total probability of 2^{-96}. Note that the MixColumns transformation in the last round can be moved behind the XOR's which combine the lanes, and is therefore irrelevant if we intend to force collisions in those XOR's. This is why the last round is omitted.

4.3.2 Identifying the optimal truncated differential

The probability of a truncated differential is, on its own, not a very good measure for its usefulness. As a trivial example, consider a truncated differential without

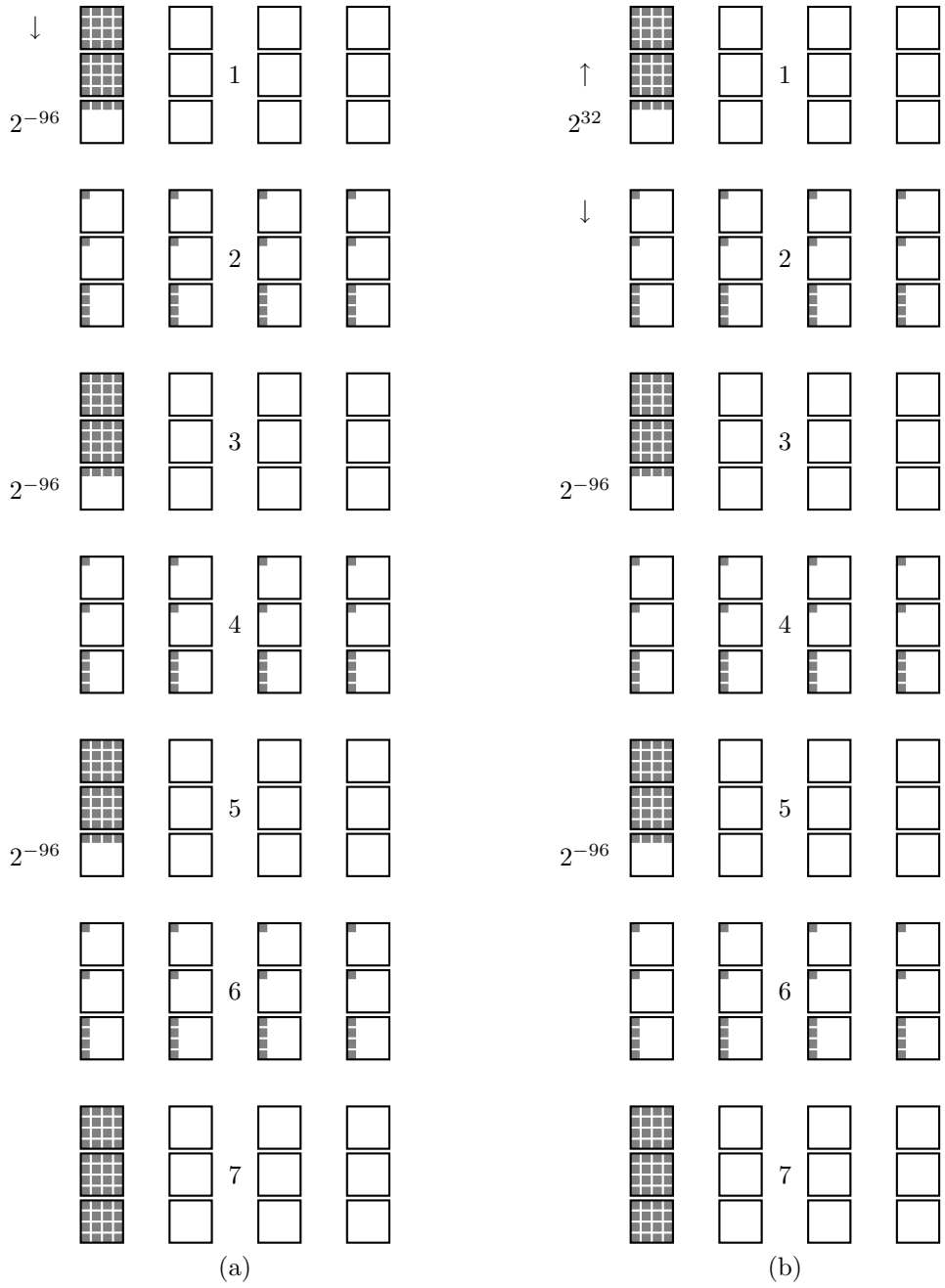

Figure 4.3: Truncated differentials in one lane of LANE-512

any zero differences, which, despite its probability of 1, is clearly useless. To be of any use, a truncated differential should demonstrate that a pair satisfying the input difference is significantly more likely to result in the desired output difference than a random pair. In Fig. 4.2(a), for instance, a consistent input pair is expected to result in 16 equal bytes at the output with a probability of 2^{-96}, versus 2^{-128} for a random pair. In fact, it can be shown by computer search that this factor of 2^{32} is the highest attainable gain for a single lane of LANE-256.

Another property that influences the usefulness of a truncated differential is the number of degrees of freedom that are left at the input. Without any additional restrictions, these degrees of freedom could be used to reduce the effort of finding right pairs. Consider for instance the set of all 2^{64} input states which are constant (say, zero) in all but the 8 grey bytes at the input of Fig. 4.2(a). By sorting the 2^{64} corresponding output states and returning all pairs which have equal values in the rightmost AES state, we would in effect have checked in the order of 2^{128} pairs. Hence, we expect to find about 2^{32} right ones, and this with an effort of only 2^{64}.

An additional optimisation which can be applied if the attacker is free to choose parts of the input state is depicted in Fig. 4.2(b). The approach is similar to the one described above, but this time the attacker starts after the first MixColumns, which saves a factor 2^{32} in probability, and therefore results in 2^{64} right pairs. For each of these pairs the attacker then reverses the first round in order to find the corresponding input pairs, each of which will be equal in the rightmost AES state.

The same reasoning can be applied to the two truncated differentials of LANE-512 shown in Fig. 4.3. However, if we start with a set of 2^{128} input states in Fig. 4.3(a), then we do not expect to find any right pair at all, since

$$2^{2 \cdot 128} \cdot 2^{-3 \cdot 96} = 2^{-32} \,.$$

In order to find a single right pair, we will therefore have to repeat this procedure 2^{32} times, resulting in a total workload of 2^{160}. Alternatively, we could start with a set of 2^{32} states after the first MixColumns, as shown in Fig. 4.3(b). This will have to be repeated 2^{128} times in order to compensate for the fact that

$$2^{2 \cdot 32} \cdot 2^{-2 \cdot 96} = 2^{-128} \,,$$

leading to the same total effort of 2^{160} as before. Note however that the first approach requires 2^{128} of memory, whereas a table of 2^{32} would suffice for the second one.

4.3.3 Using truncated differentials for collision searching

An attack using truncated differentials to construct a collision pair would then proceed as follows. First, we ensure that only 4 lanes are active, which is optimal as shown in Sect. 4.2.1. This can be achieved by choosing $\Delta m_0 = \Delta m_2$. Then, the probability that a colliding right pair exists for a given chaining input is given by

$$\text{LANE-256:} \qquad 2^{64} \cdot 2^{512} \cdot \left[\left(2^{-96}\right)^2 \cdot 2^{-128} \right]^2 = 2^{-64} \,,$$

$$\text{LANE-512:} \qquad 2^{128} \cdot 2^{1024} \cdot \left[\left(2^{-3 \cdot 96}\right)^2 \cdot 2^{-128} \right]^2 = 2^{-256} \,.$$

Note that, especially for LANE-256, even if such a pair exists, we do not expect to be able to determine this with an effort less than 2^{64}, after which the pair would need to be recovered in less than 2^{128}.

4.4 Higher order differential cryptanalysis

Higher order differentials where introduced as an extension of differential cryptanalysis, using higher order derivatives [36]. The i'th order derivative of a function f at the point $a_1, \ldots, a_i$ is defined as follows [38]:

$$\Delta_a f(x) := f(x+a) - f(x)$$
$$\Delta^{(i)}_{a_1,\ldots,a_i} f(x) := \Delta_{a_i}\left(\Delta^{(i-1)}_{a_1,\ldots,a_{i-1}} f(x)\right), \quad i > 1.$$

Standard differential cryptanalysis used first-order derivatives.

The following property of the LANE message expansion is important.

Property 1. *Let $(W)_{ti}$, $0 \le t < 5$, $0 \le i < n$, denote the $6n$ output bits of the message expansion, and let $(H)_i$, $(M^h)_i$, $(M^l)_i$, $0 \le i < n$ denote the $3n$ input bits. Then for each t there exists a set of 12 binary constants a_{0t}, a_{1t}, a_{2t}, a_{3t}, a_{4t}, a_{5t}, b_{0t}, b_{1t}, b_{2t}, b_{3t}, b_{4t}, b_{5t} such that*

$$(W)_{ti} = \begin{cases} a_{0t}(H)_i + a_{1t}(M^h)_i + a_{2t}(M^l)_i & \\ \quad + a_{3t}(H)_{i+n/2} + a_{4t}(M^h)_{i+n/2} + a_{5t}(M^l)_{i+n/2}, & \text{for } 0 \le i < n/2; \\ b_{0t}(H)_{i-n/2} + b_{1t}(M^h)_{i-n/2} + b_{2t}(M^l)_{i-n/2} & \\ \quad + b_{3t}(H)_i + b_{4t}(M^h)_i + b_{5t}(M^l)_i, & \text{for } n/2 \le i < n. \end{cases}$$

4.4.1 A fourth order differential distinguisher

Property 1 implies the existence of 4th order differentials that have probability 1 over the message expansion and the first layer of lanes.

Corollary 1. *Let $F(H, M^h, M^l)$ denote the function that consists of the message expansion, the first layer of lanes and the two XOR combiners following it. Let δ be an arbitrary $(n/2)$-bit difference and let*

$$a_0 = (0,0;0,0;0,\delta)$$
$$b_0 = (0,0;0,0;\delta,0)$$
$$a_1 = (0,0;0,\delta;0,0)$$
$$b_1 = (0,0;\delta,0;0,0)$$
$$a_2 = (0,\delta;0,0;0,0)$$
$$b_2 = (\delta,0;0,0;0,0).$$

Then

$$\Delta_{a_i,b_i,a_j,b_j} F(H, M^h, M^l) = 0 \text{ with probability 1,}$$

for all $i, j \in \{0, 1, 2\}$.

Proof. Property 1 implies that over the 16 inputs defined by the 4th order differential, each lane input W_t will take 1, 2 or 4 different values, and each value a multiple of 4 times. Consequently, each lane output will occur an even number of times. Hence for each of the XOR combiners it holds that the XOR of the 16 outputs equals zero. □

We could not determine a way to extend this property through the second layer of permutations.

4.4.2 Square attacks on the compression function

The structure of LANE is byte oriented, suggesting a possible square attack might be applicable [20, 37, 41]. The attack is made slightly more complex by the message expansion and the need to track the square property over six lanes, but it is essentially the same.

For example, in LANE-256, consider 256 message blocks, for which one byte of m_0 (*e.g.*, the first one) accepts all possible values, while all the other bytes are set to the same value. This ensures that the first byte in both halves of W_0 is active (*i.e.*, accepts all values), the first byte of the left half of W_1 is active, as well as the first byte of both halves of W_2, and the left of W_4.

In lanes where there is only one active byte, we have after two full rounds sixteen bytes which are active (columns 0,1,4,5). Looking at the third round, we learn that all bytes are balanced (*i.e.*, the sum of all values in these bytes is zero). In lanes where there are two active bytes (one in each half), after the second round all the bytes are active, and after three rounds, all bytes are balanced. Thus, if we reduce the length of the P_i permutations to three rounds, we know that the XOR of the outputs of these permutations is balanced. Without the Q_j permutations, we could at this point find that all the bytes are balanced. Thus, given the output values of 255 messages out of the set, we could predict the output of the missing message. This could also be used as a distinguisher to distinguish the output of the compression function from random.

We can extend the attack by one round, by considering structures of 2^{32} messages, chosen such that after one round they generate 2^{24} sets of 256 messages each, where in each such set, the first byte of the state (or the first byte of each 128-bit half of the state) obtains all possible values, while the other bytes are fixed. Similarly, we can extend the square property one more round, by taking structures of 2^{128} values. A possible structure would use

$$m_0 = m_1 = m_2 = m_3 = i \quad \text{for all} \quad 0 \leq i < 2^{128} \ . \tag{4.9}$$

We conclude that the best square property of LANE-256 is of 5 rounds, and thus, after the additional round, the inputs to the Q_j's permutations have no structural property. And even if somehow the attacker succeeded in finding such a structure after 6 rounds, the Q_j's would destroy the remaining structural properties.

For LANE-512 the analysis is similar, but with an extra round. Starting from only one active byte, after four rounds we expect that all the bytes of the internal state are balanced. Hence, for LANE-512 the best square property is for 6 rounds, making LANE-512 immune to this attack as well.

4.4.3 Multiset distinguishers

As a further generalisation, one can consider distinguishers based on (much) larger sets of inputs. As an example, consider the following case. Keep h_0, h_1, m_0 and m_1 constant and saturate m_2 and m_3, *i.e.* assign every possible value to them. The size of this set of messages is $2^{n/2}$. Then, we know that every expanded message word also gets assigned every possible value, except for W_3 and W_4, which are constant. Now, after the first layer, the outputs of P_0, P_1, P_2 and P_5 are saturated. After the XOR combiners, we see that the input to Q_0 sums to zero, *i.e.* it is balanced, and the input to Q_1 is still saturated. This implies that also the output of Q_1 is saturated, but nothing useful can be said about the output of Q_0 or the output of the compression function.

4.5 Cryptanalysis of wide-block Rijndael

Nakahara *et al.* investigated the security of wide-block Rijndael [33, 34]. Since a permutation in LANE has certain similarities with Rijndael-256 we summarise these attacks. Recall that a full permutation round in LANE consists of the following sequence of transformations: SubBytes, ShiftRows, MixColumns, AddConstants, AddCounter and SwapColumns, while a round in wide (large block) Rijndael consists of AddRoundKey, SubBytes, ShiftRows and MixColumns. Note that ShiftRows differs in Rijndael-128 (*i.e.* AES and LANE) and Rijndael-256. In fact, the combination of the SwapColumns and ShiftRows operations in LANE-256 can be viewed as the equivalent of the (redefined) ShiftRows operation in Rijndael-256.

There exist higher-order multiset (differential and linear) distinguishers for up to 7 rounds of Rijndael-256 [33]. These distinguishers trace the status of 128-bit words, and thus require sets of 2^{128} chosen plaintexts at a time. The rationale behind the multiset technique is to use balanced sets of bits to attack permutation mappings (cipher rounds).

Similar distinguishers can be constructed for the first layer of LANE, see for example Sect. 4.4.3. However, the second layer of permutations completely stops these distinguishers.

Impossible-differential (ID) attacks on 7-round Rijndael-256 are shown in [34]. Typical ID distinguishers follow the miss-in-the-middle technique. Two truncated differentials, one in the encryption direction and one in the decryption direction, are combined to form an impossible truncated differential. A key recovery attack can be built upon an ID distinguisher, as subkey guesses for which the impossible differential would be followed, can be eliminated with certainty.

In order to construct a distinguisher for the LANE compression function based on impossible differentials, it is not sufficient to have an impossible differential for a single LANE permutation. Assuming that the unknown keying material enters the compression function via the chaining value, four lanes in the first layer will be affected. Hence, an impossible differential needs to cover these four lanes in the first layer, as well as both lanes in the second layer. It seems very unlikely that such an impossible differential can be found.

4.6 Algebraic attacks

In algebraic attacks, the operation of a symmetric cryptographic primitive is represented as a system of polynomial equations over $GF(2)$ or $GF(2^n)$, which is then attempted to be solved using various techniques and expression manipulations. Since LANE is based on the AES, its security with regard to algebraic attacks is closely related to that of the AES. As a single state of LANE encompasses multiple AES states, the resulting equation systems for LANE are expected to have comparable degree, but higher dimension.

There has been an extensive analysis of the equation systems corresponding to the AES, however, all techniques have so far only been successful against very small scaled variants. The approaches that are theoretically best understood are methods based on Gröbner bases [5]. Improving over Buchberger's classical algorithm [13], Faugère's F4 and F5 algorithms [25, 26] are the best known methods to compute Gröbner bases. Extensive experiments indicate that those algorithms are only successful for very small AES variants, such as ten rounds of an 1×1 state or four rounds of a 2×1 state [15].

An alternative approach to solving nonlinear polynomial equations is to linearise the system by introducing new independent variables for each occurring nonlinear monomial term. Since this method can only be effective if the number of linearly independent polynomials approximately equals the number of monomials, the Extended Linearisation (XL) algorithm [18] extends the original equation system before linearisation by multiplying it with all monomials up to some degree in order to generate enough linearly independent equations. Experimental evidence indicates that the XL algorithm offers little to no advantage compared to Gröbner basis techniques [4].

Finally, the Extended Sparse Linearisation (XSL) method [19] aims at improving on the XL technique by multiplying the polynomials only by products of monomials that occur in the original system. So far, also this method has been unsuccessful in realistically-sized AES equation systems [15].

We conclude that it seems highly unlikely that algebraic attacks can be successfully applied to LANE.

4.7 Attacks based on reduced query complexity

4.7.1 General comments

Since LANE is a permutation-based hash function, it can be studied in the *ideal permutation model* [55], which is very similar to the ideal cipher model and the random oracle model. Theorem 1 of [55] states that for a compression function $f : \{0,1\}^{sn} \rightarrow \{0,1\}^{rn}$ using k calls to n-bit permutations, collisions can be found with certainty using approximately

$$k \cdot 2^{n(1-(s-r)/k)} \tag{4.10}$$

permutation queries (at most). Instantiating this with the parameters for LANE ($s = 3, r = 1, k = 8$) yields *query* complexities of 2^{195} and 2^{387} for LANE-256 and LANE-512, respectively.

An interesting discussion of the merits and limitations of the ideal/random models can be found in [28]. An important observation is that there are two

ways to measure the complexity of an attack. On the one hand, there is the *practical complexity*, which measures the (expected value of the) time complexity of the adversary. This is the most natural complexity measure and also the most relevant measure. On the other hand, there is the *query complexity*, which measures the number of queries that are made to the oracle. This complexity is often used in security proofs, mostly because it is easier to bound.

Since the practical complexity of an adversary is always larger than its query complexity, the ideal oracle model can be used to prove bounds on the security of hash function designs.

There are two important criticisms on this model. Firstly, the distinction between oracle queries and computations made by the adversary is artificial. A cryptographic hash function uses an instantiation of the permutation (block cipher), which is public. Hence, it can be argued that any cryptographic hash function can be broken without making a single query to the oracle. Secondly, the model ignores the practical complexity of the adversary. It is well-known that an information theoretic adversary who is given a full description of a hash function can always find collisions and preimages. On the other hand, returning to the case of hash function with the same dimensions as Lane-256 or Lane-512, the adversary has to find the actual collision in sets of at least 2^{256} or 2^{512} values, so that an acceleration in constructing these sets, such as the one given by (4.10), does not reduce the practical complexity of the attack.

Summarising, results on the query complexity of attacks on hash function designs do not always have a big impact on the actual security of the design. Nevertheless, they can be first steps in the development of better attacks. Therefore, we list here our results.

4.7.2 Results on Lane

The message expansion of Lane expands three inputs to six outputs which are then independently fed into the permutations of the first layer. We call a particular combination of chaining value and message block an *input* to the message expansion. If some value occurs more than once at a permutation input when hashing a set of messages, the corresponding permutation output has to be computed only once. More precisely, whenever the sum of the numbers of distinct values at each of the six outputs is lower than the number of different message expansion inputs, the output of the first layer can be computed with reduced effort, resulting in some speedup of the evaluation of the entire compression function for this set of inputs.

Property 1 leads to two corollaries which can be used to reduce the query complexity of Lane adversaries. The first corollary looks at Lane without the second layer of lanes.

Corollary 2. *It is possible to compute the inputs of the second layer of lanes for 2^{6p} different inputs to the compression function, using only $6 \cdot 2^{2p}$ queries to the P-lanes (exactly 2^{2p} queries to each of the 6 P-lanes).*

Proof. Choose p indices j_t with $0 \leq j_t < n/2$. Consider the 2^{6p} inputs where the bits h_{j_t}, $h_{j_t+n/2}$, m_{j_t}, $m_{j_t+n/2}$, $m^*_{j_t}$, $m^*_{j_t+n/2}$ take all possible values and the remaining bits are constant. Property 1 implies that the words w_t will differ in the bits at the $2p$ positions $j_t, j_t + n/2$ only. Hence each lane needs to be queried for at most 2^{2p} different values. □

Adding the queries to the lanes of the second layer, and assuming that the lanes in the second layer are twice as fast as the lanes of the first layer, we obtain an acceleration factor given by

$$\frac{7 \cdot 2^{6p}}{6 \cdot 2^{2p} + 2^{6p}} \ , \tag{4.11}$$

which very rapidly converges to 7 as p approaches infinity. This simply means that hashing many messages chosen in this way can be done up to 7 times faster than the straightforward approach. This is not considered to be an issue, as it is merely a constant factor. Similar optimisations to speed up the hashing of many messages can be applied to virtually any hash function.

The following corollary will improve upon this number.

Corollary 3. *It is possible to compute the outputs of the compression function for 2^{10p-2n} different inputs, using only 2^{2p+3} queries to the lanes.*

Proof. We start by applying Property 1 twice. First, we apply it on the first layer, with p varying bits. We compute and store the 2^{6p} outputs of the first layer in list L_1. Next, we apply it on the second layer, with q varying bits. We store the 2^{4q} inputs for which we can compute the output in list L_2.

We have then made $6 \cdot 2^{2p}$ queries to the lanes of the first layer, and $2 \cdot 2^{2q}$ queries to the lanes of the second layer. The number of inputs for which we can compute the output of the compression function, equals the number of entries that appear in both lists. This number can be approximated by $2^{6p+4q-2n}$.

If we choose $p = q > (2n + 3)/8$ then

$$\#\text{queries} = 2^{2p+3} = 2^{10p-(8p-3)} < 2^{10p-2n} = \#\text{outputs} \tag{4.12}$$

□

For $n = 256$, the number of queries drops below the number of outputs when $p = q = 65$. For $n = 512$, this happens when $p = q = 129$.

We are not aware of any method to exploit these properties to reduce the practical complexity of any attack against Lane.

4.7.3 Bounds for query complexity

4.7.3.1 A lower bound for the query complexity of Lane

Consider any message expansion mapping three inputs to six outputs, with the only requirement to have a minimum distance greater than half the number of lanes in order to prevent the meet-in-the-middle attack outlined in section 4.9. Assume that N different values (each comprising chaining value and message block) are input into the message expansion and denote the number of distinct values that occur at each of the six outputs by $L_0, \ldots, L_5$. The outputs of the first layer of parallel lanes for the entire set of N inputs can then be computed with $\sum_{i=0}^{5} L_i$ permutation queries.

A minimum distance of four implies that any mapping from the input to some three outputs is invertible. This in turn shows that for each $i \neq j \neq k$,

the product $L_i L_j L_k$ must be at least equal to N. As the latter holds for any three outputs, we also know that $\left(\frac{1}{6}\sum_{i=0}^{5} L_i\right)^3 \geq N$, and hence

$$\sum_{i=0}^{5} L_i \geq 6\sqrt[3]{N}. \tag{4.13}$$

Therefore, the number of permutation queries needed to compute the first layer of permutations for N inputs is lower bounded by $6\sqrt[3]{N}$, independent of the linearity of the message expansion (only imposing the minimum distance requirement). This lower bound is tight for the message expansion of Lane and we conclude that the query strategy of Corollary 2 with $L_i = 2^{2p}$ and $N = 2^{6p}$ is in fact optimal.

4.7.3.2 Alternative linear message expansions

Finally, we discuss the relative merits of alternative linear message expansions. For the sake of clarity, we restrict the treatment to Lane-256 (and hence Lane-224). The wider variants can be handled completely analogously by considering $GF(2^{512})$ instead of $GF(2^{256})$.

In order to construct a message expansion that does not exhibit Property 1, a linear $(6, 3, 4)$ code over $GF(2^{256})$ could be used. The systematic form of the generator matrix of such a code would be

$$\begin{bmatrix} 1 & 0 & 0 & \alpha & \beta & \gamma \\ 0 & 1 & 0 & \delta & \varepsilon & \zeta \\ 0 & 0 & 1 & \eta & \theta & \iota \end{bmatrix}, \tag{4.14}$$

where the Greek letters denote elements of $GF(2^{256})$. Since the attacker can apply invertible transformations at both input and output, this generator matrix can always be transformed to:

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & \alpha' & \beta' \\ 0 & 0 & 1 & 1 & \gamma' & \delta' \end{bmatrix}. \tag{4.15}$$

Over $GF(2)$, this is a 768×1536 matrix. We show now that a weakened form of Property 1 holds also in this case.

We start by choosing an arbitrary $2p$-dimensional vector space V ($2p < n$) and require that the inputs of each lane are in this vector space. This imposes conditions on the inputs of the message expansion. The requirements on the first 3 lanes are equivalent to: $H, M^h, M^l \in V$. This ensures that $H + M^h + M^l \in V$, which implies that also the condition on the input of the fourth lane is satisfied. The requirements on the last two lanes restrict the number of permissible M^h- and M^l-values to smaller vector spaces. We denote the dimensions of these vector spaces by $k_1, k_2 < 2p$. In this scenario, the output of the first layer for $2^{2p+k_1+k_2}$ inputs can be computed with $6 \cdot 2^{2p}$ permutation queries.

A sufficient condition on $2p$ to have both $k_1 > 0$ and $k_2 > 0$ can be obtained as follows. The conditions on the last two lanes are $H + \alpha' M^h + \gamma' M^l \in V$ and $H + \beta' M^h + \delta' M^l \in V$, respectively. If we require that, besides $M^h, M^l \in V$, also the products $\alpha' M^h, \beta' M^h, \gamma' M^l$ and $\delta' M^l$ are elements of V, then both

conditions trivially hold. The three conditions on M^h form a set of $3 \cdot (n - 2p)$ equations in n unknowns, so $k_1 > 0$ if $n - 3 \cdot (n - 2p) > 0$. Hence, $2p > \frac{2}{3}n$ is a sufficient condition for having $k_1 > 0$. Analogously, $2p > \frac{2}{3}n$ implies $k_2 > 0$. In case of a linear message expansion for LANE-256, this corresponds to $6 \cdot 2^{170.7}$ permutation queries.

4.8 Wagner's generalised birthday attack

Wagner [60] describes a sub-exponential algorithm for the generalised birthday problem where one is given k lists of n-bit values and wants to select one value from each list such that the selected values sum to zero. For $k = 2$ and XOR as summation operation, this is the well-known birthday problem. Provided that the lists are long enough and the list elements are independently and uniformly selected at random, it can be solved with good probability in $\mathcal{O}(2^{n/2})$ time. Under the same assumptions, Wagner's generalised algorithm has a running time of $\mathcal{O}(k \cdot 2^{n/(1+\lceil \log_2 k \rceil)})$, which in particular implies that the birthday problem with four lists can be solved in $\mathcal{O}(2^{n/3})$ time.

LANE has one XOR combiner with two inputs and two XOR combiners involving 3 inputs each. For the 3-sum problem, no algorithm faster than the birthday complexity is known. Since Wagner's algorithm assumes that k is a power of two, the best we can do is to emulate the case $k = 2$ by solving the problem $x_0 \oplus x_1 = c$, where c is a fixed, randomly selected value from the third list. Moreover, the assumption that the list entries are independent is invalid for LANE, since are linearly dependent according to the message expansion. Indeed, even though h, m, m^* can be chosen independently for the second 3-XOR, their choice immediately fixes the inputs for the first 3-XOR.

Consider now LANE without the message expansion, so that the independence assumption holds. Since selecting $k = 3$ yields no advantage compared to the birthday bound, an attacker can consider the XOR of all six P_i outputs. In this case, Wagner's algorithm would be applied with $k = 4$ (in $2^{n/3}$ time), searching for a solution to $x_0 \oplus x_1 \oplus x_2 \oplus x_3 = c_4 \oplus c_5$, where c_4, c_5 are random choices from the lists corresponding to P_4 and P_5. Once such a solution $x'_0, \ldots, x'_5$ is found, we know that $x'_0 \oplus \cdots \oplus x'_5 = 0$, which is equivalent to $x'_0 \oplus x'_1 \oplus x'_2 = x'_3 \oplus x'_4 \oplus x'_5$. Hence, the attacker can use this to obtain a *value* x such that $H_i = Q_0(x) \oplus Q_1(x)$ and is then left with the task of attacking a smaller number of AES-like rounds with identical input but different keys (the constants). Alternatively, he can apply the birthday attack on *differences* instead of values. In this scenario, the effect of the different constants cancels, so that any pair following this differential would immediately yield a collision for the entire compression function.

Summarising, both the message expansion and the second layer of permutations contribute to making attacks based upon Wagner's algorithm for the generalised birthday problem inapplicable to LANE.

4.9 Meet-in-the-middle attacks

A meet-in-the-middle attack can be used to construct collisions or (second) preimages by simultaneously modifying two consecutive message blocks. A ba-

sic version of the attack can be described as follows. In order to reach a certain target value for H_{i+1} from a given H_{i-1}, the attacker will determine an intermediate result V and define two maps g, g^* such that

$$g(H_{i-1}, M) = V = g^*(H_{i+1}, M^*), \tag{4.16}$$

with g, g^* efficiently computable functions and M, M^* two independent parts of the message input. Subsequently, the unknown V is eliminated from the equations and a solution for M and M^* is searched by constructing two lists. The first list contains output values for g; the second list contains output values for g^*. A match between both lists means that a message has been found which takes H_{i-1} to H_{i+1}. For example, if the compression function $f(H, M)$ is invertible, then the adversary can choose $g = f$ and $g^* = f^{-1}$.

In the case of LANE, the compression function is not invertible, but a cryptanalyst could try to construct a similar attack within one application of the compression function. This can be done only if the adversary is able to partition the lanes into two disjunct sets, whose inputs can be restricted to be independent of the other set. The message expansion prevents this attack, as its minimum distance of 4 ensures that each of these sets need to comprise at least four lanes. Since there are only six lanes in total, it is not possible to find two non-overlapping, independent sets of lanes.

4.10 Long message second-preimage attacks

The standard Merkle-Damgård iteration of a compression function guarantees collision resistance of the overall construction if the compression function itself is collision-resistant. However, as discovered by Dean [23], this does not hold in the case of second preimages, if fixed points of the compression function can be found.

Definition 5. A *fixed point* of a compression function $f(\cdot, \cdot)$ is a chaining value h and message block m for which it holds that

$$h = f(h, m) \ . \tag{4.17}$$

Fixed points of a compression function can be concatenated to form an *expandable message*. This is a set of message patterns of different lengths which all lead to the same output chaining value. In a long-message second preimage attack [23, 35], an expandable message allows an adversary to target simultaneously any intermediate chaining value instead of just a single one, reducing the expected work factor of a second preimage attack from $\mathcal{O}(2^n)$ to $\mathcal{O}(2^{n-k})$, where 2^k is the length of the message.

For hash functions based on the Davies-Meyer construction, it is very easy to find fixed points [47, 52]. Let $E_K(\cdot)$ be a block cipher with key K. Then the Davies-Meyer construction is

$$f(h, m) = E_m(h) + h \ . \tag{4.18}$$

Constructing a fixed point can be done by choosing an arbitrary message block m, and computing

$$h = E_m^{-1}(0) \ . \tag{4.19}$$

Now, it follows from (4.18) that this yields a fixed point.

For the LANE compression function, constructing fixed points in such a straightforward way does not seem to be possible. Even though the permutations in LANE are invertible, the structure of the compression function does not allow for the construction of fixed points, as the linear conditions on the expanded message words can only be satisfied probabilistically. Hence, finding a fixed point for the compression function of LANE should be no easier than constructing a preimage for the compression function.

Even more so, if a fixed point for the compression function of LANE could be found in an efficient way, it would still not allow for the construction of an expandable message. As discussed in [10], the inclusion of the counter in LANE prohibits the concatenation of fixed points. Also Kelsey and Schneier's multicollision-based method for constructing an expandable message [35] is not applicable thanks to the counter. Finally, the attacks on dithered hash functions by Andreeva *et al.* [2] are also foiled by the inclusion of the counter.

Thus, we conclude that the second preimage resistance of LANE does not degrade when the challenge message is long, and hence the iteration mode of LANE offers full n-bit security for second preimages (see also [1]).

4.11 Length-extension attacks

Given the hash value of a (partially) unknown message m (including padding etc.), length-extension attacks aim to infer the hash value of some message $m \,||\, x$, where the suffix x may be chosen freely by the adversary. This is an important consideration for message authentication codes (MAC's) based on hash functions, as a successful length-extension attack would lead to a forgery.

In the plain Merkle-Damgård construction, the mere knowledge of the message length l and the hash value $h(m)$ of a (partially) unknown message m enables an attacker to calculate the hash value of messages of the form $\mathrm{pad}_l(m) \,||\, x$, where x is an arbitrary suffix. Indeed, the intermediate chaining value after processing $\mathrm{pad}_l(m)$, which always aligns to a block boundary, is precisely equal to $h(m)$, which is known to the attacker.

In LANE, the output transformation, which can be regarded as another compression function call on a special padding block, is processed using the special counter value zero, which cannot occur in any regular message block. This makes it impossible for an attacker to emulate the output transformation using a regular message block. Therefore, length extension attacks are not applicable to LANE.

4.12 Multicollision attacks

In a multicollision setting, the attacker wants to find $k > 2$ messages all hashing to the same value. Ideally, finding a k-way multicollision should require a computational effort of $\mathcal{O}(2^{n\cdot(k-1)/k})$. However, the multicollision attack by Joux [32] allows to find a 2^l-way multicollision for an n-bit hash function using the Merkle-Damgård iteration impressively faster than that. It requires an effort of only $l \cdot C(n)$, where $C(n)$ is the complexity of finding a single collision, which will be at most $2^{n/2}$ by the birthday paradox. Joux's attack works by

concatenating a chain of internal collisions: For $i = 1, \ldots, l$, the attacker computes colliding pairs $\langle M_i, M_i' \rangle$ such that $h(H_{i-1}, M_i) = h(H_{i-1}, M_i')$, where H_0 is the initial value and $H_i := h(H_{i-1}, M_i) = h(H_{i-1}, M_i')$. Now, after appropriate padding, the 2^l messages $X_1 \,||\, \cdots \,||\, X_l$, where $X_i \in \{M, M_i'\}$, all hash to the same value, yielding a 2^l-way multicollision.

If the compression function allows for efficient calculation of fixed points, the method of Kelsey and Schneier [35] gives multicollisions of arbitrary size with $\mathcal{O}(2^{n/2})$ effort. As outlined in section 4.10, this improvement does not apply to LANE. Joux' approach, however, is applicable. In order to preclude it, a chaining value of at least $2n$ bits would be required.

Since Joux' attack combines single collisions, its complexity directly depends on the best possible single-collision attack against the hash function. Naturally, the effort of applying the multicollision attack can never be lower than that of finding a single collision. Since we require that it should be already computationally infeasible to construct a single collision, multicollisions do not present a bigger threat than collisions. In particular, LANE's security level against single collisions, which meets the theoretical bound of $2^{n/2}$, is not reduced in any regard by the fact that many messages colliding to the same hash value can be found with only little more effort.

One of the main application of multicollisions is the construction of long message second preimage attacks [35]. As was mentioned in Sect. 4.10, the inclusion of the counter precludes these attacks for LANE. Hence, we argue that, although Joux' multicollision attack does apply to LANE, it is not a threat in practice. Finally, it should be noted that the fact that this multicollision attack applies to LANE is a mere consequence of the mode of iteration and does not imply any weakness in the compression function itself.

4.13 On the mode of operation

In this section, the reduction-based provable security approach is used to assess the security of the LANE iteration. More precisely, security claims on the LANE iteration under some concrete assumptions on the underlying compression function are stated. Following this approach, concrete security bounds on the computational complexity of an adversary against the LANE iteration can be shown. We also exhibit information theoretic results on the LANE iteration which indicate security against generic attacks under the assumption that the compression function is an ideal primitive. For a full security analysis on the LANE iteration, we refer to the work of Andreeva [1]. Here, we present a summary of the main results.

Similarly to the known Merkle-Damgård iterative principle [46, 22], Andreeva shows that both the non-salted and salted versions of the LANE iteration are provably collision secure. Following Rogaway's human-ignorance approach [54], the advantage of an adversary against the LANE hash function is related to that of another adversary against the LANE compression function to derive a tight security bound.

The upper bounds on the advantage of information theoretic adversaries against the second preimage and preimage security, respectively, of the non-salted LANE hash function indicate that 2^n evaluations of the underlying ideal compression function need to performed to break the respective security prop-

erty. Moreover, making a (variant of) preimage security assumption on the output transformation of LANE and adopting some randomness extraction and regularity properties on the iterative portion of the non-salted LANE hash function, a tight preimage security bound on the LANE iteration is exhibited.

For the salted version of the LANE hash function a broad set of security notions is developed that capture most of the important attack scenarios of randomised hashing. In addition, the possibility of attacks under equal or distinct and known or secret salt values is taken into account. The same (as for the non-salted LANE hash function) security against information theoretic adversaries is obtained in the second preimage case, and the preimage case. The salted LANE iteration also provides second preimage security guarantees of order 2^n against adversaries who first commit to a target message and then are given a random target salt value.

The latter information theoretic results show that no generic attacks on the second preimage and preimage security on both salted and non-salted LANE hash function variants succeed in under 2^n number of evaluations of an ideal compression function.

Another important security feature of the LANE iterative design in both salted and non-salted versions is the security against extension attacks and lack of structural flaws ensured by the prefix-free property of the processed inputs. The latter design characteristic of LANE ensures its indifferentiability from a random oracle according to the work of Coron *et al.* [16] and pseudorandom function behaviour according to Bellare *et al.* [6].

The suggested parallel processing method, see Sect. 3.1.5 is also shown to be collision secure when the underlying hash function is collision secure. Under the second preimage/preimage security and some mild assumptions on the min entropy extraction properties of the hash functions, it is also shown that the parallel mode of operation is second preimage/preimage secure.

4.14 Expected strength of LANE

To the best of our knowledge, the complexity of finding collisions, first, and second preimages for LANE is $\mathcal{O}(2^{n/2})$, $\mathcal{O}(2^n)$, and $\mathcal{O}(2^n)$, respectively. In all three cases, the complexities refer to generic approaches applicable to any hash function: the birthday attack for finding collisions and simple brute force for preimages. As discussed in this chapter, none of the dedicated attack attempts yielded a lower complexity than the generic attacks. This holds for any of the specified digest lengths $n = 224, 256, 384, 512$.

Length-extension attacks are generally precluded, as outlined in section 4.11. Our analysis also did not indicate any imbalance concerning the strength of individual output bits, so forming an n'-bit hash function by selecting $n' < n$ digest bits in an arbitrary manner yields a hash function fulfilling the above criteria in terms of n' instead of n. In particular, no n'-bit truncation of an n-bit hash value is a valid n'-bit digest for the same message, since the initial value depends on the digest length n.

The claimed security levels for each of the specified digest lengths are summarised in Table 4.2.

Table 4.2: Expected strength of LANE against cryptanalytic attacks.

Attack	LANE-224	LANE-256	LANE-384	LANE-512
Collision attacks	2^{112}	2^{128}	2^{192}	2^{256}
Preimage attacks	2^{224}	2^{256}	2^{384}	2^{512}
Second preimage attacks	2^{224}	2^{256}	2^{384}	2^{512}
Length-extension attacks	not applicable			
Output bits equally strong	yes			

Chapter 5

Implementation aspects

This chapter discusses implementation aspects of Lane. Several software implementations of Lane targeting general purpose CPU's have been created, as well as two hardware implementations. They are presented and evaluated in Sect. 5.1 and Sect. 5.3, respectively. Sect. 5.2 discusses the implementation aspects of Lane on 8-bit embedded systems.

5.1 General purpose CPU's

As Lane is based on the AES block cipher [48, 21], the implementation techniques that are commonly used for the AES can be applied directly to Lane. A prevailing technique is to group the SubBytes and (part of) the MixColumns operations into four 8-to-32-bit lookup tables [21]. ShiftRows can be implemented as a simple reordering of the indices, so it does not require any actual instructions. The AddConstants and AddCounter operations in Lane are implemented in the same way as AddRoundKey in the AES. Finally, like ShiftRows, also the SwapColumns operation does not require any explicit instructions, just an appropriate permutation of the indices.

There is extensive literature on fast AES implementations [3, 9, 43, 42, 44, 56, 61, 62]. The design of Lane is such that entire rounds of the AES block cipher are used as components. Hence, virtually all of the fast implementation techniques and 'tricks' apply to Lane as well.

We wrote an optimised implementation of Lane in ANSI-C, using the standard, well-known techniques for the implementation of the AES [21]. In addition, we also wrote a very similar implementation in x86 assembly using the MMX instruction set as a source of eight extra registers. This reduces the register pressure, and shows a considerable improvement in performance in our test results. Finally, we developed a bitsliced implementation of Lane, inspired by the work of Matsui *et al.* on the AES [42, 44]. Details on this implementation are given in Sect. 5.1.1.

We measured the performance of our implementations, using three different software suites: Microsoft Visual Studio, GNU GCC and the Intel C compiler. Details on our test hardware and these three software are given in Table 5.1. We tested both short, 64 byte messages and long, 32 kilobyte messages, and normalised the cycle count by dividing it by the message length, *i.e.*, we use

Table 5.1: Test platform for the software implementations of Lane.

CPU:	Intel(R) Core(TM) 2 Duo T8100 2.1 GHz (supports MMX, SSE, SSE2, SSSE3, SSE4.1) 3072 KB cache memory
Memory:	1024 MB
Platform 1 (64-bit)	
Operating System	Ubuntu Linux 8.04 x86_64
Compiler	GNU C compiler (GCC) version 4.2.3
Compiler flags	`-O3 -fomit-frame-pointer`
Assembler	GNU MMX assembler (GAS) version 2.18.0
Platform 2 (32-bit)	
Operating System	Microsoft Windows XP professional SP2
Compiler	Microsoft Visual Studio 2008 Version 9.0.21022.8 RTM
Compiler flags	`/O2`
Platform 3 (64-bit)	
Operating System	Ubuntu Linux x86_64
Compiler	Intel C compiler (ICC) 10.1 20080801
Compiler flags	`-O3`

'cycles per byte' (cpb) as a performance metric. The measurement results are presented in Table 5.2. As the performance of Lane-224 and Lane-384 are identical to Lane-256 and Lane-512, respectively, we only give data for the latter two. Also, our two assembly implementations currently only support Lane-256, hence no data on Lane-512 is given for these implementations.

We note that apparently, the choice of the compiler has a large impact on the performance. The Intel C compiler (ICC) achieves a speed which is very close to that of our MMX assembly implementation. We expect that further improvements can be made to these implementations. As several highly advanced, very fast AES implementations exist, e.g., the very recent work by Bernstein and Schwabe [9], we expect that (much) faster implementations of Lane can be made using these techniques.

5.1.1 Bitsliced implementation

Similarly to the AES block cipher, the C implementation of Lane relies heavily on table lookups. Several authors have demonstrated side-channel attacks against such software implementations of AES, using cache-timing analysis to gather information on these data-dependent table lookups [7, 51]. While side-channel attacks are not relevant for hash functions in their most straightforward application, they become a potential threat once the hash function is used to process secret values such as the message authentication key in an HMAC construction [27]. Thus, we provide an alternative proof-of-concept implementation of Lane-256 that is constant-time and consequently not vulnerable to cache-timing attacks.

Table 5.2: Performance measurement results of our LANE implementations. The test platform, and the three software suites we used, are described in Table 5.1.

Implementation	Platform	64 byte message	32 kbyte message
Optimised ANSI-C implementation of LANE-256	GNU (1) Microsoft (2) Intel (3)	130.67 cpb 104.75 cpb 79.78 cpb	43.02 cpb 40.46 cpb 26.17 cpb
MMX Assembly implementation of LANE-256	GNU (1)	79.84 cpb	25.66 cpb
Bitsliced implementation of LANE-256	GNU (1)	87.19 cpb	30.20 cpb
Optimised ANSI-C implementation of LANE-512	GNU (1) Microsoft (2) Intel (3)	1069.97 cpb 923.11 cpb 864.46 cpb	176.97 cpb 152.24 cpb 145.31 cpb

5.1.1.1 A Cache-Timing Resistant Implementation of LANE-256

The constant-time implementation of LANE-256 uses bitslicing to implement the AES S-box. That is, instead of using table look-ups, we compute the S-box output on-the-fly, using its representation as a concatenation of eight 8-to-1-bit Boolean functions. Such a bitsliced implementation of the AES S-box was first proposed by Matsui [42]. Later, Matsui and Nakajima [44] reported a particularly efficient implementation of bitsliced AES on the Intel Core 2 Duo processors, achieving speeds of up to 10.2 cycles per byte in modes of operation where sufficient parallelism is possible, such as the counter mode. Similarly, the parallelism available in LANE allows for an efficient bitsliced implementation.

Our implementation uses eight 128-bit XMM registers to store part of the LANE-256 state, one register for each bit position in a LANE state. That is, we collect in the first register bits from the least significant bit position in each LANE-256 byte, in the second register bits from the second bit position, and so forth. In order to fill the 128-bit XMM registers and fully utilise their width, we need to be able to process 128 bytes of state in parallel. At first glance, LANE-256 does not lend to such optimal parallelism: the six parallel P-lanes contain altogether 192 bytes, whereas the two Q-lanes in the second, sequential layer contain only 64 bytes. However, we observe that lanes P_4 and P_5 are independent of the chaining value and thus can be processed before the actual chaining value is known. This observation allows us to process the LANE-256 state in two parts of 128 bytes each. Namely, during each compression function call, we first process the Q-lanes of the previous call together with lanes P_4 and P_5. The output of the Q-lanes then provides us with the chaining value required to process lanes P_0, P_1, P_2, P_3.

5.1.1.2 Performance

We have implemented the compression function of LANE-256 in a bitsliced manner, using GNU assembly. Table 5.3 lists the number of instructions required in one LANE round. In addition to the compression function, our implementation

SubBytes	205
ShiftRows	8
MixColumns	43
AddConstants	8
AddCounter	8
SwapColumns	72
Total	344

Table 5.3: Number of XMM instructions in one LANE round.

contains bitslicing for input messages, and inverse bitslicing for the hash output, so that the bitsliced implementation is fully compatible with the standard implementation. Notice that apart from the input message, also the counter needs to be bitsliced on the fly for each compression function call, introducing an overhead compared to an AES implementation.

Including format conversion overhead for input, output and counters, the bitsliced implementation achieves a speed of 30.2 cycles per byte on our test platform (see Table 5.1 for platform details). We conclude that at least on 64-bit platforms, it is possible to implement LANE in a cache-timing resistant manner without a significant penalty in performance.

5.1.2 Intel AES-NI instruction set

In a white paper [17], Intel has announced AES-NI, a new set of instructions that are going to be introduced in the next generation of Intel processors, as of 2009. The AES-NI extension consists of six instructions that will provide full hardware support for the AES block cipher [48, 21].

As LANE reuses rounds of the AES as components, the Intel AES-NI instruction can also be used to create a fast implementation of LANE. LANE only uses full AES rounds in the encryption direction, hence only one of the six new instructions, AESENC, is required to implement LANE.

The AESENC instruction consists of the ShiftRows, SubBytes, MixColumns and AddRoundKey operations. The AddRoundKey operation in the AES is functionally equivalent to AddConstants in LANE. Other operations used in LANE can be implemented using instructions from the SSE/SSE2 instruction set. The message expansion and AddCounter can be implemented using the PXOR instruction, and either SHUFPD or SHUFPS can be used to implement SwapColumns, depending on the LANE variant.

As the underlying hardware that implements the AESENC instruction is fully pipelined [17], a new AES round can be started each clock cycle, provided that there are no data dependencies. The latency of the AESENC instruction is 6 cycles [17], which implies that six parallel AES rounds suffice to keep the pipeline filled. LANE offers ample opportunities for parallel AES rounds. In the first layer, there are 12 independent AESENC instructions in each round. In the second layer, there are only four parallel AES rounds. But, by scheduling the second layer of one compression function call in parallel with the P_4 and P_5 lanes of the next compression function call, as explained in Sect. 5.1.1, the parallelism can be balanced out more evenly.

As processors supporting the AES-NI instruction set are not yet available on the market, we can only estimate the performance of Lane on such a machine. For a parallel mode of AES, a throughput of about 12 cycles per block, or 1.2 cycles per AES round is claimed [17]. Counting only the AES rounds in a Lane-256 compression function call would yield a performance of $84 \cdot 1.2/64 = 1.575$ cycles per byte for Lane-256. But as other components of the Lane compression function, which are negligible in other implementations, can likely no longer be ignored, a performance of around 5 cycles per byte seems more reasonable. Still, this shows that Lane has the potential to achieve a very high performance on platforms that offer a fast, hardware accelerated way to compute AES encryptions.

5.2 Embedded systems with an 8-bit CPU

As Lane is based on rounds of the AES block cipher, its performance on 8-bit CPU's can be estimated based on the existing literature on the implementation of the AES on these platforms. Rinne *et al.* [53] present an implementation of the AES on an 8-bit AVR microcontroller, inspired by the 8-bit AES code of Gladman [30]. Their implementation has a code size (ROM size) of 3410 bytes, and is able to do an AES encryption in 3766 CPU cycles.

The message expansion of Lane-256 can be implemented using 288 8-bit XOR operations. An 8-bit XOR operation typically takes a single CPU cycle on these CPU's, hence the message expansion costs about 288 CPU cycles. An AES encryption consists of ten rounds, so the cost of a single AES round can be estimated at 377 CPU cycles. One Lane-256 compression function call contains 84 AES rounds, yielding a total of 31 668 CPU cycles. The AddCounter operation consists of four 8-bit XOR's, and is used 34 times per compression function call – 136 CPU cycles in total. Finally, the computation of the constants needs to be counted. A single step of the LFSR can be implemented in 8 CPU cycles, and it needs to be carried out 272 times, resulting in a cost of 2176 CPU cycles. Adding these components gives a total cost of 34 268 CPU cycles.

Based on this rough estimate, we can expect a real implementation of Lane-256 on this 8-bit platform to require about 35 000 CPU cycles per compression function. This corresponds to an expected performance of roughly 550 cycles per byte. As most of the program code will be the same as for an AES implementation, we expect that an implementation of Lane-256 should fit in less than 5 kilobytes of ROM.

The amount of RAM required by a Lane-256 implementation on a resource-constrained system can be estimated as follows.

- 768 bits of read/write memory to store the input chaining value H_i, and for intermediate storage. Note that it is possible to reuse the memory used for storing H_i after the fourth lane of the first layer has been started, because the last two lanes are independent of H_i. This trick allows to save 256 bits of memory.

- 512 bits to store the message block. Note that the message expansion does not need to modify this memory, so it could be shared with another application, *e.g.*, a transmit or receive buffer.

Table 5.4: Hardware evaluation of the LANE hash function.

Design	Area [GE]	Frequency [MHz]	# of Cycles	Throughput [Mbps]
LANE-224/LANE-256†	16 462	100	2201	23.3
LANE-224/LANE-256‡	243 486	305	11	14 191
LANE-384/LANE-512‡	466 190	286	14	20 958

† Compact implementation.
‡ High-throughput implementations.

- 64 bits for the counter.
- (optionally) 256 bits to store the salt value.
- 32 bits for computing the constants on-the-fly.

For example, a LANE-256 implementation not using salts requires 172 bytes of RAM, of which the 64 bytes containing the current message block can be shared.

Tillich *et al.* [59] propose to add a small hardware accelerator of only 1.1 kGates to an AVR microcontroller, which increases the performance of an AES encryption by a factor 3.6 compared to the pure software implementation of [53]. Of course, such techniques will also greatly benefit the performance of LANE. It is even more interesting for embedded systems requiring both a hash function and a block cipher. When using LANE and the AES, a single investment in additional hardware can be used both for faster hashing and faster encryption.

5.3 Hardware implementation

The hardware performance evaluation of the LANE hash function was done by synthesising the proposed designs using 0.13 μm CMOS standard cell library. The code was first written in GEZEL [29], then compiled to VHDL and synthesised using the Synopsys Design Vision tool [58]. The synthesis results are given in Table 5.4.

As the ample parallelism provided by LANE allows for much flexibility in high-throughput implementations, our main goal was to show that LANE can achieve a very high throughput at the cost of consumed gate area. Additionally, by implementing a compact version, we have also shown that the same algorithm can be used in more constrained environments where the available gate area is a limiting factor.

The target for the high-throughput implementations was to minimise the critical path of the design. To perform the first layer of permutations, we used 6 permutation blocks in parallel where each of them contained 2 full AES engines (4 for LANE-384 and LANE-512). The two permutations from the first layer were also reused for the second layer.

The straightforward implementation of LANE-224/LANE-256 resulted in the critical path of 5.60 ns and the cycle count of 9. The critical path was placed from the input of the message injection function, going along the permutation block and ending at the input of the second layer of permutation. As the message injection function was performed only once per input message block, we

moved the state registers at the input of the permutation blocks. This approach resulted in the faster design, shortening the critical path to 4.28 ns. One more clock cycle had to be spent in order to perform the complete round, but the final throughput increased by about 20 %. The critical path was now placed along the permutation block and also contained the 3-input XOR gate at the output of the first layer permutations. By storing the output of the first layer back to the state registers and then performing an XOR operation, we introduced one more clock cycle and reduced the critical path down to only 3.28 ns (3.49 ns for Lane-384/Lane-512). The final design achieved a high throughput of 14.2 Gbps and 21.0 Gbps for Lane-224/Lane-256 and Lane-384/Lane-512, respectively.

As can be seen from Table 5.4, the most compact implementation is obtained for Lane-224/Lane-256 algorithm and consumes approximately 16.5 kGE. The major part of the compact design is consumed by the message expansion, though it is performed only once per input message block. Hence, we also evaluated the circuit size assuming that the message expansion is performed outside of the hash engine. This resulted in a smaller design with a gate area of only 11.7 kGE. The compact implementation was made using only one permutation block. Inside the permutation we used a single compact AES S-box [45] and a single AES MixColumns block. This approach resulted in a large number of cycles (2201), while on the other hand it efficiently reduced the final gate count. We used three 256-bit registers to maintain the internal state. Note that our only goal for the compact implementation was to have a small die size, regardless of the circuit speed. Hence, we fixed the frequency to 100 MHz and synthesised our design.

The throughput was calculated according to the following equations:

$$\text{Throughput}_{256/224} = \frac{\text{Frequency}}{\#\text{ of Cycles}} \times 512 \ , \tag{5.1}$$

$$\text{Throughput}_{512/384} = \frac{\text{Frequency}}{\#\text{ of Cycles}} \times 1024 \ . \tag{5.2}$$

Our hardware performance figures show that the Lane cryptographic hash functions can be implemented very efficiently and provide very high throughputs, up to 21 Gbps. On the other hand, the compact implementation shows that, at the cost of speed, the Lane hash functions can be considered as a good candidate for constrained environments. We believe that in the future faster and more compact Lane designs will be announced. By exploring different levels of parallelism, one can make a number of trade-offs and choose the appropriate application-driven implementation. Compact implementation of hash functions remains a challenging task in general and hence, we expect more research effort in this direction.

Bibliography

[1] Elena Andreeva. On LANE modes of operation. Technical Report 2008, COSIC, 2008.

[2] Elena Andreeva, Charles Bouillaguet, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. Second preimage attacks on dithered hash functions. In Smart [57], pages 270–288.

[3] Kazumaro Aoki and Helger Lipmaa. Fast implementations of AES candidates. In *AES Candidate Conference*, pages 106–120, 2000.

[4] Gwénolé Ars, Jean-Charles Faugère, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between XL and Gröbner basis algorithms. In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2004.

[5] Thomas Becker and Volker Weispfenning. *Gröbner bases: A computational approach to commutative algebra*, volume 141 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1993.

[6] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *FOCS*, pages 514–523, 1996.

[7] Daniel J. Bernstein. Cache-timing attacks on AES. In *preprint*, 2005.

[8] Daniel J. Bernstein. What output size resists collisions in a XOR of independent expansions? ECRYPT Workshop on Hash Functions, 2007.

[9] Daniel J. Bernstein and Peter Schwabe. New AES software speed records. In *INDOCRYPT*, Lecture Notes in Computer Science. Springer, 2008. to appear. preprint available at `http://eprint.iacr.org/2008/381`.

[10] Eli Biham and Orr Dunkelman. A framework for iterative hash functions — HAIFA. Presented at the second NIST hash workshop (August 24–25), 2006.

[11] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[12] Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[13] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965.

[14] Florent Chabaud and Antoine Joux. Differential collisions in SHA-0. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.

[15] Carlos Cid, Sean Murphy, and Matthew Robshaw. *Algebraic Aspects of the Advanced Encryption Standard*. Springer-Verlag, New York, 2006.

[16] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[17] Intel Corporation. Advanced encryption standard (AES) instructions set. White paper, July 2008.

[18] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *EUROCRYPT*, pages 392–407, 2000.

[19] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Yuliang Zheng, editor, *ASIACRYPT*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002.

[20] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher Square. In Eli Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.

[21] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[22] Ivan Damgård. A design principle for hash functions. In Brassard [12], pages 416–427.

[23] Richard D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, January 1999.

[24] Hans Dobbertin. Cryptanalysis of MD4. In Dieter Gollmann, editor, *FSE*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 1996.

[25] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F4). In *Journal of Pure and Applied Algebra*, pages 61–88. ACM Press, 1999.

[26] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In T. Mora, editor, *International Symposium on Symbolic and Algebraic Computation*, pages 75–83, 2002.

[27] Federal Information Processing Standards Publication FIPS PUB 198. *The Keyed-Hash Message Authentication Code (HMAC)*, 2002.

[28] Pierre-Alain Fouque, Jacques Stern, and Sébastien Zimmer. Cryptanalysis of tweaked versions of SMASH and reparation. In *Selected Areas in Cryptography, SAC 2008*, Lecture Notes in Computer Science. Springer, to appear.

[29] Gezel. `http://rijndael.ece.vt.edu/gezel2/index.php/Main_Page`. The GEZEL Development Environment.

[30] Brian Gladman. Byte oriented AES implementation. Available online at `http://fp.gladman.plus.com/AES/`.

[31] 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.

[32] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[33] Jorge Nakahara Jr., Daniel Santana de Freitas, and Raphael Chung-Wei Phan. New multiset attacks on Rijndael with large blocks. In Ed Dawson and Serge Vaudenay, editors, *Mycrypt*, volume 3715 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 2005.

[34] Jorge Nakahara Jr. and Ivan Carlos Pavao. Impossible-differential attacks on large-block Rijndael. In Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo, and Rene Peralta, editors, *ISC*, volume 4779 of *Lecture Notes in Computer Science*, pages 104–117. Springer, 2007.

[35] 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.

[36] Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *FSE*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 1994.

[37] Lars R. Knudsen and David Wagner. Integral cryptanalysis. In Joan Daemen and Vincent Rijmen, editors, *FSE*, volume 2365 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2002.

[38] Xuejia Lai. Higher order derivatives and differential cryptanalysis. Proc. Symposium on Communication, Coding and Cryptography, in honor of James L. Massey on the occasion of his 60'th birthday, Feb. 10-13, 1994, Monte-Verita, Ascona, Switzerland, 1994.

[39] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In *EUROCRYPT*, pages 17–38, 1991.

[40] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, revised edition edition, 1994.

[41] Stefan Lucks. The saturation attack - a bait for Twofish. In Mitsuru Matsui, editor, *FSE*, volume 2355 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2001.

[42] Mitsuru Matsui. How far can we go on the x64 processors? In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 341–358. Springer, 2006.

[43] Mitsuru Matsui and Sayaka Fukuda. How to maximize software performance of symmetric primitives on Pentium III and 4 processors. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[44] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on Intel Core2 processor. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer, 2007.

[45] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. A systematic evaluation of compact hardware implementations for the Rijndael S-box. In *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 323–333. Springer, 2005.

[46] Ralph C. Merkle. A certified digital signature. In Brassard [12], pages 218–238.

[47] Shoji Miyaguchi, Kazuo Ohta, and Masahiko Iwata. Confirmation that some hash functions are not collision free. In *EUROCRYPT*, pages 326–343, 1990.

[48] National Institue of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.

[49] National Institue of Standards and Technology. Secure hash standard. Federal Information Processing Standards Publication 180-2, 2002.

[50] 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.

[51] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[52] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: A synthetic approach. In Douglas R. Stinson, editor, *CRYPTO*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1993.

[53] Sören Rinne, Thomas Eisenbarth, and Christof Paar. Performance analysis of contemporary light-weight block ciphers on 8-bit microcontrollers. ECRYPT workshop SPEED – Software Performance Enhancement for Encryption and Decryption, 2007.

[54] Phillip Rogaway. Formalizing human ignorance. In Phong Q. Nguyen, editor, *VIETCRYPT*, volume 4341 of *Lecture Notes in Computer Science*, pages 211–228. Springer, 2006.

[55] Phillip Rogaway and John P. Steinberger. Security/efficiency tradeoffs for permutation-based hashing. In Smart [57], pages 220–236.

[56] Bruce Schneier and Doug Whiting. A performance comparison of the five AES finalists. In *AES Candidate Conference*, pages 123–135, 2000.

[57] Nigel P. Smart, editor. *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*. Springer, 2008.

[58] Synopsys. `http://www.synopsys.com/`.

[59] Stefan Tillich and Christoph Herbst. Boosting AES performance on a tiny processor core. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2008.

[60] David Wagner. A generalized birthday problem. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.

[61] Richard Weiss and Nathan L. Binkert. A comparison of AES candidates on the Alpha 21264. In *AES Candidate Conference*, pages 75–81, 2000.

[62] John Worley, Bill Worley, Tom Christian, and Christopher Worley. AES finalists on PA-RISC and IA-64: Implementations & performance. In *AES Candidate Conference*, pages 57–74, 2000.

Appendix A

The constants used in LANE

Table A.1 contains the precomputed values of the constants used in LANE.

Table A.1: The constants used in LANE

$k_0 = \texttt{07fc703d}_x$,	$k_1 = \texttt{d3fe381f}_x$,	$k_2 = \texttt{b9ff1c0e}_x$,	$k_3 = \texttt{5cff8e07}_x$,
$k_4 = \texttt{fe7fc702}_x$,	$k_5 = \texttt{7f3fe381}_x$,	$k_6 = \texttt{ef9ff1c1}_x$,	$k_7 = \texttt{a7cff8e1}_x$,
$k_8 = \texttt{83e7fc71}_x$,	$k_9 = \texttt{91f3fe39}_x$,	$k_{10} = \texttt{98f9ff1d}_x$,	$k_{11} = \texttt{9c7cff8f}_x$,
$k_{12} = \texttt{9e3e7fc6}_x$,	$k_{13} = \texttt{4f1f3fe3}_x$,	$k_{14} = \texttt{f78f9ff0}_x$,	$k_{15} = \texttt{7bc7cff8}_x$,
$k_{16} = \texttt{3de3e7fc}_x$,	$k_{17} = \texttt{1ef1f3fe}_x$,	$k_{18} = \texttt{0f78f9ff}_x$,	$k_{19} = \texttt{d7bc7cfe}_x$,
$k_{20} = \texttt{6bde3e7f}_x$,	$k_{21} = \texttt{e5ef1f3e}_x$,	$k_{22} = \texttt{72f78f9f}_x$,	$k_{23} = \texttt{e97bc7ce}_x$,
$k_{24} = \texttt{74bde3e7}_x$,	$k_{25} = \texttt{ea5ef1f2}_x$,	$k_{26} = \texttt{752f78f9}_x$,	$k_{27} = \texttt{ea97bc7d}_x$,
$k_{28} = \texttt{a54bde3f}_x$,	$k_{29} = \texttt{82a5ef1e}_x$,	$k_{30} = \texttt{4152f78f}_x$,	$k_{31} = \texttt{f0a97bc6}_x$,
$k_{32} = \texttt{7854bde3}_x$,	$k_{33} = \texttt{ec2a5ef0}_x$,	$k_{34} = \texttt{76152f78}_x$,	$k_{35} = \texttt{3b0a97bc}_x$,
$k_{36} = \texttt{1d854bde}_x$,	$k_{37} = \texttt{0ec2a5ef}_x$,	$k_{38} = \texttt{d76152f6}_x$,	$k_{39} = \texttt{6bb0a97b}_x$,
$k_{40} = \texttt{e5d854bc}_x$,	$k_{41} = \texttt{72ec2a5e}_x$,	$k_{42} = \texttt{3976152f}_x$,	$k_{43} = \texttt{ccbb0a96}_x$,
$k_{44} = \texttt{665d854b}_x$,	$k_{45} = \texttt{e32ec2a4}_x$,	$k_{46} = \texttt{71976152}_x$,	$k_{47} = \texttt{38cbb0a9}_x$,
$k_{48} = \texttt{cc65d855}_x$,	$k_{49} = \texttt{b632ec2b}_x$,	$k_{50} = \texttt{8b197614}_x$,	$k_{51} = \texttt{458cbb0a}_x$,
$k_{52} = \texttt{22c65d85}_x$,	$k_{53} = \texttt{c1632ec3}_x$,	$k_{54} = \texttt{b0b19760}_x$,	$k_{55} = \texttt{5858cbb0}_x$,
$k_{56} = \texttt{2c2c65d8}_x$,	$k_{57} = \texttt{161632ec}_x$,	$k_{58} = \texttt{0b0b1976}_x$,	$k_{59} = \texttt{05858cbb}_x$,
$k_{60} = \texttt{d2c2c65c}_x$,	$k_{61} = \texttt{6961632e}_x$,	$k_{62} = \texttt{34b0b197}_x$,	$k_{63} = \texttt{ca5858ca}_x$,
$k_{64} = \texttt{652c2c65}_x$,	$k_{65} = \texttt{e2961633}_x$,	$k_{66} = \texttt{a14b0b18}_x$,	$k_{67} = \texttt{50a5858c}_x$,
$k_{68} = \texttt{2852c2c6}_x$,	$k_{69} = \texttt{14296163}_x$,	$k_{70} = \texttt{da14b0b0}_x$,	$k_{71} = \texttt{6d0a5858}_x$,
$k_{72} = \texttt{36852c2c}_x$,	$k_{73} = \texttt{1b429616}_x$,	$k_{74} = \texttt{0da14b0b}_x$,	$k_{75} = \texttt{d6d0a584}_x$,
$k_{76} = \texttt{6b6852c2}_x$,	$k_{77} = \texttt{35b42961}_x$,	$k_{78} = \texttt{cada14b1}_x$,	$k_{79} = \texttt{b56d0a59}_x$,
$k_{80} = \texttt{8ab6852d}_x$,	$k_{81} = \texttt{955b4297}_x$,	$k_{82} = \texttt{9aada14a}_x$,	$k_{83} = \texttt{4d56d0a5}_x$,
$k_{84} = \texttt{f6ab6853}_x$,	$k_{85} = \texttt{ab55b428}_x$,	$k_{86} = \texttt{55aada14}_x$,	$k_{87} = \texttt{2ad56d0a}_x$,
$k_{88} = \texttt{156ab685}_x$,	$k_{89} = \texttt{dab55b43}_x$,	$k_{90} = \texttt{bd5aada0}_x$,	$k_{91} = \texttt{5ead56d0}_x$,
$k_{92} = \texttt{2f56ab68}_x$,	$k_{93} = \texttt{17ab55b4}_x$,	$k_{94} = \texttt{0bd5aada}_x$,	$k_{95} = \texttt{05ead56d}_x$,
$k_{96} = \texttt{d2f56ab7}_x$,	$k_{97} = \texttt{b97ab55a}_x$,	$k_{98} = \texttt{5cbd5aad}_x$,	$k_{99} = \texttt{fe5ead57}_x$,
$k_{100} = \texttt{af2f56aa}_x$,	$k_{101} = \texttt{5797ab55}_x$,	$k_{102} = \texttt{fbcbd5ab}_x$,	$k_{103} = \texttt{ade5ead4}_x$,
$k_{104} = \texttt{56f2f56a}_x$,	$k_{105} = \texttt{2b797ab5}_x$,	$k_{106} = \texttt{c5bcbd5b}_x$,	$k_{107} = \texttt{b2de5eac}_x$,
$k_{108} = \texttt{596f2f56}_x$,	$k_{109} = \texttt{2cb797ab}_x$,	$k_{110} = \texttt{c65bcbd4}_x$,	$k_{111} = \texttt{632de5ea}_x$,
$k_{112} = \texttt{3196f2f5}_x$,	$k_{113} = \texttt{c8cb797b}_x$,	$k_{114} = \texttt{b465bcbc}_x$,	$k_{115} = \texttt{5a32de5e}_x$,
$k_{116} = \texttt{2d196f2f}_x$,	$k_{117} = \texttt{c68cb796}_x$,	$k_{118} = \texttt{63465bcb}_x$,	$k_{119} = \texttt{e1a32de4}_x$,
$k_{120} = \texttt{70d196f2}_x$,	$k_{121} = \texttt{3868cb79}_x$,	$k_{122} = \texttt{cc3465bd}_x$,	$k_{123} = \texttt{b61a32df}_x$,
$k_{124} = \texttt{8b0d196e}_x$,	$k_{125} = \texttt{45868cb7}_x$,	$k_{126} = \texttt{f2c3465a}_x$,	$k_{127} = \texttt{7961a32d}_x$,
$k_{128} = \texttt{ecb0d197}_x$,	$k_{129} = \texttt{a65868ca}_x$,	$k_{130} = \texttt{532c3465}_x$,	$k_{131} = \texttt{f9961a33}_x$,
$k_{132} = \texttt{accb0d18}_x$,	$k_{133} = \texttt{5665868c}_x$,	$k_{134} = \texttt{2b32c346}_x$,	$k_{135} = \texttt{159961a3}_x$,
$k_{136} = \texttt{daccb0d0}_x$,	$k_{137} = \texttt{6d665868}_x$,	$k_{138} = \texttt{36b32c34}_x$,	$k_{139} = \texttt{1b59961a}_x$,
$k_{140} = \texttt{0daccb0d}_x$,	$k_{141} = \texttt{d6d66587}_x$,	$k_{142} = \texttt{bb6b32c2}_x$,	$k_{143} = \texttt{5db59961}_x$,
$k_{144} = \texttt{fedaccb1}_x$,	$k_{145} = \texttt{af6d6659}_x$,	$k_{146} = \texttt{87b6b32d}_x$,	$k_{147} = \texttt{93db5997}_x$,
$k_{148} = \texttt{99edacca}_x$,	$k_{149} = \texttt{4cf6d665}_x$,	$k_{150} = \texttt{f67b6b33}_x$,	$k_{151} = \texttt{ab3db598}_x$,
$k_{152} = \texttt{559edacc}_x$,	$k_{153} = \texttt{2acf6d66}_x$,	$k_{154} = \texttt{1567b6b3}_x$,	$k_{155} = \texttt{dab3db58}_x$,
$k_{156} = \texttt{6d59edac}_x$,	$k_{157} = \texttt{36acf6d6}_x$,	$k_{158} = \texttt{1b567b6b}_x$,	$k_{159} = \texttt{ddab3db4}_x$,

Table A.1: The constants used in LANE *(continued)*

$k_{160} = \texttt{6ed59eda}_x$,	$k_{161} = \texttt{376acf6d}_x$,	$k_{162} = \texttt{cbb567b7}_x$,	$k_{163} = \texttt{b5dab3da}_x$,
$k_{164} = \texttt{5aed59ed}_x$,	$k_{165} = \texttt{fd76acf7}_x$,	$k_{166} = \texttt{aebb567a}_x$,	$k_{167} = \texttt{575dab3d}_x$,
$k_{168} = \texttt{fbaed59f}_x$,	$k_{169} = \texttt{add76ace}_x$,	$k_{170} = \texttt{56ebb567}_x$,	$k_{171} = \texttt{fb75dab2}_x$,
$k_{172} = \texttt{7dbaed59}_x$,	$k_{173} = \texttt{eedd76ad}_x$,	$k_{174} = \texttt{a76ebb57}_x$,	$k_{175} = \texttt{83b75daa}_x$,
$k_{176} = \texttt{41dbaed5}_x$,	$k_{177} = \texttt{f0edd76b}_x$,	$k_{178} = \texttt{a876ebb4}_x$,	$k_{179} = \texttt{543b75da}_x$,
$k_{180} = \texttt{2a1dbaed}_x$,	$k_{181} = \texttt{c50edd77}_x$,	$k_{182} = \texttt{b2876eba}_x$,	$k_{183} = \texttt{5943b75d}_x$,
$k_{184} = \texttt{fca1dbaf}_x$,	$k_{185} = \texttt{ae50edd6}_x$,	$k_{186} = \texttt{572876eb}_x$,	$k_{187} = \texttt{fb943b74}_x$,
$k_{188} = \texttt{7dca1dba}_x$,	$k_{189} = \texttt{3ee50edd}_x$,	$k_{190} = \texttt{cf72876f}_x$,	$k_{191} = \texttt{b7b943b6}_x$,
$k_{192} = \texttt{5bdca1db}_x$,	$k_{193} = \texttt{fdee50ec}_x$,	$k_{194} = \texttt{7ef72876}_x$,	$k_{195} = \texttt{3f7b943b}_x$,
$k_{196} = \texttt{cfbdca1c}_x$,	$k_{197} = \texttt{67dee50e}_x$,	$k_{198} = \texttt{33ef7287}_x$,	$k_{199} = \texttt{c9f7b942}_x$,
$k_{200} = \texttt{64fbdca1}_x$,	$k_{201} = \texttt{e27dee51}_x$,	$k_{202} = \texttt{a13ef729}_x$,	$k_{203} = \texttt{809f7b95}_x$,
$k_{204} = \texttt{904fbdcb}_x$,	$k_{205} = \texttt{9827dee4}_x$,	$k_{206} = \texttt{4c13ef72}_x$,	$k_{207} = \texttt{2609f7b9}_x$,
$k_{208} = \texttt{c304fbdd}_x$,	$k_{209} = \texttt{b1827def}_x$,	$k_{210} = \texttt{88c13ef6}_x$,	$k_{211} = \texttt{44609f7b}_x$,
$k_{212} = \texttt{f2304fbc}_x$,	$k_{213} = \texttt{791827de}_x$,	$k_{214} = \texttt{3c8c13ef}_x$,	$k_{215} = \texttt{ce4609f6}_x$,
$k_{216} = \texttt{672304fb}_x$,	$k_{217} = \texttt{e391827c}_x$,	$k_{218} = \texttt{71c8c13e}_x$,	$k_{219} = \texttt{38e4609f}_x$,
$k_{220} = \texttt{cc72304e}_x$,	$k_{221} = \texttt{66391827}_x$,	$k_{222} = \texttt{e31c8c12}_x$,	$k_{223} = \texttt{718e4609}_x$,
$k_{224} = \texttt{e8c72305}_x$,	$k_{225} = \texttt{a4639183}_x$,	$k_{226} = \texttt{8231c8c0}_x$,	$k_{227} = \texttt{4118e460}_x$,
$k_{228} = \texttt{208c7230}_x$,	$k_{229} = \texttt{10463918}_x$,	$k_{230} = \texttt{08231c8c}_x$,	$k_{231} = \texttt{04118e46}_x$,
$k_{232} = \texttt{0208c723}_x$,	$k_{233} = \texttt{d1046390}_x$,	$k_{234} = \texttt{688231c8}_x$,	$k_{235} = \texttt{344118e4}_x$,
$k_{236} = \texttt{1a208c72}_x$,	$k_{237} = \texttt{0d104639}_x$,	$k_{238} = \texttt{d688231d}_x$,	$k_{239} = \texttt{bb44118f}_x$,
$k_{240} = \texttt{8da208c6}_x$,	$k_{241} = \texttt{46d10463}_x$,	$k_{242} = \texttt{f3688230}_x$,	$k_{243} = \texttt{79b44118}_x$,
$k_{244} = \texttt{3cda208c}_x$,	$k_{245} = \texttt{1e6d1046}_x$,	$k_{246} = \texttt{0f368823}_x$,	$k_{247} = \texttt{d79b4410}_x$,
$k_{248} = \texttt{6bcda208}_x$,	$k_{249} = \texttt{35e6d104}_x$,	$k_{250} = \texttt{1af36882}_x$,	$k_{251} = \texttt{0d79b441}_x$,
$k_{252} = \texttt{d6bcda21}_x$,	$k_{253} = \texttt{bb5e6d11}_x$,	$k_{254} = \texttt{8daf3689}_x$,	$k_{255} = \texttt{96d79b45}_x$,
$k_{256} = \texttt{9b6bcda3}_x$,	$k_{257} = \texttt{9db5e6d0}_x$,	$k_{258} = \texttt{4edaf368}_x$,	$k_{259} = \texttt{276d79b4}_x$,
$k_{260} = \texttt{13b6bcda}_x$,	$k_{261} = \texttt{09db5e6d}_x$,	$k_{262} = \texttt{d4edaf37}_x$,	$k_{263} = \texttt{ba76d79a}_x$,
$k_{264} = \texttt{5d3b6bcd}_x$,	$k_{265} = \texttt{fe9db5e7}_x$,	$k_{266} = \texttt{af4edaf2}_x$,	$k_{267} = \texttt{57a76d79}_x$,
$k_{268} = \texttt{fbd3b6bd}_x$,	$k_{269} = \texttt{ade9db5f}_x$,	$k_{270} = \texttt{86f4edae}_x$,	$k_{271} = \texttt{437a76d7}_x$,
$k_{272} = \texttt{f1bd3b6a}_x$,	$k_{273} = \texttt{78de9db5}_x$,	$k_{274} = \texttt{ec6f4edb}_x$,	$k_{275} = \texttt{a637a76c}_x$,
$k_{276} = \texttt{531bd3b6}_x$,	$k_{277} = \texttt{298de9db}_x$,	$k_{278} = \texttt{c4c6f4ec}_x$,	$k_{279} = \texttt{62637a76}_x$,
$k_{280} = \texttt{3131bd3b}_x$,	$k_{281} = \texttt{c898de9c}_x$,	$k_{282} = \texttt{644c6f4e}_x$,	$k_{283} = \texttt{322637a7}_x$,
$k_{284} = \texttt{c9131bd2}_x$,	$k_{285} = \texttt{64898de9}_x$,	$k_{286} = \texttt{e244c6f5}_x$,	$k_{287} = \texttt{a122637b}_x$,
$k_{288} = \texttt{809131bc}_x$,	$k_{289} = \texttt{404898de}_x$,	$k_{290} = \texttt{20244c6f}_x$,	$k_{291} = \texttt{c0122636}_x$,
$k_{292} = \texttt{6009131b}_x$,	$k_{293} = \texttt{e004898c}_x$,	$k_{294} = \texttt{700244c6}_x$,	$k_{295} = \texttt{38012263}_x$,
$k_{296} = \texttt{cc009130}_x$,	$k_{297} = \texttt{66004898}_x$,	$k_{298} = \texttt{3300244c}_x$,	$k_{299} = \texttt{19801226}_x$,
$k_{300} = \texttt{0cc00913}_x$,	$k_{301} = \texttt{d6600488}_x$,	$k_{302} = \texttt{6b300244}_x$,	$k_{303} = \texttt{35980122}_x$,
$k_{304} = \texttt{1acc0091}_x$,	$k_{305} = \texttt{dd660049}_x$,	$k_{306} = \texttt{beb30025}_x$,	$k_{307} = \texttt{8f598013}_x$,
$k_{308} = \texttt{97acc008}_x$,	$k_{309} = \texttt{4bd66004}_x$,	$k_{310} = \texttt{25eb3002}_x$,	$k_{311} = \texttt{12f59801}_x$,
$k_{312} = \texttt{d97acc01}_x$,	$k_{313} = \texttt{bcbd6601}_x$,	$k_{314} = \texttt{8e5eb301}_x$,	$k_{315} = \texttt{972f5981}_x$,
$k_{316} = \texttt{9b97acc1}_x$,	$k_{317} = \texttt{9dcbd661}_x$,	$k_{318} = \texttt{9ee5eb31}_x$,	$k_{319} = \texttt{9f72f599}_x$,
$k_{320} = \texttt{9fb97acd}_x$,	$k_{321} = \texttt{9fdcbd67}_x$,	$k_{322} = \texttt{9fee5eb2}_x$,	$k_{323} = \texttt{4ff72f59}_x$,
$k_{324} = \texttt{f7fb97ad}_x$,	$k_{325} = \texttt{abfdcbd7}_x$,	$k_{326} = \texttt{85fee5ea}_x$,	$k_{327} = \texttt{42ff72f5}_x$,
$k_{328} = \texttt{f17fb97b}_x$,	$k_{329} = \texttt{a8bfdcbc}_x$,	$k_{330} = \texttt{545fee5e}_x$,	$k_{331} = \texttt{2a2ff72f}_x$,
$k_{332} = \texttt{c517fb96}_x$,	$k_{333} = \texttt{628bfdcb}_x$,	$k_{334} = \texttt{e145fee4}_x$,	$k_{335} = \texttt{70a2ff72}_x$,
$k_{336} = \texttt{38517fb9}_x$,	$k_{337} = \texttt{cc28bfdd}_x$,	$k_{338} = \texttt{b6145fef}_x$,	$k_{339} = \texttt{8b0a2ff6}_x$,
$k_{340} = \texttt{458517fb}_x$,	$k_{341} = \texttt{f2c28bfc}_x$,	$k_{342} = \texttt{796145fe}_x$,	$k_{343} = \texttt{3cb0a2ff}_x$,
$k_{344} = \texttt{ce58517e}_x$,	$k_{345} = \texttt{672c28bf}_x$,	$k_{346} = \texttt{e396145e}_x$,	$k_{347} = \texttt{71cb0a2f}_x$,
$k_{348} = \texttt{e8e58516}_x$,	$k_{349} = \texttt{7472c28b}_x$,	$k_{350} = \texttt{ea396144}_x$,	$k_{351} = \texttt{751cb0a2}_x$,
$k_{352} = \texttt{3a8e5851}_x$,	$k_{353} = \texttt{cd472c29}_x$,	$k_{354} = \texttt{b6a39615}_x$,	$k_{355} = \texttt{8b51cb0b}_x$,
$k_{356} = \texttt{95a8e584}_x$,	$k_{357} = \texttt{4ad472c2}_x$,	$k_{358} = \texttt{256a3961}_x$,	$k_{359} = \texttt{c2b51cb1}_x$,
$k_{360} = \texttt{b15a8e59}_x$,	$k_{361} = \texttt{88ad472d}_x$,	$k_{362} = \texttt{9456a397}_x$,	$k_{363} = \texttt{9a2b51ca}_x$,
$k_{364} = \texttt{4d15a8e5}_x$,	$k_{365} = \texttt{f68ad473}_x$,	$k_{366} = \texttt{ab456a38}_x$,	$k_{367} = \texttt{55a2b51c}_x$,
$k_{368} = \texttt{2ad15a8e}_x$,	$k_{369} = \texttt{1568ad47}_x$,	$k_{370} = \texttt{dab456a2}_x$,	$k_{371} = \texttt{6d5a2b51}_x$,
$k_{372} = \texttt{e6ad15a9}_x$,	$k_{373} = \texttt{a3568ad5}_x$,	$k_{374} = \texttt{81ab456b}_x$,	$k_{375} = \texttt{90d5a2b4}_x$,
$k_{376} = \texttt{486ad15a}_x$,	$k_{377} = \texttt{243568ad}_x$,	$k_{378} = \texttt{c21ab457}_x$,	$k_{379} = \texttt{b10d5a2a}_x$,
$k_{380} = \texttt{5886ad15}_x$,	$k_{381} = \texttt{fc43568b}_x$,	$k_{382} = \texttt{ae21ab44}_x$,	$k_{383} = \texttt{5710d5a2}_x$,
$k_{384} = \texttt{2b886ad1}_x$,	$k_{385} = \texttt{c5c43569}_x$,	$k_{386} = \texttt{b2e21ab5}_x$,	$k_{387} = \texttt{89710d5b}_x$,
$k_{388} = \texttt{94b886ac}_x$,	$k_{389} = \texttt{4a5c4356}_x$,	$k_{390} = \texttt{252e21ab}_x$,	$k_{391} = \texttt{c29710d4}_x$,
$k_{392} = \texttt{614b886a}_x$,	$k_{393} = \texttt{30a5c435}_x$,	$k_{394} = \texttt{c852e21b}_x$,	$k_{395} = \texttt{b429710c}_x$,
$k_{396} = \texttt{5a14b886}_x$,	$k_{397} = \texttt{2d0a5c43}_x$,	$k_{398} = \texttt{c6852e20}_x$,	$k_{399} = \texttt{63429710}_x$,

Table A.1: The constants used in LANE *(continued)*

$k_{400} = \texttt{31a14b88}_x$,	$k_{401} = \texttt{18d0a5c4}_x$,	$k_{402} = \texttt{0c6852e2}_x$,	$k_{403} = \texttt{06342971}_x$,
$k_{404} = \texttt{d31a14b9}_x$,	$k_{405} = \texttt{b98d0a5d}_x$,	$k_{406} = \texttt{8cc6852f}_x$,	$k_{407} = \texttt{96634296}_x$,
$k_{408} = \texttt{4b31a14b}_x$,	$k_{409} = \texttt{f598d0a4}_x$,	$k_{410} = \texttt{7acc6852}_x$,	$k_{411} = \texttt{3d663429}_x$,
$k_{412} = \texttt{ceb31a15}_x$,	$k_{413} = \texttt{b7598d0b}_x$,	$k_{414} = \texttt{8bacc684}_x$,	$k_{415} = \texttt{45d66342}_x$,
$k_{416} = \texttt{22eb31a1}_x$,	$k_{417} = \texttt{c17598d1}_x$,	$k_{418} = \texttt{b0bacc69}_x$,	$k_{419} = \texttt{885d6635}_x$,
$k_{420} = \texttt{942eb31b}_x$,	$k_{421} = \texttt{9a17598c}_x$,	$k_{422} = \texttt{4d0bacc6}_x$,	$k_{423} = \texttt{2685d663}_x$,
$k_{424} = \texttt{c342eb30}_x$,	$k_{425} = \texttt{61a17598}_x$,	$k_{426} = \texttt{30d0bacc}_x$,	$k_{427} = \texttt{18685d66}_x$,
$k_{428} = \texttt{0c342eb3}_x$,	$k_{429} = \texttt{d61a1758}_x$,	$k_{430} = \texttt{6b0d0bac}_x$,	$k_{431} = \texttt{358685d6}_x$,
$k_{432} = \texttt{1ac342eb}_x$,	$k_{433} = \texttt{dd61a174}_x$,	$k_{434} = \texttt{6eb0d0ba}_x$,	$k_{435} = \texttt{3758685d}_x$,
$k_{436} = \texttt{cbac342f}_x$,	$k_{437} = \texttt{b5d61a16}_x$,	$k_{438} = \texttt{5aeb0d0b}_x$,	$k_{439} = \texttt{fd758684}_x$,
$k_{440} = \texttt{7ebac342}_x$,	$k_{441} = \texttt{3f5d61a1}_x$,	$k_{442} = \texttt{cfaeb0d1}_x$,	$k_{443} = \texttt{b7d75869}_x$,
$k_{444} = \texttt{8bebac35}_x$,	$k_{445} = \texttt{95f5d61b}_x$,	$k_{446} = \texttt{9afaeb0c}_x$,	$k_{447} = \texttt{4d7d7586}_x$,
$k_{448} = \texttt{26bebac3}_x$,	$k_{449} = \texttt{c35f5d60}_x$,	$k_{450} = \texttt{61afaeb0}_x$,	$k_{451} = \texttt{30d7d758}_x$,
$k_{452} = \texttt{186bebac}_x$,	$k_{453} = \texttt{0c35f5d6}_x$,	$k_{454} = \texttt{061afaeb}_x$,	$k_{455} = \texttt{d30d7d74}_x$,
$k_{456} = \texttt{6986beba}_x$,	$k_{457} = \texttt{34c35f5d}_x$,	$k_{458} = \texttt{ca61afaf}_x$,	$k_{459} = \texttt{b530d7d6}_x$,
$k_{460} = \texttt{5a986beb}_x$,	$k_{461} = \texttt{fd4c35f4}_x$,	$k_{462} = \texttt{7ea61afa}_x$,	$k_{463} = \texttt{3f530d7d}_x$,
$k_{464} = \texttt{cfa986bf}_x$,	$k_{465} = \texttt{b7d4c35e}_x$,	$k_{466} = \texttt{5bea61af}_x$,	$k_{467} = \texttt{fdf530d6}_x$,
$k_{468} = \texttt{7efa986b}_x$,	$k_{469} = \texttt{ef7d4c34}_x$,	$k_{470} = \texttt{77bea61a}_x$,	$k_{471} = \texttt{3bdf530d}_x$,
$k_{472} = \texttt{cdefa987}_x$,	$k_{473} = \texttt{b6f7d4c2}_x$,	$k_{474} = \texttt{5b7bea61}_x$,	$k_{475} = \texttt{fdbdf531}_x$,
$k_{476} = \texttt{aedefa99}_x$,	$k_{477} = \texttt{876f7d4d}_x$,	$k_{478} = \texttt{93b7bea7}_x$,	$k_{479} = \texttt{99dbdf52}_x$,
$k_{480} = \texttt{4cedefa9}_x$,	$k_{481} = \texttt{f676f7d5}_x$,	$k_{482} = \texttt{ab3b7beb}_x$,	$k_{483} = \texttt{859dbdf4}_x$,
$k_{484} = \texttt{42cedefa}_x$,	$k_{485} = \texttt{21676f7d}_x$,	$k_{486} = \texttt{c0b3b7bf}_x$,	$k_{487} = \texttt{b059dbde}_x$,
$k_{488} = \texttt{582cedef}_x$,	$k_{489} = \texttt{fc1676f6}_x$,	$k_{490} = \texttt{7e0b3b7b}_x$,	$k_{491} = \texttt{ef059dbc}_x$,
$k_{492} = \texttt{7782cede}_x$,	$k_{493} = \texttt{3bc1676f}_x$,	$k_{494} = \texttt{cde0b3b6}_x$,	$k_{495} = \texttt{66f059db}_x$,
$k_{496} = \texttt{e3782cec}_x$,	$k_{497} = \texttt{71bc1676}_x$,	$k_{498} = \texttt{38de0b3b}_x$,	$k_{499} = \texttt{cc6f059c}_x$,
$k_{500} = \texttt{663782ce}_x$,	$k_{501} = \texttt{331bc167}_x$,	$k_{502} = \texttt{c98de0b2}_x$,	$k_{503} = \texttt{64c6f059}_x$,
$k_{504} = \texttt{e263782d}_x$,	$k_{505} = \texttt{a131bc17}_x$,	$k_{506} = \texttt{8098de0a}_x$,	$k_{507} = \texttt{404c6f05}_x$,
$k_{508} = \texttt{f0263783}_x$,	$k_{509} = \texttt{a8131bc0}_x$,	$k_{510} = \texttt{54098de0}_x$,	$k_{511} = \texttt{2a04c6f0}_x$,
$k_{512} = \texttt{15026378}_x$,	$k_{513} = \texttt{0a8131bc}_x$,	$k_{514} = \texttt{054098de}_x$,	$k_{515} = \texttt{02a04c6f}_x$,
$k_{516} = \texttt{d1502636}_x$,	$k_{517} = \texttt{68a8131b}_x$,	$k_{518} = \texttt{e454098c}_x$,	$k_{519} = \texttt{722a04c6}_x$,
$k_{520} = \texttt{39150263}_x$,	$k_{521} = \texttt{cc8a8130}_x$,	$k_{522} = \texttt{66454098}_x$,	$k_{523} = \texttt{3322a04c}_x$,
$k_{524} = \texttt{19915026}_x$,	$k_{525} = \texttt{0cc8a813}_x$,	$k_{526} = \texttt{d6645408}_x$,	$k_{527} = \texttt{6b322a04}_x$,
$k_{528} = \texttt{35991502}_x$,	$k_{529} = \texttt{1acc8a81}_x$,	$k_{530} = \texttt{dd664541}_x$,	$k_{531} = \texttt{beb322a1}_x$,
$k_{532} = \texttt{8f599151}_x$,	$k_{533} = \texttt{97acc8a9}_x$,	$k_{534} = \texttt{9bd66455}_x$,	$k_{535} = \texttt{9deb322b}_x$,
$k_{536} = \texttt{9ef59914}_x$,	$k_{537} = \texttt{4f7acc8a}_x$,	$k_{538} = \texttt{27bd6645}_x$,	$k_{539} = \texttt{c3deb323}_x$,
$k_{540} = \texttt{b1ef5990}_x$,	$k_{541} = \texttt{58f7acc8}_x$,	$k_{542} = \texttt{2c7bd664}_x$,	$k_{543} = \texttt{163deb32}_x$,
$k_{544} = \texttt{0b1ef599}_x$,	$k_{545} = \texttt{d58f7acd}_x$,	$k_{546} = \texttt{bac7bd67}_x$,	$k_{547} = \texttt{8d63deb2}_x$,
$k_{548} = \texttt{46b1ef59}_x$,	$k_{549} = \texttt{f358f7ad}_x$,	$k_{550} = \texttt{a9ac7bd7}_x$,	$k_{551} = \texttt{84d63dea}_x$,
$k_{552} = \texttt{426b1ef5}_x$,	$k_{553} = \texttt{f1358f7b}_x$,	$k_{554} = \texttt{a89ac7bc}_x$,	$k_{555} = \texttt{544d63de}_x$,
$k_{556} = \texttt{2a26b1ef}_x$,	$k_{557} = \texttt{c51358f6}_x$,	$k_{558} = \texttt{6289ac7b}_x$,	$k_{559} = \texttt{e144d63c}_x$,
$k_{560} = \texttt{70a26b1e}_x$,	$k_{561} = \texttt{3851358f}_x$,	$k_{562} = \texttt{cc289ac6}_x$,	$k_{563} = \texttt{66144d63}_x$,
$k_{564} = \texttt{e30a26b0}_x$,	$k_{565} = \texttt{71851358}_x$,	$k_{566} = \texttt{38c289ac}_x$,	$k_{567} = \texttt{1c6144d6}_x$,
$k_{568} = \texttt{0e30a26b}_x$,	$k_{569} = \texttt{d7185134}_x$,	$k_{570} = \texttt{6b8c289a}_x$,	$k_{571} = \texttt{35c6144d}_x$,
$k_{572} = \texttt{cae30a27}_x$,	$k_{573} = \texttt{b5718512}_x$,	$k_{574} = \texttt{5ab8c289}_x$,	$k_{575} = \texttt{fd5c6145}_x$,
$k_{576} = \texttt{aeae30a3}_x$,	$k_{577} = \texttt{87571850}_x$,	$k_{578} = \texttt{43ab8c28}_x$,	$k_{579} = \texttt{21d5c614}_x$,
$k_{580} = \texttt{10eae30a}_x$,	$k_{581} = \texttt{08757185}_x$,	$k_{582} = \texttt{d43ab8c3}_x$,	$k_{583} = \texttt{ba1d5c60}_x$,
$k_{584} = \texttt{5d0eae30}_x$,	$k_{585} = \texttt{2e875718}_x$,	$k_{586} = \texttt{1743ab8c}_x$,	$k_{587} = \texttt{0ba1d5c6}_x$,
$k_{588} = \texttt{05d0eae3}_x$,	$k_{589} = \texttt{d2e87570}_x$,	$k_{590} = \texttt{69743ab8}_x$,	$k_{591} = \texttt{34ba1d5c}_x$,
$k_{592} = \texttt{1a5d0eae}_x$,	$k_{593} = \texttt{0d2e8757}_x$,	$k_{594} = \texttt{d69743aa}_x$,	$k_{595} = \texttt{6b4ba1d5}_x$,
$k_{596} = \texttt{e5a5d0eb}_x$,	$k_{597} = \texttt{a2d2e874}_x$,	$k_{598} = \texttt{5169743a}_x$,	$k_{599} = \texttt{28b4ba1d}_x$,
$k_{600} = \texttt{c45a5d0f}_x$,	$k_{601} = \texttt{b22d2e86}_x$,	$k_{602} = \texttt{59169743}_x$,	$k_{603} = \texttt{fc8b4ba0}_x$,
$k_{604} = \texttt{7e45a5d0}_x$,	$k_{605} = \texttt{3f22d2e8}_x$,	$k_{606} = \texttt{1f916974}_x$,	$k_{607} = \texttt{0fc8b4ba}_x$,
$k_{608} = \texttt{07e45a5d}_x$,	$k_{609} = \texttt{d3f22d2f}_x$,	$k_{610} = \texttt{b9f91696}_x$,	$k_{611} = \texttt{5cfc8b4b}_x$,
$k_{612} = \texttt{fe7e45a4}_x$,	$k_{613} = \texttt{7f3f22d2}_x$,	$k_{614} = \texttt{3f9f9169}_x$,	$k_{615} = \texttt{cfcfc8b5}_x$,
$k_{616} = \texttt{b7e7e45b}_x$,	$k_{617} = \texttt{8bf3f22c}_x$,	$k_{618} = \texttt{45f9f916}_x$,	$k_{619} = \texttt{22fcfc8b}_x$,
$k_{620} = \texttt{c17e7e44}_x$,	$k_{621} = \texttt{60bf3f22}_x$,	$k_{622} = \texttt{305f9f91}_x$,	$k_{623} = \texttt{c82fcfc9}_x$,
$k_{624} = \texttt{b417e7e5}_x$,	$k_{625} = \texttt{8a0bf3f3}_x$,	$k_{626} = \texttt{9505f9f8}_x$,	$k_{627} = \texttt{4a82fcfc}_x$,
$k_{628} = \texttt{25417e7e}_x$,	$k_{629} = \texttt{12a0bf3f}_x$,	$k_{630} = \texttt{d9505f9e}_x$,	$k_{631} = \texttt{6ca82fcf}_x$,
$k_{632} = \texttt{e65417e6}_x$,	$k_{633} = \texttt{732a0bf3}_x$,	$k_{634} = \texttt{e99505f8}_x$,	$k_{635} = \texttt{74ca82fc}_x$,
$k_{636} = \texttt{3a65417e}_x$,	$k_{637} = \texttt{1d32a0bf}_x$,	$k_{638} = \texttt{de99505e}_x$,	$k_{639} = \texttt{6f4ca82f}_x$,

Table A.1: The constants used in LANE *(continued)*

$k_{640} = \texttt{e7a65416}_x$, $k_{641} = \texttt{73d32a0b}_x$, $k_{642} = \texttt{e9e99504}_x$, $k_{643} = \texttt{74f4ca82}_x$,
$k_{644} = \texttt{3a7a6541}_x$, $k_{645} = \texttt{cd3d32a1}_x$, $k_{646} = \texttt{b69e9951}_x$, $k_{647} = \texttt{8b4f4ca9}_x$,
$k_{648} = \texttt{95a7a655}_x$, $k_{649} = \texttt{9ad3d32b}_x$, $k_{650} = \texttt{9d69e994}_x$, $k_{651} = \texttt{4eb4f4ca}_x$,
$k_{652} = \texttt{275a7a65}_x$, $k_{653} = \texttt{c3ad3d33}_x$, $k_{654} = \texttt{b1d69e98}_x$, $k_{655} = \texttt{58eb4f4c}_x$,
$k_{656} = \texttt{2c75a7a6}_x$, $k_{657} = \texttt{163ad3d3}_x$, $k_{658} = \texttt{db1d69e8}_x$, $k_{659} = \texttt{6d8eb4f4}_x$,
$k_{660} = \texttt{36c75a7a}_x$, $k_{661} = \texttt{1b63ad3d}_x$, $k_{662} = \texttt{ddb1d69f}_x$, $k_{663} = \texttt{bed8eb4e}_x$,
$k_{664} = \texttt{5f6c75a7}_x$, $k_{665} = \texttt{ffb63ad2}_x$, $k_{666} = \texttt{7fdb1d69}_x$, $k_{667} = \texttt{efed8eb5}_x$,
$k_{668} = \texttt{a7f6c75b}_x$, $k_{669} = \texttt{83fb63ac}_x$, $k_{670} = \texttt{41fdb1d6}_x$, $k_{671} = \texttt{20fed8eb}_x$,
$k_{672} = \texttt{c07f6c74}_x$, $k_{673} = \texttt{603fb63a}_x$, $k_{674} = \texttt{301fdb1d}_x$, $k_{675} = \texttt{c80fed8f}_x$,
$k_{676} = \texttt{b407f6c6}_x$, $k_{677} = \texttt{5a03fb63}_x$, $k_{678} = \texttt{fd01fdb0}_x$, $k_{679} = \texttt{7e80fed8}_x$,
$k_{680} = \texttt{3f407f6c}_x$, $k_{681} = \texttt{1fa03fb6}_x$, $k_{682} = \texttt{0fd01fdb}_x$, $k_{683} = \texttt{d7e80fec}_x$,
$k_{684} = \texttt{6bf407f6}_x$, $k_{685} = \texttt{35fa03fb}_x$, $k_{686} = \texttt{cafd01fc}_x$, $k_{687} = \texttt{657e80fe}_x$,
$k_{688} = \texttt{32bf407f}_x$, $k_{689} = \texttt{c95fa03e}_x$, $k_{690} = \texttt{64afd01f}_x$, $k_{691} = \texttt{e257e80e}_x$,
$k_{692} = \texttt{712bf407}_x$, $k_{693} = \texttt{e895fa02}_x$, $k_{694} = \texttt{744afd01}_x$, $k_{695} = \texttt{ea257e81}_x$,
$k_{696} = \texttt{a512bf41}_x$, $k_{697} = \texttt{82895fa1}_x$, $k_{698} = \texttt{9144afd1}_x$, $k_{699} = \texttt{98a257e9}_x$,
$k_{700} = \texttt{9c512bf5}_x$, $k_{701} = \texttt{9e2895fb}_x$, $k_{702} = \texttt{9f144afc}_x$, $k_{703} = \texttt{4f8a257e}_x$,
$k_{704} = \texttt{27c512bf}_x$, $k_{705} = \texttt{c3e2895e}_x$, $k_{706} = \texttt{61f144af}_x$, $k_{707} = \texttt{e0f8a256}_x$,
$k_{708} = \texttt{707c512b}_x$, $k_{709} = \texttt{e83e2894}_x$, $k_{710} = \texttt{741f144a}_x$, $k_{711} = \texttt{3a0f8a25}_x$,
$k_{712} = \texttt{cd07c513}_x$, $k_{713} = \texttt{b683e288}_x$, $k_{714} = \texttt{5b41f144}_x$, $k_{715} = \texttt{2da0f8a2}_x$,
$k_{716} = \texttt{16d07c51}_x$, $k_{717} = \texttt{db683e29}_x$, $k_{718} = \texttt{bdb41f15}_x$, $k_{719} = \texttt{8eda0f8b}_x$,
$k_{720} = \texttt{976d07c4}_x$, $k_{721} = \texttt{4bb683e2}_x$, $k_{722} = \texttt{25db41f1}_x$, $k_{723} = \texttt{c2eda0f9}_x$,
$k_{724} = \texttt{b176d07d}_x$, $k_{725} = \texttt{88bb683f}_x$, $k_{726} = \texttt{945db41e}_x$, $k_{727} = \texttt{4a2eda0f}_x$,
$k_{728} = \texttt{f5176d06}_x$, $k_{729} = \texttt{7a8bb683}_x$, $k_{730} = \texttt{ed45db40}_x$, $k_{731} = \texttt{76a2eda0}_x$,
$k_{732} = \texttt{3b5176d0}_x$, $k_{733} = \texttt{1da8bb68}_x$, $k_{734} = \texttt{0ed45db4}_x$, $k_{735} = \texttt{076a2eda}_x$,
$k_{736} = \texttt{03b5176d}_x$, $k_{737} = \texttt{d1da8bb7}_x$, $k_{738} = \texttt{b8ed45da}_x$, $k_{739} = \texttt{5c76a2ed}_x$,
$k_{740} = \texttt{fe3b5177}_x$, $k_{741} = \texttt{af1da8ba}_x$, $k_{742} = \texttt{578ed45d}_x$, $k_{743} = \texttt{fbc76a2f}_x$,
$k_{744} = \texttt{ade3b516}_x$, $k_{745} = \texttt{56f1da8b}_x$, $k_{746} = \texttt{fb78ed44}_x$, $k_{747} = \texttt{7dbc76a2}_x$,
$k_{748} = \texttt{3ede3b51}_x$, $k_{749} = \texttt{cf6f1da9}_x$, $k_{750} = \texttt{b7b78ed5}_x$, $k_{751} = \texttt{8bdbc76b}_x$,
$k_{752} = \texttt{95ede3b4}_x$, $k_{753} = \texttt{4af6f1da}_x$, $k_{754} = \texttt{257b78ed}_x$, $k_{755} = \texttt{c2bdbc77}_x$,
$k_{756} = \texttt{b15ede3a}_x$, $k_{757} = \texttt{58af6f1d}_x$, $k_{758} = \texttt{fc57b78f}_x$, $k_{759} = \texttt{ae2bdbc6}_x$,
$k_{760} = \texttt{5715ede3}_x$, $k_{761} = \texttt{fb8af6f0}_x$, $k_{762} = \texttt{7dc57b78}_x$, $k_{763} = \texttt{3ee2bdbc}_x$,
$k_{764} = \texttt{1f715ede}_x$, $k_{765} = \texttt{0fb8af6f}_x$, $k_{766} = \texttt{d7dc57b6}_x$, $k_{767} = \texttt{6bee2bdb}_x$.