NORX

v1

Designers / Submitters:
Jean-Philippe Aumasson
Philipp Jovanovic
Samuel Neves

contact@norx.io

March 19, 2014

Contents

1 Introduction 3

2 Specification 5
- 2.1 Parameters and interface . . . 5
- 2.2 Naming conventions . . . 6
- 2.3 Instances . . . 6
- 2.4 Layout overview . . . 6
- 2.5 The round function F . . . 7
- 2.6 Encrypting and authenticating a message . . . 8
  - 2.6.1 Structure . . . 9
  - 2.6.2 Padding . . . 9
  - 2.6.3 Domain separation . . . 10
  - 2.6.4 Initialisation . . . 10
  - 2.6.5 Message processing . . . 11
  - 2.6.6 Tag generation . . . 13
- 2.7 Decrypting a ciphertext and verifying a tag . . . 13
  - 2.7.1 Structure . . . 13
  - 2.7.2 Padding . . . 14
  - 2.7.3 Domain separation . . . 14
  - 2.7.4 Initialisation . . . 14
  - 2.7.5 Message processing . . . 14
  - 2.7.6 Tag generation . . . 14
  - 2.7.7 Tag verification . . . 14
- 2.8 Datagrams . . . 15
  - 2.8.1 Fixed parameters . . . 15
  - 2.8.2 Variable parameters . . . 15

3 Security goals 18

4 Features 19
- 4.1 List of characteristics . . . 19
- 4.2 Recommended parameter sets . . . 20
- 4.3 Performance . . . 21
  - 4.3.1 Generalities . . . 21
  - 4.3.2 Software . . . 22
  - 4.3.3 Hardware . . . 24

5 Design rationale 26
- 5.1 The parallel duplex construction . . . 26
- 5.2 The G function . . . 26
- 5.3 The F function . . . 28
- 5.4 Selection of rotation offsets . . . 29

5.5 Number of rounds . . . 30
5.6 Selection of constants . . . 31
5.7 The padding rule . . . 31
5.8 Absence of backdoors . . . 32

6 Security analysis **33**
6.1 Differential cryptanalysis . . . 33
6.1.1 Notation . . . 33
6.1.2 Differential properties of G . . . 34
6.1.3 Simple differentials . . . 35
6.1.4 Impossible differentials . . . 37
6.2 Algebraic cryptanalysis . . . 39
6.3 Other attacks . . . 39
6.3.1 Fixed points . . . 39
6.3.2 Slide attacks . . . 40
6.3.3 Rotational cryptanalysis . . . 40

7 Intellectual property **41**

8 Consent **42**

9 Acknowledgements **43**

Bibliography **44**

A Test Vectors and Intermediate Values **48**
A.1 Traces for G . . . 48
A.2 Traces for F . . . 50
A.3 Full AEAD computations . . . 52
A.3.1 Values for NORX32 . . . 52
A.3.2 Values for NORX64 . . . 53

B Miscellaneous **57**
B.1 Diffusion statistics for inverse round functions . . . 57
B.2 Addenda to cryptanalysis . . . 57
B.2.1 Visualisation of differentials for G_1 . . . 57
B.2.2 Impossible differential cryptanalysis . . . 57

1 Introduction

The *Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR)* [3] invites cryptographers to submit authenticated encryption schemes supporting associated data (AEAD) [43], that offer advantages over AES-GCM [29, 40] and are suitable for widespread adoption.

NORX[1] is our candidate for CAESAR. It is a novel authenticated encryption scheme with associated data supporting an arbitrary parallelism degree, based on ARX primitives yet not using modular additions. NORX has a unique parallel architecture based on the monkeyDuplex construction [17, 20], where the parallelism degree and tag size can be tuned arbitrarily. An original domain separation scheme allows simple processing of header/payload/trailer data. NORX was optimized for efficiency in both software and hardware, with a SIMD-friendly core, almost byte-aligned rotations, no secret-dependent memory lookups, and only bitwise operations. The NORX core is inspired by the ARX primitive ChaCha [14], however it replaces integer addition with the approximation $a \oplus b \oplus (a \wedge b) \ll 1$[2]. This simplifies cryptanalysis and improves hardware efficiency. Furthermore, NORX specifies a dedicated datagram to facilitate interoperability and avoid users the trouble of defining custom encoding and signalling.

Notation. Hexadecimal numbers are denoted in `typewriter` style, for example `ab` $= 171$. A *word* is either a 32-bit or 64-bit string, which depends on the context. Unless stated otherwise we always use little-endian representation for integers, for example when converting data streams into word arrays. Table 1.1 summarizes the basic operations used throughout the document.

Symbols	Meaning
$a \parallel b$	Concatenation of bitstrings a and b.
$\lvert x \rvert$	Size of bitstring x in bits.
$\mathsf{hw}(x)$	Hamming weight of bitstring x.
$\neg, \wedge, \vee, \oplus$	Bitwise negation, `AND`, `OR` and `XOR`.
$x \ll n, x \gg n$	Left-/Right-shift of bitstring x by n bits.
$x \lll n, x \ggg n$	Left-/Right-rotation of bitstring x by n bits.
$\longleftarrow$	Variable assignment.

Table 1.1: Operations used throughout the document.

Outline. Chapter 2 gives a complete specification of the NORX family of AEAD schemes. Chapter 3 lists the security goals for confidentiality and integrity of the plaintext and for integrity of associated data and public message numbers. Chapter 4 presents features of NORX, justifies our parameter choices, and reports on performance measurements of

[1] The name stems from "NO(T A)RX" and is pronounced like "norcks".

[2] Derived from the well-known identity $a + b = (a \oplus b) + (a \wedge b) \ll 1$ [11, 38].

software implementations on 32- and 64-bit processors and presents preliminary results for an hardware evaluation of an ASIC implementation. Chapter 5 motivates design decisions and Chapter 6 presents preliminary results from the cryptanalysis of various aspects of NORX. Finally, we conclude with notes on the intellectual property, a consent of the CAESAR competition, acknowledgements, references and appendices.

2 Specification

This section gives a complete specification of NORX and its proposed instances.

2.1 Parameters and interface

A NORX instance is parameterised by

- a *word size* of $W \in \{32, 64\}$ bits
- a *number of rounds* $1 \leq R \leq 63$
- a *parallelism degree* $0 \leq D \leq 255$
- a *tag size* $|A| \leq 10W$ bits, with a default of $4W$ bits

Encryption mode

A NORX instance in encryption mode takes as input:

- a *key* K of $4W$ bits
- a *nonce* N of $2W$ bits
- a *message* $M = H \parallel P \parallel T$ where
  - H is a *header*
  - P is a *payload*
  - T is a *trailer*

  and $|H|$, $|P|$ and $|T|$ are allowed to be 0

NORX encryption produces a *ciphertext* (or *encrypted payload*) C of the same size as P and an *authentication tag* A.

Decryption mode

A NORX instance in decryption mode takes as input:

- a key K of $4W$ bits
- a nonce N of $2W$ bits
- a message $M = H \parallel C \parallel T$ where
  - H is a header
  - C is an encrypted payload
  - T is a trailer

and $|H|$, $|C|$ and $|T|$ are allowed to be 0

- an authentication tag A

NORX decryption either returns failure, upon failed verification of the tag, or produces a plaintext P of the same size as C if the tag verification succeeds.

2.2 Naming conventions

A NORX instance is denoted by NORXW-R-D-$|A|$, where W, R, D, and $|A|$ are the parameters of the instance, see §§2.1. If the default tag size is used, i.e. if $|A| = 4W$, the notation for an instance is shortened to NORXW-R-D. So for example, NORX64-6-1 has $(W, R, D, |A|) = (64, 6, 1, 256)$.

2.3 Instances

We propose five instances of NORX, which are specified in Table 2.1.

W	64	32	64	32	64
R	4	4	6	6	4
D	1	1	1	1	4

Table 2.1: NORX instances.

All instances use the default tag size of $4W$ bits, i.e. 128 bit for NORX32 and 256 bit for NORX64. Table 2.1 also reflects the priority order of the recommended parameter sets from highest at the very left (NORX64-4-1) to lowest at the very right (NORX64-4-4). A more detailed discussion on those parameter combinations can be found in §§4.2.

2.4 Layout overview

NORX relies on the monkeyDuplex construction [17, 20], enhanced with the capability of parallel payload processing. The number i of parallel encryption lanes L_i is controlled by the parameter $0 \leq D \leq 255$. For the value $D = 1$, the layout of NORX corresponds to a standard (sequential) duplex construction, see Figure 2.1. For $D > 1$, the number of lanes L_i is bounded by the latter value, e.g. for $D = 2$ see Figure 2.2. If $D = 0$, the number of lanes L_i is bounded by the size of the payload. In that case, the layout of NORX is similar to the PPAE construction [24].

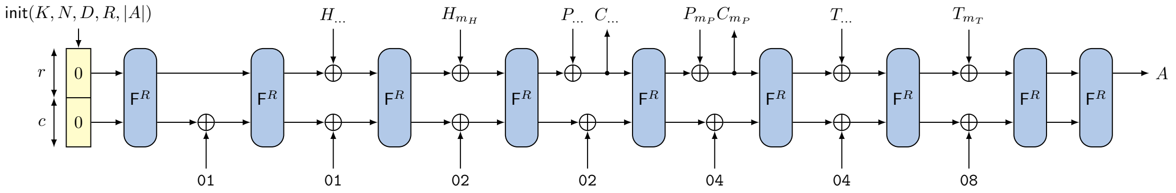


Figure 2.1: Layout of NORX for $D = 1$.

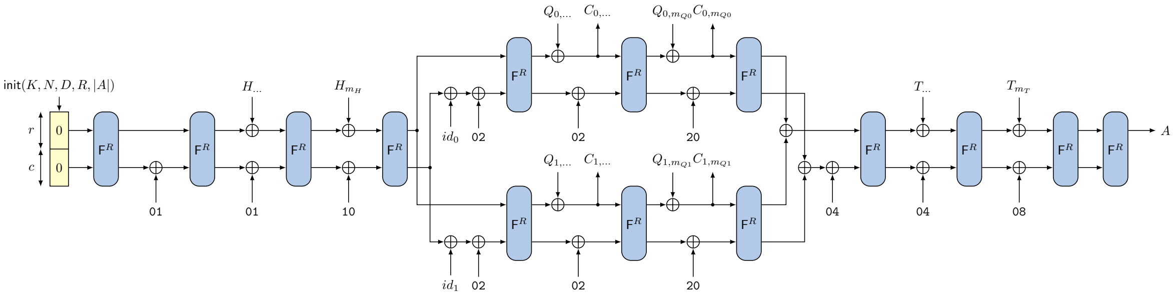


Figure 2.2: Layout of NORX for $D = 2$.

The core algorithm F of NORX is a permutation of $b = r + c$ bits, where b is called the *width*, r the *rate* (or block length), and c the *capacity*. We call F a *round* and F^R denotes its R-fold iteration. The internal state S of NORX64 has $b = 640 + 384 = 1024$ bits and that of NORX32 has $b = 320 + 192 = 512$ bits. The state is viewed as a concatenation of 16 words, i.e. $S = s_0 \parallel \cdots \parallel s_{15}$, where

- $s_0, \dots, s_9$ are called the *rate words* (where data blocks are injected)
- $s_{10}, \dots, s_{15}$ are called the *capacity words* (which remain untouched).

The 16 state words are conceptually arranged in a 4×4 matrix:

$$\begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{pmatrix}$$

2.5 The round function F

The NORX round F processes a state S by first transforming its columns with

$$\mathsf{G}(s_0, s_4, s_8, s_{12}) \quad \mathsf{G}(s_1, s_5, s_9, s_{13}) \quad \mathsf{G}(s_2, s_6, s_{10}, s_{14}) \quad \mathsf{G}(s_3, s_7, s_{11}, s_{15})$$

and then transforming its diagonals with

$$\mathsf{G}(s_0, s_5, s_{10}, s_{15}) \quad \mathsf{G}(s_1, s_6, s_{11}, s_{12}) \quad \mathsf{G}(s_2, s_7, s_8, s_{13}) \quad \mathsf{G}(s_3, s_4, s_9, s_{14})$$

Those two operations are called *column step* and *diagonal step*, as in BLAKE2 [10], and will be denoted by col and diag, respectively. An illustration of the operations is shown in Figure 2.3.

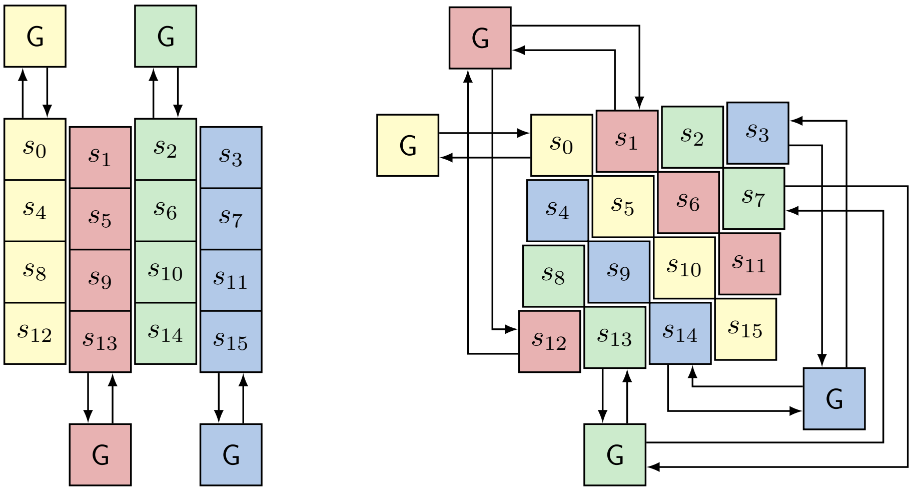


Figure 2.3: Column step and diagonal step of F.

The permutation G transforms four words a, b, c, d by computing

$$\begin{aligned}
a &\longleftarrow (a \oplus b) \oplus \big((a \wedge b) \ll 1\big) \\
d &\longleftarrow (a \oplus d) \ggg r_0 \\
c &\longleftarrow (c \oplus d) \oplus \big((c \wedge d) \ll 1\big) \\
b &\longleftarrow (b \oplus c) \ggg r_1 \\
a &\longleftarrow (a \oplus b) \oplus \big((a \wedge b) \ll 1\big) \\
d &\longleftarrow (a \oplus d) \ggg r_2 \\
c &\longleftarrow (c \oplus d) \oplus \big((c \wedge d) \ll 1\big) \\
b &\longleftarrow (b \oplus c) \ggg r_3
\end{aligned}$$

The rotation offsets r_0, r_1, r_2, r_3 for 32- and 64-bit NORX are specified in Table 2.2. Moreover, Figure 2.4 shows the G circuit.

W	r_0	r_1	r_2	r_3
64	8	19	40	63
32	8	11	16	31

Table 2.2: Rotation offsets for 32- and 64-bit NORX.

2.6 Encrypting and authenticating a message

NORX encryption can process messages M of the form $M = H \parallel P \parallel T$, where H denotes a *header*, P a *payload* and T a *trailer*. H and T are also called *associated data*. Each of H, P and T are allowed to be the empty string.

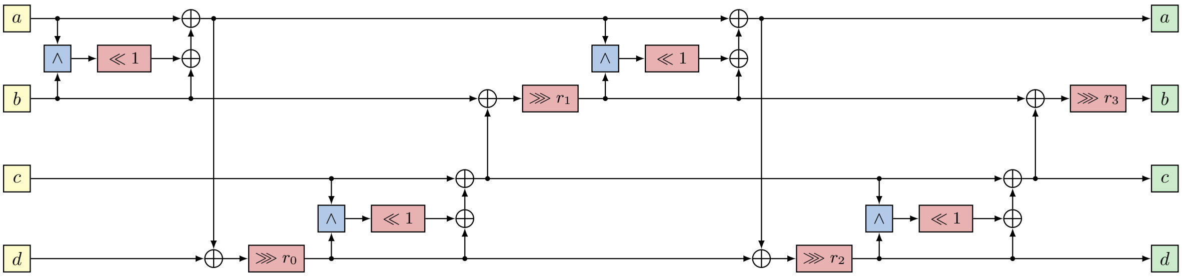


Figure 2.4: The G circuit.

2.6.1 Structure

NORX encryption and authentication consists of multiple processing phases:

1. Initialisation
2. Message processing
   - 2.1. Header processing
   - 2.2. Branching (only for $D \neq 1$)
   - 2.3. Payload processing
   - 2.4. Merging (only for $D \neq 1$)
   - 2.5. Trailer processing
3. Tag generation

An overview of those phases is depicted in Figures 2.1 and 2.2. Processing of a message $M = H \parallel P \parallel T$ is done in one to five steps. The number of steps depends on whether H, T, or P are empty or not and whether $D = 1$ or not. NORX skips the processing phases of empty message parts. For example, in the simplest case when $|H| = |T| = 0$, $|P| > 0$ and $D = 1$, message processing is done in one step since only the payload P needs to be encrypted and authenticated.

Below we first describe the padding and domain separation rules, then each of the aforementioned phases.

2.6.2 Padding

NORX adopts the so-called *multi-rate padding*, which is specified in [20]. This padding rule is defined by the map

$$\mathsf{pad}_r : X \longmapsto X \parallel \mathsf{1}\mathsf{0}^q\mathsf{1}$$

with bitstrings X and $\mathsf{1}\mathsf{0}^q\mathsf{1}$, and $q = (-|X| - 2) \bmod r$. The multi-rate padding extends X to a multiple of the rate r and guarantees that the last block of $\mathsf{pad}_r(X)$ differs from the all-zero block $\mathsf{0}^r$. There are three special cases:

$$q = \begin{cases} r-2, & \text{if} \quad\quad 0 \equiv |X| \bmod r \\ 0, & \text{if} \;\; r-2 \equiv |X| \bmod r \\ r-1, & \text{if} \;\; r-1 \equiv |X| \bmod r \end{cases}$$

In the first case $|X|$ is a multiple of the rate r and thus a full r-bit sized block $10^{(r-2)}1$ is appended to X. This includes the case where X is the empty message, i.e. $|X| = 0$. The second and third cases describe the situations where the smallest and largest number of bits is appended to X. This corresponds to the padding block 11 of size 2 in the former and the padding block $10^{(r-1)}1$ of size $r+1$ in the latter case.

2.6.3 Domain separation

NORX performs domain separation by XORing a *domain separation constant* to the least significant byte of s_{15} each time before the state is transformed by the permutation F^R. Distinct constants are used for the different phases of message processing, for tag generation, and in case of $D \neq 1$, for branching and merging steps. Table 2.3 gives the specification of those constants and Figures 2.1 and 2.2 illustrate their integration into the state of NORX.

header	payload	trailer	tag	branching	merging
01	02	04	08	10	20

Table 2.3: Domain separation constants.

The type of the domain separation constant used at a particular step is determined by the type of the *next* processing phase. The constants are switched together with the phases. For example, as long as the next block is from the header, the domain separation constant 01 is applied. During the processing of the last header block, the constant is switched. If $D = 1$ and the next data block belongs to the payload, the new constant is 02. Then, as long as the next block is from the payload, 02 is used, and so on.

This technique also allows NORX to skip unneeded processing phases. For example, if $D = 1$, $|H| > 0$, $|P| > 0$ and $|T| = 0$, the constant 08 is integrated during processing of the *last* payload block, which means that the trailer phase is skipped and NORX advances directly to the generation of the authentication tag. For the extra initial and final permutations no domain separation constants are used, which is equivalent to XORing 00 to s_{15}. For the special case $D \neq 1$ and $|P| = 0$ not only payload processing is skipped but also branching and merging phases.

2.6.4 Initialisation

Initialisation processes a $4W$-bit key $K = k_0 \parallel k_1 \parallel k_2 \parallel k_3$, a $2W$-bit nonce $N = n_0 \parallel n_1$ and the instance parameters D, R, W and $|A|$.

1. **Basic setup:** The internal state $S = s_0 \parallel \cdots \parallel s_{15}$ is initialised as follows:

$$\begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{pmatrix} \longleftarrow \begin{pmatrix} u_0 & n_0 & n_1 & u_1 \\ k_0 & k_1 & k_2 & k_3 \\ u_2 & u_3 & u_4 & u_5 \\ u_6 & u_7 & u_8 & u_9 \end{pmatrix}$$

where the constants $u_0, \ldots, u_3$ of NORX32 (left) and NORX64 (right) respectively are defined as follows:

$$\begin{aligned} u_0 &= \texttt{243f6a88} & \qquad u_0 &= \texttt{243f6a8885a308d3} \\ u_1 &= \texttt{85a308d3} & \qquad u_1 &= \texttt{13198a2e03707344} \\ u_2 &= \texttt{13198a2e} & \qquad u_2 &= \texttt{a4093822299f31d0} \\ u_3 &= \texttt{03707344} & \qquad u_3 &= \texttt{082efa98ec4e6c89} \end{aligned}$$

The other constants are computed by

$$(u_{4j+4}, u_{4j+5}, u_{4j+6}, u_{4j+7}) = \mathsf{G}(u_{4j}, u_{4j+1}, u_{4j+2}, u_{4j+3})$$

for $j \in \{0, 1\}$. A complete list of the constants is given in Table 5.2.

2. **Parameter integration:** The parameters D, R, W and $|A|$ are integrated into the state S by adding $v = (R \lll 26) \oplus (D \lll 18) \oplus (W \lll 10) \oplus |A|$ to s_{14} followed by R iterations of the round function F:

$$\begin{aligned} s_{14} &\longleftarrow s_{14} \oplus v \\ S &\longleftarrow \mathsf{F}^R(S) \end{aligned}$$

3. **Finalisation:** This step integrates a domain separation constant v into S, whose value is determined according to §§2.6.3, and then updates S by F^R:

$$\begin{aligned} s_{15} &\longleftarrow s_{15} \oplus v \\ S &\longleftarrow \mathsf{F}^R(S) \end{aligned}$$

2.6.5 Message processing

Message processing is the main phase of NORX encryption or decryption. Unless noted otherwise, the value of the domain separation constant v is always determined according to §§2.6.3.

1. **Header processing:**

   If $|H| = 0$, this step is skipped, otherwise H is padded to a multiple of r bits using the multi-rate padding. Let $\mathsf{pad}_r(H) = H_0 \parallel \cdots \parallel H_{m_H - 1}$ denote the padded header data, with r-bit sized header blocks $H_l = h_{l,0} \parallel \cdots \parallel h_{l,9}$ and $0 \leq l \leq m_H - 1$. Then H_l is processed by:

$$\begin{aligned} s_j &\longleftarrow s_j \oplus h_{l,j}, \quad \text{for} \quad 0 \leq j \leq 9 \\ s_{15} &\longleftarrow s_{15} \oplus v \\ S &\longleftarrow \mathsf{F}^R(S) \end{aligned}$$

2. **Branching:** This step is only performed if $D \neq 1$ and $|P| > 0$. In that case, NORX encrypts payload data in parallel on up to D *lanes* L_i, with $0 \leq i \leq D - 1$ if $D > 1$, or $0 \leq i \leq \lceil |P| \,/\, r \rceil$ if $D = 0$. For each lane L_i a copy $S_i = s_{i,0} \parallel \cdots \parallel s_{i,15}$ of the state S is created. The *lane number* i and the domain separation constant $v = 02$ are integrated

into the least significant bytes of $s_{i,13} \parallel s_{i,14}$ and $s_{i,15}$, respectively. Finally each S_i is updated by F^R. In summary, NORX does:

$$\begin{aligned} S_i &\longleftarrow S \\ (s_{i,14}, s_{i,13}) &\longleftarrow (s_{i,14}, s_{i,13}) \oplus (\lfloor i \,/\, 2^W \rfloor, i \bmod 2^W) \\ s_{i,15} &\longleftarrow s_{i,15} \oplus v \\ S_i &\longleftarrow \mathsf{F}^R(S_i) \end{aligned}$$

3. **Payload processing:** If $|P| = 0$, payload processing is skipped. Otherwise, payload data is padded using the multi-rate padding and then encrypted. For padding we distinguish three cases:

   - $D = 1$: This is the standard case, where P is padded in the obvious way, i.e. $\mathsf{pad}_r(P) = P_0 \parallel \cdots \parallel P_{m_P - 1}$.
   - $D > 1$: Here a fixed number of lanes L_i is available for payload encryption, with $0 \leq i \leq D - 1$. Let $P = P_0 \parallel \cdots \parallel P_{m_P-1}$ denote the unpadded payload, with $|P_j| = r$ for $0 \leq j \leq m_P - 2$, and $|P_{m_P-1}| \leq r$. Those blocks are assembled in at most D strings as
     $$Q_i = P_i \parallel P_{D+i} \parallel P_{2\cdot D+i} \parallel \ldots$$
     for $0 \leq i \leq D - 1$. That is, Q_i includes blocks P_l for which $l \equiv i \bmod D$. Then $\mathsf{pad}_r(Q_i) = Q_{i,0} \parallel \cdots \parallel Q_{i,m_i-1}$ is assigned to lane L_i for encryption.
   - $D = 0$: In this case the number of lanes is limited by the payload size. Let $P = P_0 \parallel \cdots \parallel P_{m_P-1}$ denote the unpadded payload data, with $|P_i| = r$ for $0 \leq i \leq m_P - 2$, and $|P_{m_P-1}| \leq r$. Then we simply assign $\mathsf{pad}_r(P_i)$ to lane L_i for encryption.

   The data encryption itself works equivalently for each value of D, hence we describe it only in a generic way.

   Let $\mathsf{pad}_r(P) = P_0 \parallel \cdots \parallel P_{m_P-1}$ denote the padded payload data. To encrypt a payload block $P_l = p_{l,0} \parallel \cdots \parallel p_{l,9}$ and generate a new ciphertext block $C_l = c_{l,0} \parallel \cdots \parallel c_{l,9}$ the following operations are executed:

   $$\begin{aligned} s_j &\longleftarrow s_j \oplus p_{l,j}, \text{ for } 0 \leq j \leq 9 \\ c_{l,j} &\longleftarrow s_j \\ s_{15} &\longleftarrow s_{15} \oplus v \\ S &\longleftarrow \mathsf{F}^R(S) \end{aligned}$$

   for $0 \leq l < m_P - 1$. For $l = m_P - 1$ the procedure is principally the same, but only a truncated ciphertext block is created such that C has the same length as (unpadded) P. In other words, bits used for padding are never written to C.

4. **Merging:** This step is only performed if $D \neq 1$ and $|P| > 0$. After processing of all payload data blocks, the states S_i are merged back into a single state S. Then a domain

separation constant v is integrated, and S is updated by F^R:

$$\begin{aligned} S &\longleftarrow \bigoplus_{i=0}^{D-1} S_i \\ s_{15} &\longleftarrow s_{15} \oplus v \\ S &\longleftarrow \mathsf{F}^R(S) \end{aligned}$$

5. **Trailer processing:** Digestion of trailer data is done analogously to the processing of header data as already described above. Hence, if $|T| = 0$, trailer processing is skipped. If T is non-empty, let $\mathsf{pad}_r(T) = T_0 \parallel \cdots \parallel T_{m_T-1}$ denote the padded trailer data with r-bit trailer blocks T_l and $0 \leq l \leq m_T - 1$. A trailer block $T_l = t_{l,0} \parallel \cdots \parallel t_{l,9}$ is then processed by executing the following steps:

$$\begin{aligned} s_j &\longleftarrow s_j \oplus t_{l,j}, \quad \text{for} \quad 0 \leq j \leq 9 \\ s_{15} &\longleftarrow s_{15} \oplus v \\ S &\longleftarrow \mathsf{F}^R(S) \end{aligned}$$

2.6.6 Tag generation

NORX generates an authentication tag A by transforming S one last time with F^R and then extracting the $|A|$ least significant bits from the rate words $s_0 \parallel \cdots \parallel s_9$ and setting them as A.

$$\begin{aligned} S &\longleftarrow \mathsf{F}^R(S) \\ A &\longleftarrow \bigoplus_{i=0}^{9} (s_i \ll W \cdot i) \bmod 2^{|A|} \end{aligned}$$

2.7 Decrypting a ciphertext and verifying a tag

NORX decryption can process messages M of the form $M = H \parallel C \parallel T$, where H denotes a *header*, C an *encrypted payload* and T a *trailer*. Like in encryption, associated data H and T and payload C can be potentially empty. Decryption additionally takes a *tag* A as input.

2.7.1 Structure

NORX decryption and tag verification has a very similar structure to encryption:

1. Initialisation
2. Message processing
   - 2.1. Header processing
   - 2.2. Branching (only for $D \neq 1$)
   - 2.3. Encrypted payload processing
   - 2.4. Merging (only for $D \neq 1$)

   2.5. Trailer processing

3. Tag generation

4. Tag verification

2.7.2 Padding

Padding is similar to that of encryption, see §§2.6.2.

2.7.3 Domain separation

The domain separation constants and their application are the same as in encryption, see §§2.6.3.

2.7.4 Initialisation

Initialisation is identical to that of encryption, see §§2.6.4.

2.7.5 Message processing

Message processing in decryption is similar to that in encryption, see §§2.6.5, except for the payload processing, which is done as follows:

$$\begin{aligned} p_{l,j} &\longleftarrow s_j \oplus c_{l,j} \\ s_j &\longleftarrow c_{l,j} \\ s_{15} &\longleftarrow s_{15} \oplus v \\ S &\longleftarrow \mathsf{F}^R(S) \end{aligned}$$

Like in encryption as many bits are extracted and written to P as unpadded encrypted payload bits.

2.7.6 Tag generation

Tag generation is identical to that in encryption, see §§2.6.6.

2.7.7 Tag verification

Tag verification consists of comparing the *received tag* A to the *generated tag* A'. If $A = A'$, tag verification succeeds; otherwise tag verification fails, the decrypted payload is discarded and an error is returned.

Implementations of tag verification should satisfy the following requirements:

- Tag verification should not leak information on the (relative) values of the strings compared. In particular tag verification should be implemented in constant time, so that a comparison of identical strings take the same time as a comparison of distinct strings.

- The decrypted payload should not be returned to the user if tag verification fails. Ideally, extracted bytes should be securely erased from any temporary memory if tag verification fails.

2.8 Datagrams

Many issues with encryption interoperability are due to ad hoc ways to represent and transport cryptograms and the associated data. For example IVs are sometimes prepended to the ciphertext, sometimes appended, or sent separately. We thus specify datagrams that can be integrated in a protocol stack, encapsulating the ciphertext as a payload. Using a standardized encoding simplifies the transmission of NORX cryptograms across different APIs, and reduces the risk of insecure or suboptimal encodings. We specify two distinct types of datagrams, depending on whether the NORX parameters are fixed or need to be signaled in the datagram header.

2.8.1 Fixed parameters

With *fixed parameters* shared by the parties (for example through the application using NORX), there is no need to include the parameters in the *header of the datagram*[1]. The datagram for fixed parameters thus only needs to contain N, H, C, T, and A, as well as information to parse those elements.

We encode the byte length of H and T on 16 bits, allowing for headers and trailers of up to 64 KiB, a large enough value for most real applications. The byte length of the encrypted payload is encoded on 32 bits for NORX32 and on 64 bits for NORX64, which translates to a maximum payload size of 4 GiB and 16 EiB, respectively[2]. Similarly to frame check sequences in data link protocols, the tag is added as a *trailer of the datagram* specified. The header, encrypted payload, and trailer of the underlying protocol are viewed as the *payload of the datagram*. The default tag length being a constant value of the NORX instance, it needs not be signalled.

Tables 2.4 and 2.5 show the fixed-parameters datagrams for NORX32 and NORX64. The length of the datagram header is 28 bytes for NORX64 and 16 bytes for NORX32.

Note that the CAESAR API (as per the final call, see [3]) receives the nonce and the associated data in two separate buffers, but the tag is included in the ciphertext buffer.

2.8.2 Variable parameters

With *variable parameters*, the datagram needs to signal the values of W, R, and D. The header is thus extended to encode those values, as specified in Tables 2.6 and 2.7. To minimize bandwidth, W is encoded on one bit, supporting the two choices 32-bit ($W = 0$) and 64-bit ($W = 1$), R on 7 bits (with the MSB fixed at 0, i.e. supporting up to 63 rounds), and D on 8 bits (supporting parallelization degree up to 255). The datagram header is thus only 2 bytes longer than the header for fixed parameters.

[1] The header referred to is that of the datagram specified, not that of the data processed by the NORX instance.

[2] Note that NORX is capable of (safely) processing much larger data sizes, those are just the maximum values when our proposed datagrams are used.

Offset	0	1	2	3
0 4	Nonce N			
8	Header byte length $\|H\|$		Trailer byte length $\|T\|$	
12	Encrypted payload byte length $\|C\|$			
16 ... ??	Header H			
?? ... ??	Encrypted payload C			
?? ... ??	Trailer T			
?? ... ??	Tag A			

Table 2.4: NORX32 datagram for fixed parameters (offsets are in bytes).

Offset	0	1	2	3
0 4 8 12	Nonce N			
16	Header byte length $\|H\|$		Trailer byte length $\|T\|$	
20 24	Encrypted payload byte length $\|C\|$			
28 ... ??	Header H			
?? ... ??	Encrypted payload C			
?? ... ??	Trailer T			
?? ... ??	Tag A			

Table 2.5: NORX64 datagram for fixed parameters (offsets are in bytes).

Offset	0	1	2	3
0 4	Nonce N			
8	Header byte length $\lvert H\rvert$		Trailer byte length $\lvert T\rvert$	
12	Encrypted payload byte length $\lvert C\rvert$			
16	$W(1)\Vert R(7)$	D		
20 ... ??	Header H			
?? ... ??	Encrypted payload C			
?? ... ??	Trailer T			
?? ... ??	Tag A			

Table 2.6: NORX32 datagram for variable parameters (offsets are in bytes).

Offset	0	1	2	3
0 4 8 12	Nonce N			
16	Header byte length $\lvert H\rvert$		Trailer byte length $\lvert T\rvert$	
20 24	Encrypted payload byte length $\lvert C\rvert$			
28	$W(1)\Vert R(7)$	D		
32 ... ??	Header H			
?? ... ??	Encrypted payload C			
?? ... ??	Trailer T			
?? ... ??	Tag A			

Table 2.7: NORX64 datagram for variable parameters (offsets are in bytes).

3 Security goals

We expect NORX with $R \geq 4$ to provide the maximum security for any AEAD scheme with the same interface (input and output types and lengths). The following requirements should be satisfied in order to use NORX securely:

1. **Unique nonces.** Each key and nonce pair should not be used to process more than one message.

2. **Abort on verification failure.** If the tag verification fails, only an error is returned. In particular, the decrypted plaintext and the wrong authentication tag must not be given as an output and should be erased from memory in a safe way.

We do not make any claim regarding attackers using “related keys”, “known keys”, “chosen keys”, etc. We also exclude from the claims below models where information is “leaked” on the internal state or key.

The security of NORX is limited by the key length (128 or 256 bits) and by the tag length (128 or 256 bits). Plaintext confidentiality should thus have the order of 128 or 256 bits of security. The same level of security should hold for integrity of the plaintext or of associated data (based on the fact that an attacker trying 2^n tags will succeed with probability 2^{n-256}, $n < 256$). In particular, recovery of a k-bit NORX key should require resources (“computations”, energy, etc.) comparable to those required to recover the key of an ideal k-bit key cipher. Table 3.1 summarizes the security goals of NORX.

security goal	NORX32	NORX64
plaintext confidentiality	128	256
plaintext integrity	128	256
associated data integrity	128	256
public message number integrity	128	256

Table 3.1: Overview on the security levels (in bits).

Note that NORX restricts the number of messages processed with a given key: in [16] the *usage exponent* e is defined as the value such that the implementation imposes an upper limit of 2^e uses to a given key. In NORX we set it to $e_{64} = 128$ for 64-bit and $e_{32} = 64$ for 32-bit, which corresponds in both cases to the size of the nonce. NORX has capacities of $c_{64} = 384$ (64-bit) and $c_{32} = 192$ (32-bit). As a consequence, security levels of at least $c_{64} - e_{64} - 1 = 384 - 128 - 1 = 255$ bits for NORX64 and $c_{32} - e_{32} - 1 = 192 - 64 - 1 = 127$ bit for NORX32 are expected, see [16].

4 Features

NORX was designed for users, provides several features desirable for practical applications and offers a couple of advantages over AES-GCM [40]. First we list these characteristics in detail, then give a justification of our recommended parameter sets and finally present our performance results.

4.1 List of characteristics

- **High security.** NORX supports 128- and 256-bit keys and authentication tags of arbitrary size, thanks to its duplex construction. The core permutation of NORX was designed and evaluated to be cryptographically strong. The minimal number of 8 rounds for initialisation / finalisation and of 4 rounds for the data processing part ensures a high security margin against cryptanalytic attacks. Large internal states of 512 and 1024 bits and the duplex construction offer protection against generic attacks.

- **Efficiency.** NORX was designed with 64-bit processors in mind, but is also compatible with smaller architectures like 8- to 32-bit platforms. Software implementations of NORX are able to take advantage of multi-core processors, due to the parallel duplex construction, and specialised instruction sets like AVX / AVX2 or NEON. Moreover, state sizes of 512 and 1024 bits make NORX very cache-friendly. Hardware implementations benefit from hardware-friendly operations, next to the arbitrary parallelism degree for payload processing, which results in highly competitive hardware performance of NORX.

- **Simplicity.** The core algorithm iterates a simple round function and can be implemented by translating our pseudocode into the programming language used: NORX requires no SBoxes, no Galois field operations, and no integer arithmetic; `AND`, `XOR`, and shifts are the only instructions required. This simplifies cryptanalysis and the task of implementing the cipher.

- **High key agility.** NORX requires no key expansion when setting up a new key, in contrast to many block-cipher based schemes, like AES-GCM. Switching the secret key is therefore very cheap. As an additional benefit, there are also no hidden costs of loading precomputed expanded keys from DRAM into L1 cache.

- **Adjustable tag sizes.** The NORX family allows tag sizes of up to $10W$ bits, with a default of $4W$ bits for our proposed instances. Thanks to the duplex construction, tag sizes can be easily adapted to the demands of any given application.

- **Simple integration.** NORX can be easily integrated into a protocol stack, as it supports flexible processing of arbitrary datagrams: any header and trailer are authenticated (and left in clear) and the payload is both encrypted and authenticated.

- **Interoperability.** Dedicated datagrams encode parameters of the cipher and encapsulate the protected data. This aims to increase interoperability across implementations.

- **Single pass.** Encryption and decryption of data is done in a single pass of the algorithm.
- **Online.** NORX supports encryption of data streams, i.e. the size of processed data needs not to be known in advance.
- **High data processing volume.** NORX allows to process very large data sizes from a single key-nonce pair. The usage exponent (see §§ 3) theoretically limits the number of calls to the core permutation to values of 2^{64} (NORX32) and 2^{128} (NORX64). This translates to data sizes, which are orders of magnitude beyond everything relevant for current real-world applications. Especially, these values are a lot higher than the maximum of 2^{32} calls to the authenticated encryption function of AES-GCM, which could be easily reached already nowadays in practical applications.
- **Minimal overhead.** Payload encryption is non-expanding, i.e. the ciphertext has the same length as the plaintext. The authentication tag, has a length of 16 or 32 bytes depending on the concrete instance of NORX.
- **Robustness against timing attacks.** By avoiding data-dependent table look-ups, like SBoxes, and integer additions, the goal to harden soft- and hardware implementations of NORX against timing attacks should be comparably easy to achieve.
- **Moderate misuse resistance.** NORX retains its security on nonce reuse as long as it can be guaranteed that header data is unique[1]. For comparison, nonce reuse in AES-GCM is a major security issue, allowing an attacker to recover the secret key [34].
- **Autonomy.** NORX requires no external primitive.
- **Diversity.** The cipher does not depend on AES instructions, thereby adding to the diversity among cryptographic algorithms.

4.2 Recommended parameter sets

We consider NORX32-4-1 and NORX64-4-1 as the standard instances for the respective word sizes of 32 and 64 bit. These configurations offer a good balance between performance and security. We recommend NORX32-4-1 for low resource applications on 8- to 32-bit platforms and NORX64-4-1 for software implementations on modern 64-bit CPUs or standard hardware implementations. Applications that require a higher security margin and where performance has less priority are advised to use the instances NORX32-6-1 and NORX64-6-1.

For use cases where very high data throughput is necessary, we recommend NORX64-4-4, which allows payload encryption on four parallel lanes, thus enabling very high data processing speeds. Finally, we advise hardware implementers not to realise multiple instances of NORX with different parameter combinations at the same time. This holds especially for different values of the parallelism degree D. An implementation should rather be optimised for one set of parameters to gain higher efficiency.

[1] Nevertheless, the designers discourage this approach, and recommend that nonce freshness should be ensured by all means.

4.3 Performance

NORX was designed to perform well across both software and hardware. This section details our implementations and performance results.

4.3.1 Generalities

In this part we analyse some general performance-relevant properties of NORX, like number of operations in G and F^R, parallelism degree, and upper bounds for the speed of NORX on different platforms.

Number of operations

Table 4.1 shows the number of operations required for the NORX core functions. We omit the overhead of initialisation, integration of parameters, domain separation constants, padding messages, and so on, as those costs are negligible compared to that of the core permutation F^R.

function	#XOR	#AND	#shifts	#rotations	total
G	12	4	4	4	24
F	96	32	32	32	192
F^4	384	128	128	128	768
F^6	576	192	192	192	1152
F^8	768	256	256	256	1536
F^{12}	1152	384	384	384	2304

Table 4.1: Overview on the number of operations of the NORX functions.

Memory

NORX32 and NORX64 require at least 16 and 32 bytes to be stored in ROM for the initialisation constants[2]. To store all initialisation constants 40 and 80 bytes of ROM are necessary.

Processing a message in NORX requires enough RAM to store the internal state, i.e., 64 bytes in NORX32 and 128 bytes in NORX64. The data being processed need not be in memory for more than 1 byte at a time. In practice, however, it is preferable to process blocks of 40 (resp. 80) bytes at a time.

Parallelism

NORX's core permutation F has a natural parallelism of 4 independent G applications. Additionally, NORX allows for greater parallelism levels using multiple lanes. Using the $D = 0$ mode (cf. §§2.6.5), the internal parallelism level of NORX is effectively unbounded for long enough messages.

[2] The 10 constants can be generated on the fly from the four basic constants $u_0, \dots, u_3$, see §§2.6.4.

4.3.2 Software

NORX is easily implemented for 32-bit and 64-bit processors, as it works on 32- and 64-bit words and uses only word-based operations (`XOR`, `AND`, shifts and rotations). The specification can directly be translated to code and requires no specific technique such as look-up tables or bitslicing. The core of NORX essentially consists of repeated usage of the G function, which allows simple and compact implementations (e.g., by having only one copy of the G code).

Furthermore, constant-time implementations of NORX are straightforward to write, due to the absence of secret-dependent instructions or branchings.

Bit interleaving

While NORX's lack of integer addition avoids dealing with carry chains, the implementer may still have to perform full-word rotations and shifts in words wider than the natural CPU word size. In 8-bit processors, some of this burden is alleviated by 2 out of 4 rotations being multiples of 8. However, this is only a half-measure.

Instead, the implementer can employ the *bit interleaving* technique presented in [21]. This technique consists of splitting an n-bit word w into $s = n/m$ m-bit words b_i, with $b_{ij} = w_{i+jn/m}$. A rotation by r in this representation can be performed by rotating each b_i by $\lfloor r/w \rfloor + 1$ if $i + r \bmod m < r$, $\lfloor r/w \rfloor$ otherwise, and moving b_i to $b_{i+r \bmod m}$. Rotations by 1 or $n - 1$ are particularly attractive, since they result in a single m-bit rotation. For example, consider implementing NORX64 on a 32-bit CPU. Each state word w will be split into the 2 words b_0 and b_1. To rotate by r:

- If $r \bmod 2 = 0$, rotate both b_0 and b_1 by $\lfloor r/2 \rfloor$;
- If $r \bmod 2 = 1$, rotate b_1 by $\lfloor r/2 \rfloor + 1$, b_0 by $\lfloor r/2 \rfloor$, and swap them.

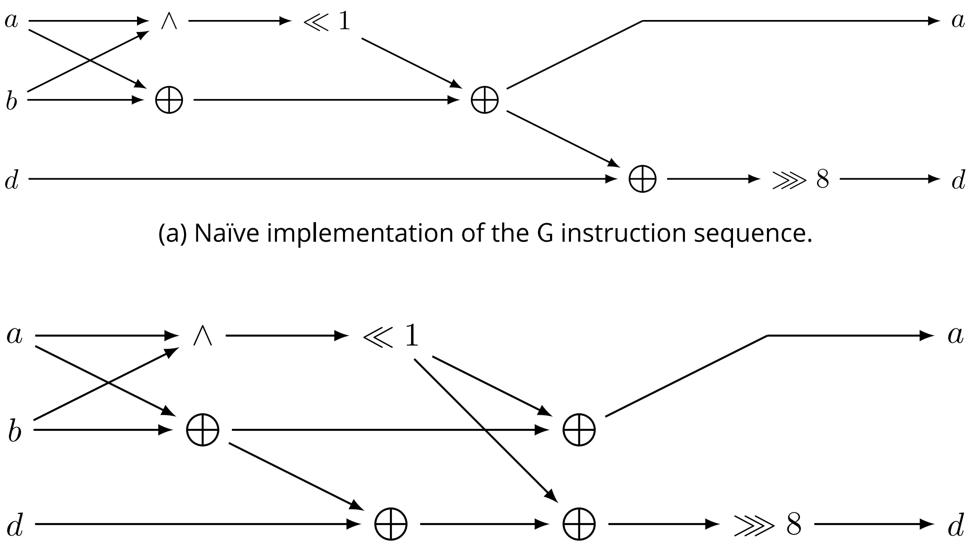
Conversion between representations can be performed in logarithmic time using bit "zip" and "unzip" operations [6].

Avoiding latency

One drawback of G is that it has little instruction parallelism. In architectures where one is limited by the latency of the G function, an implementer can trade a few extra instructions by reduced latency:

$$\begin{aligned}
t_0 &\longleftarrow a \oplus b \\
t_1 &\longleftarrow a \wedge b \\
t_1 &\longleftarrow t_1 \ll 1 \\
a &\longleftarrow t_0 \oplus t_1 \\
d &\longleftarrow d \oplus t_0 \\
d &\longleftarrow d \oplus t_1 \\
d &\longleftarrow d \ggg r_0
\end{aligned}$$

This tweak saves up to 1 cycle per instruction sequence, of which there are 4 per G, at the cost of 1 extra instruction (cf. Figure 4.1). In a sufficiently parallel architecture, this can save at least $4 \times 2 \times R$ cycles, which translates to $6.4R/W$ cycles per byte saved overall. In our measurements, this translated to a performance improvement of NORX from 0.4 to 0.7 cycles per byte, depending on the target architecture, word size, and number of rounds.



(a) Naïve implementation of the G instruction sequence.

(b) Latency-oriented version of the G instruction sequence.

Figure 4.1: Improving the latency of G.

Vectorization

NORX lends itself quite well to implementations taking advantage of SIMD extensions present in modern processors, such as AVX or NEON.

The typical vectorized implementation of NORX, when $D = 1$, works in full rows of the 4×4 state, and computes whole column and diagonal steps of F in parallel.

Results

We wrote portable C reference implementations for both NORX64 and NORX32, as well as optimized versions for CPUs supporting AVX and AVX2 and for NEON-enabled ARMs. Table 4.2 shows speed measurements on various platforms for messages with varying lengths. The listed CPU frequencies are nominal ones, i.e. without dynamic overclocking features like Turbo Boost, which improves the accuracy of measurements. Furthermore we listed only those platform-compiler combinations that achieved the highest speeds. Unless stated otherwise we used the compiler flags

```
-O3 -march=native -std=c89 -Wall -pedantic -Wno-long-long
```

The top speed of NORX (for $D = 1$), in terms of bytes per second, was achieved by an AVX2 implementation of NORX64-4-1 on a Haswell CPU, listed in Table 4.2. It achieves a throughput of about 1.39 GiBps (2.51 cycles per byte at 3.5 GHz). The overhead for short messages (≤ 64 bytes) is mainly due to the additional initialisation and finalisation rounds (see Figure 2.1).

However the cost per byte quickly decreases, and stabilizes for messages larger than about 1 KiB.

Note that the speed between reference and optimized implementations differs by a factor of less than 2, suggesting that straightforward and portable implementations will provide sufficient performance in most applications. Such consistent performance reduces development costs and improves interoperability.

data length [byte]		long	4096	1536	576	64	8
Samsung Exynos 4412 Prime (Cortex-A9) at 1.7 GHz							
NORX32-6-1	Ref	31.56	32.35	35.45	42.14	128.66	794.12
	NEON	15.66	16.70	18.36	22.79	77.78	541.00
NORX32-4-1	Ref	21.57	22.86	24.94	30.50	97.94	663.75
	NEON	10.57	11.41	12.77	16.40	61.73	434.88
Intel Core i7-2630QM at 2.0 GHz							
NORX64-6-1	Ref	7.69	8.14	9.08	11.54	37.75	304.00
	AVX	4.94	5.24	5.90	7.52	24.81	198.00
NORX64-4-1	Ref	5.28	5.59	6.24	7.94	26.00	208.00
	AVX	3.28	3.49	3.91	5.03	16.69	133.50
Intel Core i7-3667U at 2.0 GHz							
NORX64-6-1	Ref	7.04	7.46	8.32	10.59	34.87	371.50
	AVX	5.04	5.37	6.03	7.71	25.44	276.00
NORX64-4-1	Ref	4.92	5, 24	5.86	7.43	24.93	310.00
	AVX	3.37	3.59	4.01	5.16	17.18	218.00
Intel Core i7-4770K at 3.5 GHz							
NORX64-6-1	Ref	6.63	7.00	7.77	9.85	32.12	256.50
	AVX2	3.73	3.98	4.47	5.71	19.19	153.00
NORX64-4-1	Ref	4.50	4.76	5.27	6.71	22.06	176.00
	AVX2	2.51	2.66	3.01	3.83	13.06	104.00

Table 4.2: Software performance of NORX in cycles per byte.

4.3.3 Hardware

Hardware architectures of NORX are efficient and easy to design from the specification: vertical and parallel folding are naturally derived from the iterated and parallel structure of NORX. The cipher benefits from the hardware-friendliness of the function G, which requires only bitwise logical AND, XOR, and bit shifts, and the iterated usage of G inside the core permutation of NORX.

A hardware architecture was designed, supporting parameters $W \in \{32, 64\}$, $R \in \{2, \ldots, 16\}$

and $D = 1$. It was synthesized with the Synopsys Design Compiler for an ASIC using 180 nm UMC technology. The implementation was targeted at high data throughput. The requirements in area amounted to about 62 kGE. Simulations for NORX64-4-1 report a throughput of about 10 Gbps (1.2 GiBps), at a frequency of 125 MHz.

A more thorough evaluation of all hardware aspects of NORX is planned for the future. Due to the similarity of NORX to ChaCha and the fact that NORX has only little overhead compared to a blank stream cipher, we expect results similar to those of Chacha as presented in [33].

5 Design rationale

In this chapter we motivate the design choices made in NORX. We pursue a top-down approach, starting with the general layout and going into the details of the cipher's components in the later sections.

5.1 The parallel duplex construction

The layout of NORX is based on the monkeyDuplex construction [17, 20], but enhanced by the capability of parallel payload processing on multiple lanes (cf. Figures 2.1 and 2.2). The *parallel duplex construction* is similar to the tree-hashing mode for sponge functions [18]. It allows NORX to take advantage of multi-core processors and enables high-throughput hardware implementations. Associated data can be authenticated as header and/or trailer data but only on a single lane. We felt that it is not worth the effort to enable processing of H and T in parallel, as they are usually rather short. The number of encryption lanes is controlled by the parallelism degree $0 \leq D \leq 255$, which is a fixed instance parameter. Hence two instances with distinct D values cannot decrypt data from each other. Obviously the same holds for differing W and R values.

To ensure that the payload blocks on parallel lanes are encrypted with distinct key streams, we use the branching phase to include an id into each of the parallel lanes. For NORX the id is a simple counter. Once the parallel payload processing is finished, the states are re-combined in the merging phase and NORX advances to the processing of the trailer (if present) or generation of the authentication tag.

There does not exist a formal proof of security for the parallel duplex construction yet. Note that the most problematic step could be the merging phase for $D \neq 1$, due to the fact that (multi-)collisions could occur. However, we expect that the construction is safe in case of a nonce-respecting adversary. We will try to hand in the proof at a later point of time.

5.2 The G function

The G function of NORX is inspired by the quarter-round function of the stream cipher ChaCha [14], which itself is an advancement of the quarter-round function of the eSTREAM finalist Salsa20 [1, 15]. Variants of ChaCha's quarter-round function can be found for example in the SHA-3 finalist BLAKE [2, 9] and its successor BLAKE2 [10].

Overview

One of the main goals for NORX was to design a core primitive, which does not rely on integer addition to introduce non-linearity. Instead it should use exclusively more hardware-friendly bitwise logic operations like `NOT`, `AND`, `OR`, or `XOR` and bit-shifts. Figure 5.1 shows how the G function of NORX transforms an input (a, b, c, d) compared to the quarter-round function of

ChaCha . The rotation offsets for NORX are specified in Table 2.2. The offsets of ChaCha are $(s_0, s_1, s_2, s_3) = (16, 12, 8, 7)$ for 32-bit and $(s_0, s_1, s_2, s_3) = (32, 24, 16, 63)$ for 64-bit.[1]

$$\begin{array}{lcl|lcl}
a & \longleftarrow & (a \oplus b) \oplus \big((a \wedge b) \lll 1\big) & a & \longleftarrow & a + b \\
d & \longleftarrow & (a \oplus d) \ggg r_0 & d & \longleftarrow & (a \oplus d) \ggg s_0 \\
c & \longleftarrow & (c \oplus d) \oplus \big((c \wedge d) \lll 1\big) & c & \longleftarrow & c + d \\
b & \longleftarrow & (b \oplus c) \ggg r_1 & b & \longleftarrow & (b \oplus c) \ggg s_1 \\
a & \longleftarrow & (a \oplus b) \oplus \big((a \wedge b) \lll 1\big) & a & \longleftarrow & a + b \\
d & \longleftarrow & (a \oplus d) \ggg r_2 & d & \longleftarrow & (a \oplus d) \ggg s_2 \\
c & \longleftarrow & (c \oplus d) \oplus \big((c \wedge d) \lll 1\big) & c & \longleftarrow & c + d \\
b & \longleftarrow & (b \oplus c) \ggg r_3 & b & \longleftarrow & (b \oplus c) \ggg s_3
\end{array}$$

Figure 5.1: Comparison of NORX (left) and ChaCha (right) core functions.

In NORX the integer additions is replaced by the following expression

$$x \longleftarrow (x \oplus y) \oplus \big((x \wedge y) \lll 1\big)$$

which uses bitwise logical AND to introduce non-linearity. It mimics integer addition of two bit strings x and y with a 1-bit carry propagation and thus provides, in addition to non-linearity, also a slight diffusion of bits. In conformity with the main design principle of NORX, we tried to make the non-linear operation as simple as possible in order to simplify cryptanalysis and to reduce the risk of overlooking potential security weaknesses.

Bijectivity

The only expression in G which is not obviously invertible at a first glance, is the non-linear operation

$$z = (x \oplus y) \oplus ((x \wedge y) \lll 1)$$

with n-bit words x, y and z. In order to proof bijectivity of the above expression we show how to invert it, under the assumption that one of its inputs is fixed. Therefore we write $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$, $y = \sum_{i=0}^{n-1} y_i \cdot 2^i$ and $z = \sum_{i=0}^{n-1} z_i \cdot 2^i$ with x_i, y_i and $z_i \in \{0, 1\}$ and assume that y is fixed. Writing down the inverse non-linear operation at bit level is then straightforward:

$$\begin{aligned}
x_0 &= (z_0 \oplus y_0) \\
x_1 &= (z_1 \oplus y_1) \oplus (x_0 \wedge y_0) \\
&\vdots \\
x_i &= (z_i \oplus y_i) \oplus (x_{i-1} \wedge y_{i-1}) \\
&\vdots \\
x_{n-1} &= (z_{n-1} \oplus y_{n-1}) \oplus (x_{n-2} \wedge y_{n-2})
\end{aligned}$$

[1] The original ChaCha stream cipher is defined for 32-bit words. For the 64-bit version we used the rotation offsets $(32, 24, 16, 63)$ from the BLAKE2 specification [10].

This proves that G is indeed a permutation. Further, it is a permutation when either of its input arguments is fixed, making it also a latin square.

Features

The only operations required to define G are bitwise `XOR`, `AND` and logical bit shifts, which has several advantages: All of the mentioned instructions can be implemented in constant time regardless of the word size. Especially for hardware implementations there are no carry-propagations to worry about, for example, as there would be for integer addition $\bmod 2^n$.
Moreover no table-lookup instructions, like SBoxes, are required, where the table index is data-dependent. Those operations, if not handled with extreme care, are often the reason for implementations leaking side-channel information, making the affected algorithm vulnerable, e.g., to timing-attacks [12]. By avoiding them, the task of hardening the cipher against side-channel attacks gets obviously much easier. No specialised implementations are required, e.g., bit-sliced SBoxes [4, 28], for table-lookups in constant time. Additionally, the waiving of more sophisticated instructions like integer addition, multiplication, Galois field arithmetic or other constructs based on linear algebra, has the effect that the algorithm is much easier to implement (both in soft- and hardware) and thus reduces the threat of introducing unwanted bugs.

5.3 The F function

The layout of the round function F of NORX is the same as used in ChaCha [14].

Overview

Recall that F transforms a state $S = s_0 \parallel \cdots \parallel s_{15}$ in two phases. First a column step is applied

$$\mathsf{G}(s_0, s_4, s_8, s_{12}) \quad \mathsf{G}(s_1, s_5, s_9, s_{13}) \quad \mathsf{G}(s_2, s_6, s_{10}, s_{14}) \quad \mathsf{G}(s_3, s_7, s_{11}, s_{15})$$

followed by a diagonal step

$$\mathsf{G}(s_0, s_5, s_{10}, s_{15}) \quad \mathsf{G}(s_1, s_6, s_{11}, s_{12}) \quad \mathsf{G}(s_2, s_7, s_8, s_{13}) \quad \mathsf{G}(s_3, s_4, s_9, s_{14})$$

Bijectivity

As G is a permutation, F is obviously a permutation, too. This means that there exist no states S and S', with $S \neq S'$, which produce the same result, i.e. $\mathsf{F}^R(S) = \mathsf{F}^R(S')$, after any number of rounds R. This characteristic of F is important for the duplex construction [20, 17] in order to retain some desirable security properties.

Features

One great advantage of the ChaCha-related layout of F is, that the modification of a single bit in the input has the chance of affecting all 16 output words[2] after only one application of F. This features greatly enhances diffusion. Another benefit of the layout is the ability to execute the four applications of G in a step completely in parallel, which improves performance.

[2] In fact we found for NORX only one case where less than 16 words are affected. This can be achieved through the modification of three very specific bits in the input. See chapter §§6 on cryptanalysis for more details.

5.4 Selection of rotation offsets

The rotation offsets (r_0, r_1, r_2, r_3) used by NORX provide a good balance between security and efficiency. The values r_i, with $0 \leq i \leq 3$, were selected according to the following conditions:

1. At least two out of four offsets are multiples of 8.
2. The remaining offsets are odd and have the form $8n \pm 1$ or $8n \pm 3$, with a preference for the first shape.

The motivation behind those criteria has the following reasons: An offset which is a multiple of 8 preserves byte alignment and thus is much faster than an unaligned rotation on many non-64-bit architectures. Many 8-bit microcontrollers have only 1-bit shifts of bytes, so for example rotations by 5 bits are particularly expensive. Using aligned rotations, i.e. permutations of bytes, greatly increases the performance of the entire algorithm. Even 64-bit architectures benefit from such aligned rotations, for example when an instruction sequence of two shifts followed by `XOR` can be replaced by SSSE3's byte shuffling instruction `pshufb`. Odd offsets break up the byte structure and therefore increase diffusion.

In order to find good rotation offsets and assess their diffusion properties, we used an automated search combined with a diffusion test. Therefore let R denote a round number and let L and L_R be lists. For each offset tuple (r_0, r_1, r_2, r_3) with $r_i \in \{1, \ldots, W-1\}$ satisfying the above criteria, the following steps are repeated 10^6 times, after the offsets have been plugged into G:

1. Choose two b-bit sized states S and S' uniformly at random, such that $\mathsf{hw}(S \oplus S') = 1$.
2. Compute $X = \mathsf{F}^R(S) \oplus \mathsf{F}^R(S')$, where F denotes the round function of NORX.
3. Save $\mathsf{hw}(X)$ to L_R.

After the above loop is finished the test computes minimum, maximum, average and median values of the elements of L_R, saves the latter together with the offsets to L and resets L_R. Then it proceeds to the analysis of the next rotation tuple. This test is repeated until all candidate offsets have been processed.

Finally, we chose the offsets $(8, 19, 40, 63)$ for NORX64 and $(8, 11, 16, 31)$ for NORX32, which belonged to those having very high values for average and median Hamming weight for $R = 1$, achieve full diffusion after $R = 2$, and additionally offer good performance.

Table 5.1 lists the results of the test for 32- and 64-bit core functions with $R \leq 4$ and rotation offsets as specified above. The test results show that the diffusion speed of NORX's round function F is almost as high as ChaCha's and that full diffusion is reached after two rounds. Figure 5.2 shows how single bit changes in the word s_0 propagate through the NORX state over the course of 5 steps ($= \mathsf{F}^{2.5}$). Unfortunately there seems to be no combination of rotation values with 3 offsets being a multiple of 8 and one being $W - 1$, like BLAKE2's $(32, 24, 16, 63)$, where F achieves a comparably strong diffusion as illustrated in Table 5.1. The reason for this can be traced back to the replacement of integer addition by the non-linear operation of NORX.

	NORX32				ChaCha (32-bit)			
R	min	max	avg	median	min	max	avg	median
1	83	280	179.222	181	73	294	182.195	185
2	194	307	256.024	256	199	312	255.999	256
3	198	312	255.995	256	204	313	255.988	256
4	201	307	255.996	256	200	314	255.989	256
	NORX64				ChaCha (64-bit)			
R	min	max	avg	median	min	max	avg	median
1	95	429	230.136	222	73	506	248.843	246
2	440	589	511.982	512	430	591	512.013	512
3	434	589	512.008	512	439	589	511.971	512
4	428	589	511.986	512	435	585	512.008	512

Table 5.1: Diffusion statistics for NORX and ChaCha round functions.

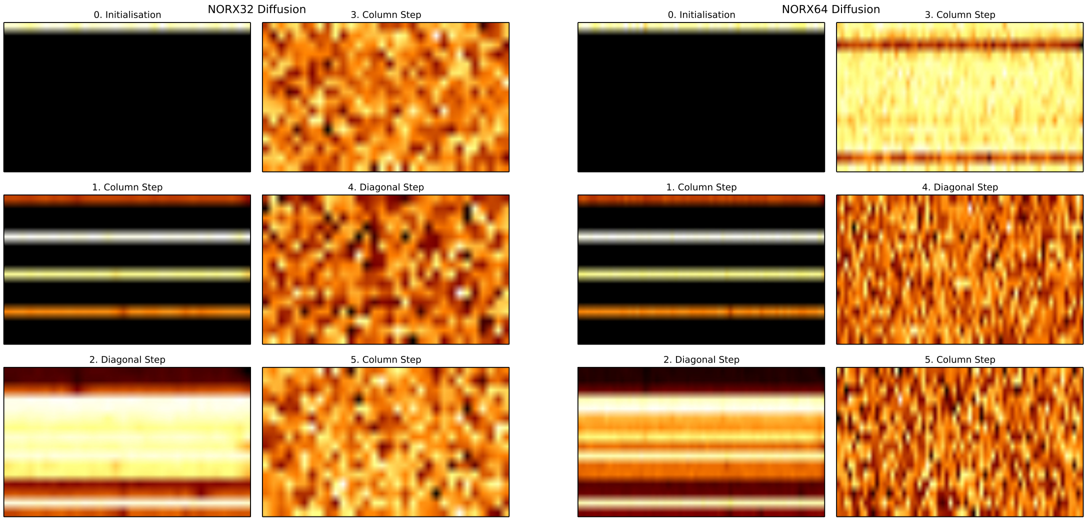


Figure 5.2: Visualisation of NORX diffusion.

5.5 Number of rounds

For a higher protection of the key and authentication tag, e.g. against differential cryptanalysis, we chose twice the number of rounds for initialisation and finalisation, compared to the data processing phases. This measure was already proposed in [17] and has only minor effects on the overall performance, but greatly increases the security of NORX. The minimal value of $R = 4$ is based on the following observations:

1. The best attacks on Salsa20 and ChaCha [8, 45, 47] break 8 and 7 rounds, respectively, which roughly corresponds to 4 and 3.5 rounds of the NORX core. However this is within a much stronger attack model than that provided by the duplex construction of NORX.
2. The preliminary cryptanalysis of NORX as presented in Section 6. The best differentials we were able to find, belong to a class of high-probability truncated differentials over 1.5 rounds and a class of impossible differentials over 3.5 rounds. Despite the fact that those differentials cannot be used to mount an attack on NORX, it might be possible to

find similar differentials, using more advanced cryptanalytic techniques, which could be used for an attack.

The number of rounds may be adjusted according to the future cryptanalytic results on NORX.

5.6 Selection of constants

Initialisation constants

The four basic constants $u_0, \ldots, u_3$ of 32-bit and 64-bit NORX correspond to the first digits of π. The other six constants are derived iteratively from $u_0, \ldots, u_3$ by

$$(u_{4j+4}, u_{4j+5}, u_{4j+6}, u_{4j+7}) = \mathsf{G}(u_{4j}, u_{4j+1}, u_{4j+2}, u_{4j+3})$$

for $j \in \{0, 1\}$. The complete list of constants is depicted in Table 5.2. The main purpose of the constants is to bring asymmetry during initialisation and to limit the freedom of an attacker where he might inject differences.

	NORX32	NORX64		NORX32	NORX64
u_0	243F6A88	243F6A8885A308D3	u_5	38531D48	670A134EE52D7FA6
u_1	85A308D3	13198A2E03707344	u_6	839C6E83	C4316D80CD967541
u_2	13198A2E	A4093822299F31D0	u_7	F97A3AE5	D21DFBF8B630B762
u_3	03707344	082EFA98EC4E6C89	u_8	8C91D88C	375A18D261E7F892
u_4	254F537A	AE8858DC339325A1	u_9	11EAFB59	343D1F187D92285B

Table 5.2: Initialisation constants of NORX.

Domain separation constants

The NORX algorithm is separated into different data processing phases. Each phase uses its own domain separation constant to mark the end of certain events like the absorbing of data blocks or merging and branching steps in case of an instance with parallelism degree $D \neq 1$. A domain separation constant is always added to the least significant byte of the capacity word s_{15}. The constants are given in Table 2.3. The separation of the processing phase is important for the security proofs of the indifferentiability of the duplex construction [19, 20]. In addition they help to break the self-similarity of the round function and thus increase the complexity of certain kind of attacks on NORX, for example, like slide attacks, see §§6.3.2.

5.7 The padding rule

The sponge (or duplex) construction offers protection against generic attacks if the padding rule is sponge-compliant, i.e. if it is injective and ensures that the last block is different from the all-zero block. In [18] it has been proven that the multi-rate padding satisfies those properties. Moreover it is simple to describe, easy to implement and very efficient. Thus it was a natural choice to be used in NORX. Additionally, the multi-rate padding increases the complexity to mount certain kind of attacks on NORX, like slide attacks, see §§6.3.2.

5.8 Absence of backdoors

We, the designers of NORX, faithfully declare that we have not inserted any hidden weaknesses in this cipher.

6 Security analysis

This chapter presents preliminary cryptanalysis of NORX.

6.1 Differential cryptanalysis

Differential attacks cover all attacks that exploit non-ideal propagation of differences in a cryptographic algorithm (or of its components). Differential cryptanalysis is one of the standard tools in the repertoire of every cryptanalyst and usually a lot of attacks on a cipher are at least partially differential. It is thus crucial to analyse the resistance of new designs to differential attacks.

First we introduce some of the required notations, then we analyse the propagation of differences through the G function, show how to construct high-probability truncated differentials of low weight for the core permutation F^R and finally study impossible differential cryptanalysis.

6.1.1 Notation

Definition 1. Let x and x' be n-bit strings. We call $\alpha = x \oplus x'$ the *difference* of x and x' with respect to bitwise `XOR`. Furthermore for tuples of n-bit strings $(x_0, \ldots, x_{m-1})$ and $(x'_0, \ldots, x'_{m-1})$ we call the component-wise difference

$$(\alpha_0, \ldots, \alpha_{m-1}) = (x_0, \ldots, x_{m-1}) \oplus (x'_0, \ldots, x'_{m-1}) = (x_0 \oplus x'_0, \ldots, x_{m-1} \oplus x'_{m-1})$$

a *tuple of differences*.

Definition 2. An n-bit difference α with $\mathsf{hw}(\alpha) = m$ and 1-entries at bit positions $0 \leq i_0 \leq \cdots \leq i_m \leq n-1$ is denoted by $\alpha[i_0, \ldots, i_m]$.

Definition 3. Let $f : \{0,1\}^{m \cdot n} \longrightarrow \{0,1\}^{k \cdot n}$, $f(a_0, \ldots, a_{m-1}) = (b_0, \ldots, b_{k-1})$ be a boolean function. Let $\alpha := (\alpha_0, \ldots, \alpha_{m-1}) = (x_0, \ldots, x_{m-1}) \oplus (x_0, \ldots, x_{m-1})$ and let $\beta := (\beta_0, \ldots, \beta_{k-1}) = f(x_0, \ldots, x_{m-1}) \oplus f(x'_0, \ldots, x'_{m-1})$ be tuples of differences. Then we call (α, β) a *differential* with respect to the function f and denote it by

$$\alpha \xrightarrow{f} \beta$$

If the context is clear we skip the f above the arrow and just write $\alpha \longrightarrow \beta$. Furthermore, we call α an *input difference* and β an *output difference* of f.

In our later analysis of NORX we usually consider functions f having $k = 1$ or $k = m$.

Definition 4. Let $f_0, \ldots, f_{l-1}$ be boolean functions defined by

$$f_i : \{0,1\}^{m \cdot n} \longrightarrow \{0,1\}^{m \cdot n},\ f_i(a_0, \ldots, a_{m-1}) = (b_0, \ldots, b_{m-1})$$

for $i \in \{0, \ldots, l-1\}$. Further let $\alpha^0 := (\alpha^0_0, \ldots, \alpha^0_{m-1}), \ldots, \alpha^l := (\alpha^l_0, \ldots, \alpha^l_{m-1})$ be tuples of differences such that

$$\alpha^i \xrightarrow{f_i} \alpha^{i+1}$$

Then we call $(\alpha^0, \ldots, \alpha^l)$ a *differential characteristic* with respect to the functions $f_0, \ldots, f_{l-1}$ and denote it by

$$\alpha^0 \xrightarrow{f_0} \ldots \xrightarrow{f_{i-1}} \alpha^i \xrightarrow{f_i} \ldots \xrightarrow{f_l} \alpha^l$$

The tuples α^j with $j \in \{1, \ldots, l-1\}$ are also called *internal differences*. In the case where $f := f_0 = \cdots = f_{l-1}$ we also say that $(\alpha^0, \ldots, \alpha^l)$ is a differential characteristic with respect to the *iterated function f*.

The notion of a differential characteristic can obviously be defined for arbitrary boolean functions f_i, but it is not required at this point. Thus, for reasons of simplicity, we decided to define it only for the special case, where the dimension of the domain equals the dimension of the codomain of f_i.

Definition 5. Every differential (α, β) of a function f has a probability $p \in [0, 1]$ associated to it, which will be written as

$$\Pr(\alpha \xrightarrow{f} \beta) = p$$

To capture all those informations in a compact form, we denote a differential (α, β) of probability p with respect to a function f by:

$$\alpha \xrightarrow[p]{f} \beta$$

We use the commonly accepted assumption that the probability of a differential is equal to the sum of probabilities of all differential characteristics corresponding to this differential. Moreover it is commonly assumed that the probability of the best differential can accurately be estimated by the probability of the best differential characteristic.

6.1.2 Differential properties of G

In this section we analyse how n-bit input differences α with $\mathsf{hw}(\alpha) = 1$ propagate through G and present the probabilities of the resulting output differences. Therefore, we decompose G into two functions G_1 and G_2 and initially analyse the behaviour of G_1.

Definition 6. Let $\mathsf{G}_1 : \{0,1\}^{4n} \longrightarrow \{0,1\}^{4n}$ be defined as

$$\begin{aligned} a &\longleftarrow (a \oplus b) \oplus ((a \wedge b) \ll 1) \\ d &\longleftarrow (a \oplus d) \ggg r_0 \\ c &\longleftarrow (c \oplus d) \oplus ((c \wedge d) \ll 1) \\ b &\longleftarrow (b \oplus c) \ggg r_1 \end{aligned}$$

The function G_2 is defined analogously to G_1 but with rotation offsets r_2 and r_3, instead of r_0 and r_1. Thus, we obviously have $\mathsf{G}(a, b, c, d) = \mathsf{G}_2(\mathsf{G}_1(a, b, c, d))$.

Let (x_0, x_1, x_2, x_3) and (x'_0, x'_1, x'_2, x'_3) be two tuples of n-bit strings having difference

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3) = (x_0, x_1, x_2, x_3) \oplus (x'_0, x'_1, x'_2, x'_3)$$

and let

$$(\beta_0, \beta_1, \beta_2, \beta_3) = \mathsf{G}_1(x_0, x_1, x_2, x_3) \oplus \mathsf{G}_1(x'_0, x'_1, x'_2, x'_3)$$

Further assume that $\mathsf{hw}(a_v) = 1$ for a fixed $v \in \{0, \ldots, 3\}$ where the 1-entry is a bit position i and $\mathsf{hw}(a_u) = 0$ for all $u \in \{0, \ldots, 3\} \setminus \{v\}$. Then we get differentials

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3) \xrightarrow{\mathsf{G}_1} (\beta_0, \beta_1, \beta_2, \beta_3)$$

and associated probabilities as presented in Table 6.1. Note that the output difference β_w, for $w \in \{0, \ldots, 3\}$ is the XOR sum of the 1-bit differences $\beta_w[j]$ in a given column. The resulting $\beta_w[j]$ do not hold for arbitrary $\alpha_v[i]$ with $i \in \{0, \ldots, n-1\}$. For example if $i = n-1$ the difference $\alpha_v[i]$ will be erased by the shift operation $\alpha_v[i] \ll 1$, thereby cancelling all output differences depending [1] on the latter.

The differentials in Table 6.1 only hold for input differences having exactly one active bit. Obviously, when allowing input differences with a larger number of active bits the situation gets immediately a lot more complex. This could lead to situations where active bits of different words interact and cancel each other out. For example an input difference $(\alpha_0[n-1], \alpha_1[n-1], 0, 0)$ leads to a cancellation of the probability 1 output difference $\alpha_0[n-1]$ in the output word a: The two active bits in the input words a and b neutralise each other during the update of the word a. We will see below how this property can be exploited to build differentials for G having high probability and low weight output differences.

To compute the output differences for G we can obviously proceed in the following way:

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3) \xrightarrow{\mathsf{G}_1} (\beta_0, \beta_1, \beta_2, \beta_3) \xrightarrow{\mathsf{G}_2} (\gamma_0, \gamma_1, \gamma_2, \gamma_3)$$

Listing all 1-bit output differences $\gamma_w[j]$ of G on an arbitrary input difference $\alpha_v[i]$ is quite a complex task. Thus we only give an estimation of the maximum number of active bits in the output difference $\gamma := (\gamma_0, \gamma_1, \gamma_2, \gamma_3)$ after one application of G. Table 6.2 lists the results, which were also confirmed experimentally.

6.1.3 Simple differentials

In this section we show how to construct a class of high probability differentials for the round function F and a small number of iterations F^R. We will focus here on NORX64, but similar considerations should hold for NORX32.

We first consider a simple attack model where the initial state is assumed chosen uniformly at random and where one seeks differences in the initial state that give biased differences in the state obtained after a small number of iterations of F. High-probability truncated differentials wherein the output difference concerns only a small subset of bits (e.g., a single bit) are sufficient to distinguish a (reduced-round) permutation from a random one, and are easier to

[1] We refer to Figure B.1 in the appendix for a visualisation of the relations between input and output differences of G_1.

	$\beta_0[j]$	$\beta_1[j]$	$\beta_2[j]$	$\beta_3[j]$	$\Pr(\alpha_v[i] \xrightarrow{\mathsf{G}_1} \beta_w[j])$
$\alpha_0[i]$	$\alpha_0[i]$	0	0	0	1
	$\alpha_0[i] \ll 1$	0	0	0	2^{-1}
	0	$\alpha_0[i] \ggg (r_0 + r_1)$	0	0	1
	0	$((\alpha_0[i] \ggg r_0) \ll 1) \ggg r_1$	0	0	2^{-1}
	0	$(\alpha_0[i] \ll 1) \ggg (r_0 + r_1)$	0	0	2^{-1}
	0	$(((\alpha_0[i] \ll 1) \ggg r_0) \ll 1) \ggg r_1$	0	0	2^{-2}
	0	0	$\alpha_0[i] \ggg r_0$	0	1
	0	0	$(\alpha_0[i] \ggg r_0) \ll 1$	0	2^{-1}
	0	0	$(\alpha_0[i] \ll 1) \ggg r_0$	0	2^{-1}
	0	0	$((\alpha_0[i] \ll 1) \ggg r_0) \ll 1$	0	2^{-2}
	0	0	0	$\alpha_0[i] \ggg r_0$	1
	0	0	0	$(\alpha_0[i] \ll 1) \ggg r_0$	2^{-1}
$\alpha_1[i]$	$\alpha_1[i]$	0	0	0	1
	$\alpha_1[i] \ll 1$	0	0	0	2^{-1}
	0	$\alpha_1[i] \ggg r_1$	0	0	1
	0	$\alpha_1[i] \ggg (r_0 + r_1)$	0	0	1
	0	$((\alpha_1[i] \ggg r_0) \ll 1) \ggg r_1$	0	0	2^{-1}
	0	$(\alpha_1[i] \ll 1) \ggg (r_0 + r_1)$	0	0	2^{-1}
	0	$(((\alpha_1[i] \ll 1) \ggg r_0) \ll 1) \ggg r_1$	0	0	2^{-2}
	0	0	$\alpha_1[i] \ggg r_0$	0	1
	0	0	$(\alpha_1[i] \ggg r_0) \ll 1$	0	2^{-1}
	0	0	$(\alpha_1[i] \ll 1) \ggg r_0$	0	2^{-1}
	0	0	$((\alpha_1[i] \ll 1) \ggg r_0) \ll 1$	0	2^{-2}
	0	0	0	$\alpha_1[i] \ggg r_0$	1
	0	0	0	$(\alpha_1[i] \ll 1) \ggg r_0$	2^{-1}
$\alpha_2[i]$	0	$\alpha_2[i] \ggg r_1$	0	0	1
	0	$(\alpha_2[i] \ll 1) \ggg r_1$	0	0	2^{-1}
	0	0	$\alpha_2[i]$	0	1
	0	0	$\alpha_2[i] \ll 1$	0	2^{-1}
$\alpha_3[i]$	0	$\alpha_3[i] \ggg (r_0 + r_1)$	0	0	1
	0	$(\alpha_3[i] \ll 1) \ggg (r_0 + r_1)$	0	0	2^{-1}
	0	0	$\alpha_3[i] \ggg r_0$	0	1
	0	0	$(\alpha_3[i] \ll 1) \ggg r_0$	0	2^{-1}
	0	0	0	$\alpha_3[i] \ggg r_0$	1

Table 6.1: Output differences $\beta_w[j]$ and their probabilities after G_1 on an input difference $\alpha_v[i]$.

	$a_0[i]$	$a_1[i]$	$a_2[i]$	$a_3[i]$
max. hw(γ_w)	102	115	34	39

Table 6.2: Maximum Hamming weight of an output difference γ_w after one application of G on an input difference $\alpha_v[i]$.

find for an adversary than differentials on all b bits of the state. To find such differentials we start from our previous analysis of G and extend it to F^R. First, we observe that it is easy to track differences during the first few steps, and in particular to find probability-1 (truncated) differential characteristics for a small number of iterations of F.

For example, by setting the active bit in the MSB of one of the input words a, b, c or d of G a lot of differences are erased due to the shift operation $\ll 1$, as already noted previously. Concretely, using two input words with the input difference $\alpha_0[63]$, i.e. the MSB being active in input a, six of the twelve output differences of G_1 (!) are erased by $\ll 1$ (cf. Table 6.1). As the shift is applied to the non-linear part of G a lot of non-probability-1 differences are deleted, while mainly probability-1 differences remain. Additionally, if distinct input words have active bits in the same positions it leads to further cancellations. Using this simple strategy we found three notable differentials for G of high probability and with low weight output differences:

$$\begin{aligned}
&(\mathtt{8000000000000000}, \mathtt{8000000000000000}, \mathtt{8000000000000000}, \mathtt{0000000000000000}) \xrightarrow[1]{\mathrm{G}}\\
&(\mathtt{0000000000000000}, \mathtt{0000000000000001}, \mathtt{8000000000000000}, \mathtt{0000000000000000})\\[1ex]
&(\mathtt{0000000000000000}, \mathtt{8000000000000000}, \mathtt{8000000000000000}, \mathtt{8000000000000000}) \xrightarrow[2^{-1}]{\mathrm{G}}\\
&(\mathtt{8000000000000000}, \mathtt{0000000001000001}, \mathtt{8000000000800000}, \mathtt{0000000000800000})\\[1ex]
&(\mathtt{0000000000000000}, \mathtt{8000000000000000}, \mathtt{8000000000000000}, \mathtt{8000000000000000}) \xrightarrow[2^{-1}]{\mathrm{G}}\\
&(\mathtt{8000000000000000}, \mathtt{0000000003000001}, \mathtt{8000000001800000}, \mathtt{0000000000800000})
\end{aligned}$$

Applying those differentials to F has the effect that the diffusion of the state is delayed by one step. Note that input differences with other combinations of active MSBs lead to similar output differences, but none with a lower or equal Hamming weight as the above. Using the first of the above differentials, we were able to easily derive a truncated differential over 3 steps (i.e. $\mathrm{F}^{1.5}$), which has probability 1. This truncated differential can be used to construct an impossible differential over 3.5 rounds for the 64-bit version of F, which is shown in the next section.

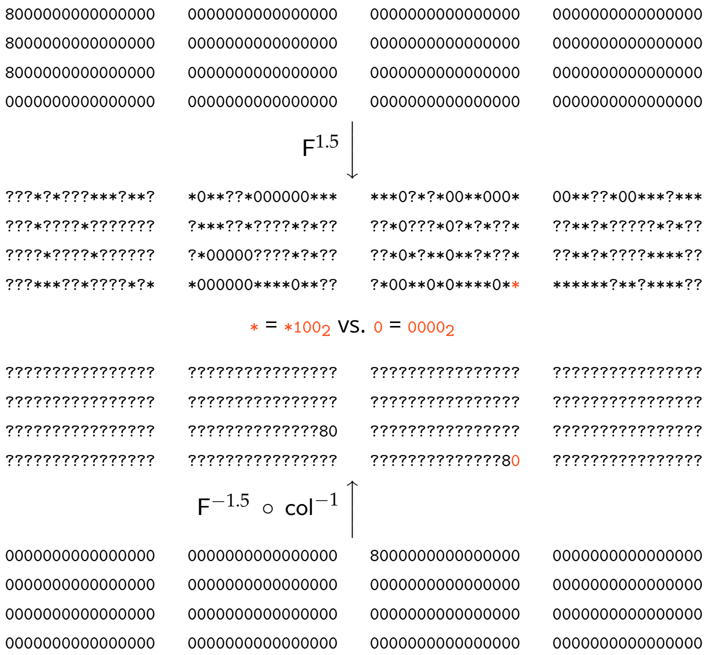
We expect that advanced search techniques are able to find better differential distinguishers for a higher number of iterations of F, such that the sparse difference occurs at a later step than in the first. Nevertheless we expect that it is not possible to find differential distinguishers for as much rounds as specified for our instances, see Table 2.1, taking into account the reduced freedom an adversary has, when attacking the initialisation or round permutation.

6.1.4 Impossible differentials

Cryptanalysis using impossible differentials was introduced in 1998 by Knudsen to attack the block cipher DEAL [37]. Later it was extended by Biham et al. in order to attack the block ciphers Skipjack [22] and IDEA [23]. The latter introduces the so called *miss-in-the-middle* technique. This approach combines two probability 1 differentials, one in forward and one in backward direction which exhibit a conflict when both directions are joined. This strategy leads to an impossible event, i.e. an incident having probability 0, and can be used to construct distinguishers or even mount key recovery attacks.

In our case we construct an impossible differential over 3.5 rounds of the 64-bit version of F, namely 3 steps in forward and 4 steps in backward direction, using the miss-in-the-middle approach from above. An illustration[2] of the used differentials and the resulting conflict is given in Figure 6.1. A $*$ denotes a partially known and a $?$ an unknown entry. Our analysis

[2] We refer to Figure B.2 in the appendix for the bit representation of the output differences.



```
8000000000000000   0000000000000000   0000000000000000   0000000000000000
8000000000000000   0000000000000000   0000000000000000   0000000000000000
8000000000000000   0000000000000000   0000000000000000   0000000000000000
0000000000000000   0000000000000000   0000000000000000   0000000000000000
```

$F^{1.5} \downarrow$

```
???*?*???***?**?   *0**??*000000***   ***0?*?*00**000*   00**??*00***?***
???*????*???????   ?***??*????*?*??   ??*0???*0?*?*??*   ??**?*?????*?*??
????*????*??????   ?*00000????*?*??   ??*0*?**0**?*??*   ??**?*????****??
???***??*????*?*   *000000****0**??   ?*00**0*0****0**   ******?**?****??
```

$* = *100_2$ vs. $0 = 0000_2$

```
????????????????   ????????????????   ????????????????   ????????????????
????????????????   ????????????????   ????????????????   ????????????????
????????????????   ??????????????80   ????????????????   ????????????????
????????????????   ????????????????   ??????????????80   ????????????????
```

$F^{-1.5} \circ \mathrm{col}^{-1} \uparrow$

```
0000000000000000   0000000000000000   8000000000000000   0000000000000000
0000000000000000   0000000000000000   0000000000000000   0000000000000000
0000000000000000   0000000000000000   0000000000000000   0000000000000000
0000000000000000   0000000000000000   0000000000000000   0000000000000000
```

Figure 6.1: An impossible differential over 3.5 rounds of 64-bit F.

shows that the conflict occurs in the 2nd bit of the 14th word. In forward direction this bit has always[3] value 1 whereas in backward direction it has always value 0. Note that there are many more impossible differentials of the above type starting from comparable input differences in forward and backward direction. Nevertheless, using such a simple approach, we were not able to construct impossible differentials stretching over more than 3.5 rounds.

Those impossible differentials cannot be used to attack (round-reduced) NORX, due to the following reasons:

1. The state setup used during initialisation prevents an attacker from setting the required input difference in forward direction. It would be necessary to set differences in the first three consecutive MSBs of a column, which is impossible, as every column is initialised with at least two constant values (see §§2.6.4). Thus, even in the *related-key attack model* it is not possible to exploit this class of impossible differentials.

2. Under the assumption that an attacker is *nonce-respecting* [44] and that F^R provides maximum security for $R \geq 4$, two states being set up with two different nonces lead to two distinct internal states after the initialisation phase. Therefore an attacker does not know how to set header blocks to construct the required input difference in forward direction. The same holds for the payload phase. In summary the impossible differential cannot be exploited at a later phase of the algorithm either.

[3] The impossible differential was validated empirically in about 2^{32} runs.

6.2 Algebraic cryptanalysis

Algebraic attacks on cryptographic algorithms discussed in the literature [5, 7, 27, 30] target ciphers whose internal state is mainly updated in a linear way and thus exploit a low algebraic degree of the attacked primitive. However, this is not the case for NORX, where the b inner state bits are updated in a strongly non-linear fashion. In the following we briefly discuss the non-linearity properties of NORX, demonstrating why it is unlikely that algebraic attacks can be successfully mounted against the cipher.

A convenient way of representing a Boolean function is through its *Algebraic Normal Form* (ANF). Given a Boolean function $f : \{0,1\}^n \longrightarrow \{0,1\}$, the ANF representing f is a multivariate polynomial, i.e. a sum of monomials in n input variables. Both a large number of monomials in the ANF and a good distribution of their degrees are important properties of non-linear building blocks in ciphers.

We constructed the ANF of G and measured the degree of every of the $4W$ polynomials and the distribution of the monomials. Table 6.3 reports the number of polynomials per degree for the 32- and 64-bit versions, as well as information on the distribution of monomials.

	#polynomials by degree						#monomials			
	3	4	5	6	7	8	min	max	avg	median
64-bit	2	6	122	2	8	116	12	489	253	49.5
32-bit	2	6	58	2	8	52	12	489	242	49.5

Table 6.3: Number of polynomials by degree, and number of monomials by polynomial.

In both cases most polynomials have degree 5 or 8 and merely 2 have degree 3. Multiplying each of the above values by 4 gives the distribution of degrees for the ANF of the whole state after one column or diagonal step. Due to memory constraints, we were unable to construct[4] the ANF for a single full round F, neither for the 64-bit nor for the 32-bit version. In summary, this shows that the state of NORX is updated in a strongly non-linear fashion. Due to the rapid growth of the degree and the huge state size of NORX we believe that it is unlikely that algebraic cryptanalysis can be used to successfully mount an attack on the AEAD scheme.

6.3 Other attacks

In this section we briefly review other kinds of attacks that may be used against NORX.

6.3.1 Fixed points

The G permutation and thus any iteration of the round function F have a trivial distinguisher: the fixed points $\mathsf{G}(0) = 0$ and $\mathsf{F}^R(0) = 0$. Nevertheless it seems hard to exploit this property, as hitting the all-zero state is as hard as hitting any other arbitrary state. Thus the ability to hit a predefined state implies the ability to recover the key, which is equivalent to completely breaking NORX. Therefore the zero-to-zero point is no significant threat to the security of NORX.

[4] Using SAGE [46] on a workstation with 64 GiB RAM.

Furthermore, we used the constraint solver STP [31] to prove that there are no further fixed points. For NORX32 the solver was able to show that this is indeed the case, but for NORX64 the proof is a lot more complex. Even after over 1000 hours, STP was unable to finish its computation with a positive or negative result. We find it unlikely that there are any other fixed points in NORX64 besides the zero-to-zero point.

6.3.2 Slide attacks

Slide attacks try to exploit the symmetry in a primitive that consists of the iteration of a number of identical rounds. They were introduced by Biryukov et al. [25, 26] to cryptanalyse block ciphers. Later they were also extended to stream ciphers [42] and hash functions [32]. To protect sponge constructions against slide attacks two simple countermeasures can be found in the literature:

1. In [32] it is proposed to add a non-zero constant to the state just before applying the permutation.
2. In [41] it is recommended to use a message padding, which ensures that the last processed data block is different from the all-zero message.

The duplex constructions is derived from sponge functions, hence the above countermeasures should hold for the former, too, and thus for NORX. Both defensive mechanisms are already integrated into NORX: The domain separation constants, see §§2.6.3, are added to the state just before the permutation F^R is applied and the multi-rate padding, see §§2.6.5, ensures that the last processed data block is different from the all-zero block. Hence, slide attacks should pose little to no threat to NORX.

6.3.3 Rotational cryptanalysis

Rotational cryptanalysis was introduced by Khovratovich and Nikolić in [35] to analyse ARX based primitives. The idea is to track the propagation of rotational relations through a cryptographic transformation. Once rotation-invariant behaviour is detected, it can be used to construct distinguishers, mount key recovery attacks and so on. Rotational cryptanalysis was successfully applied to several simplified cryptographic primitives including Skein [36] and Keccak [39].

NORX includes several defense mechanisms to increase the difficulty of finding exploitable rotation-invariant behaviour:

1. During state setup 10 out of 16 words are initialised with asymmetric constants, which impedes the occurrence of rotation-invariant behaviour and limits the freedom of an attacker. A similar approach is also used in Salsa20 [13].
2. The non-linear operation of NORX contains a non rotation-invariant bit-shift $\ll 1$.
3. NORX is based on the duplex construction, which prevents an attacker from modifying the complete internal state at a given time. He is only able to influence the rate bits, i.e. at most $r = 10W$ bits of the state, and has to “guess” the other $6W$ bits in order to mount an attack.

7 Intellectual property

We, the designers of NORX, do hereby declare that

- NORX is free for everyone to use;
- We are not aware of any patent or patent application that may cover the practice of the NORX algorithm;
- We have not filed any patent application related to the NORX algorithm.

If any of this information changes, the submitter/submitters will promptly (and within at most one month) announce these changes on the `crypto-competitions` mailing list.

8 Consent

The submitter/submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitter/submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitter/submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitter/submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitter/-submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitter/submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

9 Acknowledgements

The authors thank Frank K. Gürkaynak, Mauro Salomon, Tibor Keresztfalvi and Christoph Keller for implementing NORX in hardware and for giving insightful feedback from their hardware evaluation.

Moreover, the authors would like to thank Alexander Peslyak (Solar Designer), for giving them access to one of his Haswell machines, so that they could test their AVX2 implementations of NORX.

Bibliography

[1] eSTREAM - the ECRYPT Stream Cipher Project, 2004–2008. http://www.ecrypt.eu.org/stream.

[2] SHA-3 Competition, 2007–2012. http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/index.html.

[3] CAESAR — Competition for Authenticated Encryption: Security, Applicability, and Robustness, 2013. http://competitions.cr.yp.to/caesar.html.

[4] Martin Albrecht, Nicolas T. Courtois, Daniel Hulme, and Guangyan Song. Bit-Slice Implementation of PRESENT in Pure Standard C, v1.5, 2011. Opensource code available at https://bitbucket.org/malb/research-snippets/src.

[5] Frederik Armknecht. On the Existence of Low-Degree Equations for Algebraic Attacks. Cryptology ePrint Archive, Report 2004/185, 2004. http://eprint.iacr.org/2004/185.

[6] Jörg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010. http://jjj.de/fxt/fxtpage.html#fxtbook.

[7] Jean-Philippe Aumasson, Itai Dinur, Luca Henzen, Willi Meier, and Adi Shamir. Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128. Cryptology ePrint Archive, Report 2009/218.

[8] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New Features of Latin Dances: Analysis of Salsa, ChaCha and Rumba. In Kaisa Nyberg, editor, *FSE 2008*, volume 5086 of *LNCS*, pages 470–488. Springer, Heidelberg, 2008.

[9] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 Proposal BLAKE. In *NIST SHA-3 Proposal*, 2010. https://131002.net/blake.

[10] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, Smaller, Fast as MD5. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 2013*, volume 7954 of *LNCS*, pages 119–135. Springer, Heidelberg, 2013.

[11] Michael Beeler, R. William Gosper, and Richard Schroeppel. HAKMEM. Artificial Intelligence Memo 239, Massachusetts Institute of Technology, February 1972. http://dspace.mit.edu/handle/1721.1/6086.

[12] Daniel J. Bernstein. Cache-Timing Attacks on AES, 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[13] Daniel J. Bernstein. Salsa20 Security, 2005. http://cr.yp.to/snuffle/security.pdf.

[14] Daniel J. Bernstein. ChaCha, a Variant of Salsa20. In *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*, 2008. http://cr.yp.to/chacha.html.

[15] Daniel J. Bernstein. The Salsa20 Family of Stream Ciphers. In Matthew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs*, volume 4986 of *LNCS*, pages 84–97. Springer, Heidelberg, 2008.

[16] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the Security of Keyed Sponge Constructions. Presented at SKEW 2011, 16–17 February 2011, Lyngby, Denmark, http://sponge.noekeon.org/SpongeKeyed.pdf.

[17] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Permutation-based Encryption, Authentication and Authenticated Encryption. Presented at DIAC 2012, 05–06 July 2012, Stockholm, Sweden.

[18] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Cryptographic Sponge Functions, 2008. http://sponge.noekeon.org/CSF-0.1.pdf.

[19] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In Nigel Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 181–197. Springer, Heidelberg, 2008.

[20] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In A. Miri and S. Vaudenay, editors, *SAC 2011*, volume 7118 of *LNCS*, pages 320–337. Springer, Heidelberg, 2011.

[21] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny van Keer. Keccak implementation overview, May 2012. http://keccak.noekeon.org.

[22] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In Jacques Stern, editor, *EUROCRYPT 1999*, volume 1592 of *LNCS*, pages 12–23. Springer, Heidelberg, 1999.

[23] Eli Biham, Alex Biryukov, and Adi Shamir. Miss in the Middle Attacks on IDEA and Khufu. In Lars Knudsen, editor, *FSE 1999*, volume 1636 of *LNCS*, pages 124–138. Springer, Heidelberg, 1999.

[24] Alex Biryukov and Dmitry Khovratovich. PPAE: Parallelizable Permutation-based Authenticated Encryption. Presented at DIAC 2013, 11–13 August 2013, Chicago, USA, http://2013.diac.cr.yp.to/slides/khovratovich.pdf.

[25] Alex Biryukov and David Wagner. Slide Attacks. In Lars Knudsen, editor, *FSE 1999*, volume 1636 of *LNCS*, pages 245–259. Springer, Heidelberg, 1999.

[26] Alex Biryukov and David Wagner. Advanced Slide Attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 589–606. Springer, Heidelberg, 2000.

[27] Nicolas T. Courtois. Algebraic Attacks on Combiners with Memory and Several Outputs. In Choon sik Park and Seongtaek Chee, editors, *Information Security and Cryptology (ICISC)*, volume 3506 of *LNCS*, pages 3–20. Springer, Heidelberg, 2004. http://eprint.iacr.org/2003/125.

[28] Nicolas T. Courtois, Daniel Hulme, and Theodosis Mourouzis. Solving Circuit Optimisation Problems in Cryptography and Cryptanalysis. In *SHARCS*, 2012. http://eprint.iacr.org/2011/475.

[29] Joan Daemen and Vincent Rijmen. The Advanced Encryption Standard, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[30] Itai Dinur and Adi Shamir. Cube Attacks on Tweakable Black Box Polynomials. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 278–299. Springer, Heidelberg, 2009.

[31] Vijay Ganesh, Ryan Govostes, Khoo Yit Phang, Mate Soos, and Ed Schwartz. *STP — A Simple Theorem Prover*, 2006–2013. http://stp.github.io/stp.

[32] Michael Gorski, Stefan Lucks, and Thomas Peyrin. Slide Attacks on a Class of Hash Functions. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 143–160. Springer, Heidelberg, 2008.

[33] Luca Henzen, Flavio Carbognani, Norbert Felber, and Wolfgang Fichtner. VLSI Hardware Evaluation of the Stream Ciphers Salsa20 and ChaCha, and the Compression Function Rumba. In *2nd International Conference on Signals, Circuits and Systems 2008*, pages 1–5. IEEE, 2008.

[34] Antoine Joux. Authentication Failures in NIST Version of GCM, 2006. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/Joux_comments.pdf.

[35] Dmitry Khovratovich and Ivica Nikolić. Rotational Cryptanalysis of ARX. In Seokhie Hong and Tetsu Iwata, editors, *FSE 2010*, volume 6147 of *LNCS*, pages 333–346. Springer, Heidelberg, 2010.

[36] Dmitry Khovratovich, Ivica Nikolić, and Christian Rechberger. Rotational Rebound Attacks on Reduced Skein. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *LNCS*, pages 1–19. Springer, Heidelberg, 2010.

[37] Lars R. Knudsen. DEAL — A 128-bit Block Cipher. In *NIST AES Proposal*, 1998.

[38] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*, volume 4A. Addison-Wesley, Upper Saddle River, New Jersey, 2011. http://www-cs-faculty.stanford.edu/~uno/taocp.html.

[39] Pawel Morawiecki, Josef Pieprzyk, and Marian Srebrny. Rotational Cryptanalysis of Round-Reduced Keccak. Cryptology ePrint Archive, Report 2012/546, 2012. http://eprint.iacr.org/2012/546.

[40] National Institute of Standards and Technology. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, 2007. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

[41] Thomas Peyrin. Security Analysis of Extended Sponge Functions. Presented at the ECRYPT Workshop Hash Functions in Cryptology: Theory and Practice, Leiden, The Netherlands, June 4th 2008, http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Peyrin.pdf.

[42] Deike Priemuth-Schmid and Alex Biryukov. Slid Pairs in Salsa20 and Trivium. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Indocrypt*, volume 5365 of *LNCS*, pages 1–14. Springer, Heidelberg, 2008. http://eprint.iacr.org/2008/405.

[43] Phillip Rogaway. Authenticated-Encryption with Associated-Data. In *ACM Conference on Computer and Communications Security (CCS'02)*, pages 98–107. ACM press, 2002.

[44] Phillip Rogaway. Nonce-Based Symmetric Encryption. In Bimal Roy and Willi Meier, editors, *FSE 2004*, volume 3017 of *LNCS*, pages 348–358. Springer, Heidelberg, 2004.

[45] Zhenqing Shi, Bin Zhang, Dengguo Feng, and Wenling Wu. Improved Key Recovery Attacks on Reduced Round Salsa20 and ChaCha. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC 2012*, volume 7839 of *LNCS*, pages 337–351. Springer, Heidelberg, 2012.

[46] W. A. Stein. *Sage Mathematics Software*. The Sage Development Team, 2005–2013. http://sagemath.org.

[47] Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki, and Hiroki Nakashima. Differential Cryptanalysis of Salsa20/8. In *The State of the Art of Stream Ciphers (SASC)*, 2007.

A Test Vectors and Intermediate Values

All of the following test vectors and intermediate values are denoted in little-endian format. Order of values is left to right, top to bottom.

A.1 Traces for G

$W = 32$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{00000000}, \mathtt{00000000}, \mathtt{00000000}, \mathtt{00000000})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{00000000}, \mathtt{00000000}, \mathtt{00000000}, \mathtt{00000000})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{00000001}, \mathtt{00000000}, \mathtt{00000000}, \mathtt{00000000})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{00002001}, \mathtt{42024200}, \mathtt{21010100}, \mathtt{20010100})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{00000000}, \mathtt{00000001}, \mathtt{00000000}, \mathtt{00000000})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{00202001}, \mathtt{42424240}, \mathtt{21010120}, \mathtt{20010120})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{00000000}, \mathtt{00000000}, \mathtt{00000001}, \mathtt{00000000})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{00200000}, \mathtt{00400042}, \mathtt{00000021}, \mathtt{00000020})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{00000000}, \mathtt{00000000}, \mathtt{00000000}, \mathtt{00000001})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{00002000}, \mathtt{42004200}, \mathtt{21000100}, \mathtt{20000100})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{80000000}, \mathtt{00000000}, \mathtt{00000000}, \mathtt{00000000})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{80001000}, \mathtt{21012100}, \mathtt{10808080}, \mathtt{10008080})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{00000000}, \mathtt{80000000}, \mathtt{00000000}, \mathtt{00000000})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{80101000}, \mathtt{21212120}, \mathtt{10808090}, \mathtt{10008090})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{00000000}, \mathtt{00000000}, \mathtt{80000000}, \mathtt{00000000})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{00100000}, \mathtt{00200021}, \mathtt{80000010}, \mathtt{00000010})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{00000000}, \mathtt{00000000}, \mathtt{00000000}, \mathtt{80000000})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{00001000}, \mathtt{21002100}, \mathtt{10800080}, \mathtt{10000080})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{FFFFFFFF}, \mathtt{FFFFFFFF}, \mathtt{FFFFFFFF}, \mathtt{FFFFFFFF})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{FFFF5FFE}, \mathtt{35F939FC}, \mathtt{1AFCFCFE}, \mathtt{5FFEFEFF})\end{aligned}$$

$$\begin{aligned}(a,b,c,d) &= (\mathtt{01234567}, \mathtt{89ABCDEF}, \mathtt{FEDCBA98}, \mathtt{7654321F})\\ \mathsf{G}(a,b,c,d) &= (\mathtt{B7BF8099}, \mathtt{65A6E720}, \mathtt{1E22F5Cb}, \mathtt{1AA9E143})\end{aligned}$$

$W = 64$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{0000000000000001}, \texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{0000002000000001}, \texttt{4200004002020000}, \texttt{2100000001010000}, \texttt{2000000001010000})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{0000000000000000}, \texttt{0000000000000001}, \texttt{0000000000000000}, \texttt{0000000000000000}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{0000202000000001}, \texttt{4200404002020040}, \texttt{2100000001010020}, \texttt{2000000001010020})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000001}, \texttt{0000000000000000}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{0000200000000000}, \texttt{0000400000000042}, \texttt{0000000000000021}, \texttt{0000000000000020})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000001}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{0000002000000000}, \texttt{4200004000020000}, \texttt{2100000000010000}, \texttt{2000000000010000})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{8000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{8000001000000000}, \texttt{2100002001010000}, \texttt{1080000000808000}, \texttt{1000000000808000})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{0000000000000000}, \texttt{8000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{8000101000000000}, \texttt{2100202001010020}, \texttt{1080000000808010}, \texttt{1000000000808010})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{8000000000000000}, \texttt{0000000000000000}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{0000100000000000}, \texttt{0000200000000021}, \texttt{8000000000000010}, \texttt{0000000000000010})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{0000000000000000}, \texttt{8000000000000000}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{0000001000000000}, \texttt{2100002000010000}, \texttt{1080000000008000}, \texttt{1000000000008000})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{FFFFFFFFFFFFFFFF}, \texttt{FFFFFFFFFFFFFFFF}, \texttt{FFFFFFFFFFFFFFFF}, \texttt{FFFFFFFFFFFFFFFF}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{FFFFFF5FFFFFFFFE}, \texttt{35FFFF3FF9F9FFFC}, \texttt{1AFFFFFFFCFCFFFE}, \texttt{5FFFFFFFFEFEFFFF})
\end{aligned}$$

$$\begin{aligned}
(a,b,c,d) &= (\texttt{0123456789ABCDEF}, \texttt{FEDCBA9876543210}, \texttt{0123456789ABCDEF}, \texttt{FEDCBA9876543210}) \\
\mathsf{G}(a,b,c,d) &= (\texttt{06E0F91F53B5CA4B}, \texttt{1D4225AFF0B8887D}, \texttt{26541088639A5752}, \texttt{5A343C6186E9E1DA})
\end{aligned}$$

A.2 Traces for F

$W = 32$

$$S = \begin{pmatrix} \texttt{00000001} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \\ \texttt{00000000} & \texttt{00000000} & \texttt{00000000} & \texttt{00000000} \end{pmatrix}$$

$$\mathsf{F}(S) = \begin{pmatrix} \texttt{04004001} & \texttt{20200400} & \texttt{20042020} & \texttt{4A4A8A08} \\ \texttt{01880885} & \texttt{8A424A40} & \texttt{4A024A02} & \texttt{C24A0248} \\ \texttt{41212104} & \texttt{888C4C4A} & \texttt{41210520} & \texttt{05212101} \\ \texttt{05012000} & \texttt{20202004} & \texttt{884A4A08} & \texttt{40210500} \end{pmatrix}$$

$$\mathsf{F}^2(S) = \begin{pmatrix} \texttt{EFDB6055} & \texttt{4EB0C8FD} & \texttt{4D66BAD5} & \texttt{A5716F6F} \\ \texttt{3315BA06} & \texttt{B5E09122} & \texttt{44A18E71} & \texttt{51E36297} \\ \texttt{F137B870} & \texttt{3C7265F6} & \texttt{00C30D5B} & \texttt{295A09AA} \\ \texttt{B42B85E7} & \texttt{AC007723} & \texttt{742077A7} & \texttt{4BADCF9B} \end{pmatrix}$$

$$\mathsf{F}^3(S) = \begin{pmatrix} \texttt{B49E8FA1} & \texttt{B87AED22} & \texttt{86152D27} & \texttt{BEB398AD} \\ \texttt{BD48EB80} & \texttt{1D4447DA} & \texttt{B7458BA9} & \texttt{A9E9EF9B} \\ \texttt{F7599C6A} & \texttt{203FB309} & \texttt{694A1283} & \texttt{C4875743} \\ \texttt{F4E78B62} & \texttt{50BE8206} & \texttt{7BEF5DF7} & \texttt{F92F6B9C} \end{pmatrix}$$

$$\mathsf{F}^4(S) = \begin{pmatrix} \texttt{D8936EA9} & \texttt{4FDFA7F9} & \texttt{2E23D116} & \texttt{ED7C3692} \\ \texttt{3E463C40} & \texttt{A5AA5D55} & \texttt{A05A6E11} & \texttt{D22C7D58} \\ \texttt{3C0D461D} & \texttt{5D78E74F} & \texttt{88C9121B} & \texttt{ECA4CA13} \\ \texttt{E12928CB} & \texttt{0167E06D} & \texttt{90E1494E} & \texttt{7CBBCCDA} \end{pmatrix}$$

$$\mathsf{F}^5(S) = \begin{pmatrix} \texttt{DC4D4AE5} & \texttt{2EA22D30} & \texttt{0F46317D} & \texttt{61B76178} \\ \texttt{317CF942} & \texttt{AA617101} & \texttt{B1B646B0} & \texttt{9FB8201C} \\ \texttt{31E77E87} & \texttt{0E87682D} & \texttt{AB27674A} & \texttt{1C00EF33} \\ \texttt{49676DA0} & \texttt{5E36BB3F} & \texttt{369CB43A} & \texttt{F6E575E8} \end{pmatrix}$$

$$\mathsf{F}^6(S) = \begin{pmatrix} \texttt{472112C6} & \texttt{EBBA21DD} & \texttt{69FAF1B0} & \texttt{06AADA3C} \\ \texttt{958968BA} & \texttt{FAF43AF0} & \texttt{8A346D6C} & \texttt{04DAD629} \\ \texttt{28C63C70} & \texttt{F49BAA13} & \texttt{57DE5F7C} & \texttt{28841E18} \\ \texttt{EA3F594F} & \texttt{8D744A62} & \texttt{57B54FF1} & \texttt{753A4160} \end{pmatrix}$$

$$\mathsf{F}^7(S) = \begin{pmatrix} \texttt{865ACF57} & \texttt{0B1CD341} & \texttt{44571AAD} & \texttt{1E351C75} \\ \texttt{679AB711} & \texttt{8D923CDC} & \texttt{115DC180} & \texttt{CF5E7435} \\ \texttt{94D66EB3} & \texttt{6B643DA7} & \texttt{C71FD3A8} & \texttt{EACD114A} \\ \texttt{FE5A4582} & \texttt{101A0A61} & \texttt{DEF929CE} & \texttt{F81307CE} \end{pmatrix}$$

$$\mathsf{F}^8(S) = \begin{pmatrix} \texttt{EE830EF5} & \texttt{EFEDB52C} & \texttt{D9B5DDE0} & \texttt{11699703} \\ \texttt{A59F827F} & \texttt{E7DA769E} & \texttt{9ACF9688} & \texttt{FE6B4EE6} \\ \texttt{2D99EFFF} & \texttt{C1F42728} & \texttt{1B33FCE4} & \texttt{2484C32D} \\ \texttt{454DEF51} & \texttt{65220E90} & \texttt{D8B53023} & \texttt{10265221} \end{pmatrix}$$

$W = 64$

$$S = \begin{pmatrix} \texttt{0000000000000001} & \texttt{0000000000000000} & \texttt{0000000000000000} & \texttt{0000000000000000} \\ \texttt{0000000000000000} & \texttt{0000000000000000} & \texttt{0000000000000000} & \texttt{0000000000000000} \\ \texttt{0000000000000000} & \texttt{0000000000000000} & \texttt{0000000000000000} & \texttt{0000000000000000} \\ \texttt{0000000000000000} & \texttt{0000000000000000} & \texttt{0000000000000000} & \texttt{0000000000000000} \end{pmatrix}$$

$$\mathsf{F}(S) = \begin{pmatrix} \texttt{0000004000000401} & \texttt{2020000400000000} & \texttt{2000042000000020} & \texttt{42400888420A0840} \\ \texttt{1008008580981891} & \texttt{8240004842020800} & \texttt{4800020A00420200} & \texttt{C200084042420048} \\ \texttt{4100000021210004} & \texttt{8844080A80440408} & \texttt{4120000421010000} & \texttt{0420010100210100} \\ \texttt{0400010100200000} & \texttt{2000000020200004} & \texttt{8802080A40420208} & \texttt{4020000401010000} \end{pmatrix}$$

$$\mathsf{F}^2(S) = \begin{pmatrix} \texttt{9D802FD127A732A1} & \texttt{BFDC94FCF7EDB4F6} & \texttt{50E28C54A198AD0E} & \texttt{09FCDB8FCCC9DDA8} \\ \texttt{7ACEC81E5BAA6D25} & \texttt{10C9CBCF5BFEFC27} & \texttt{11A152F2C1A43FCA} & \texttt{6BA77CCFA2D9F407} \\ \texttt{0E03AD8E4F36AD96} & \texttt{B405D697E680A2BB} & \texttt{3651B1301374F05D} & \texttt{EC2A3CD28E701034} \\ \texttt{D793C96953AA22B3} & \texttt{81B56FC8F78827DD} & \texttt{A5F18C894182A861} & \texttt{F95F620C599E1A7D} \end{pmatrix}$$

$$\mathsf{F}^3(S) = \begin{pmatrix} \texttt{6D9C774FB118B930} & \texttt{0AD4888256442919} & \texttt{B2625AFA68288616} & \texttt{3F682524B541B12D} \\ \texttt{09FB30C77ED1253C} & \texttt{D276B00A56FA3BB2} & \texttt{D1A3ED2B432628E0} & \texttt{59DE47C408703466} \\ \texttt{730C85F6CF7CD9B4} & \texttt{D731F331C620402D} & \texttt{664456562656A61E} & \texttt{10F001A72ABF1CCA} \\ \texttt{E04F26164B84BCD5} & \texttt{E1CE43EA4AC71790} & \texttt{BE0A7BDA26AB8C3E} & \texttt{083CB972BE746F0D} \end{pmatrix}$$

$$\mathsf{F}^4(S) = \begin{pmatrix} \texttt{9AE671BAC4106A33} & \texttt{2532A3AF80EB8C24} & \texttt{8807B8748AAF89BB} & \texttt{CCBD275D7AC0180C} \\ \texttt{9E3C9A644E2EE2B1} & \texttt{6EF830BF37A17BB2} & \texttt{A56A3F09DA96ABC9} & \texttt{6674A590854EA97D} \\ \texttt{D58BFB1A8D2677C5} & \texttt{5696D8DEA26A6D6D} & \texttt{2E973803C96922A4} & \texttt{9C8EC44641A390FD} \\ \texttt{ABE2F120F069F77A} & \texttt{305FE9E02B725884} & \texttt{1D2A9380316FE1A6} & \texttt{8FA5B15C10F77415} \end{pmatrix}$$

$$\mathsf{F}^5(S) = \begin{pmatrix} \texttt{E7BC1BB342393A06} & \texttt{4497F473D8AE5B3A} & \texttt{238B885A51663B54} & \texttt{FCFD9F88948D42A7} \\ \texttt{5B6E332077A59C5D} & \texttt{C798AA981789AC8D} & \texttt{F916664458B5AD3F} & \texttt{F7086A16B2407A56} \\ \texttt{8DD6CEC45AC62D09} & \texttt{2C217A7DC1AB282C} & \texttt{8AA14855B8A7A065} & \texttt{1BA096650A8E8F6D} \\ \texttt{9ECAB9E7A91D59FE} & \texttt{A57F363A65CF10D3} & \texttt{F16FCED7A605DFE9} & \texttt{C02D0A46B23E8C31} \end{pmatrix}$$

$$\mathsf{F}^6(S) = \begin{pmatrix} \texttt{2FCA68C9B1691627} & \texttt{59E2B79D4B2A88F8} & \texttt{D44A3CC624C9028F} & \texttt{6295CCEC81F0F5AF} \\ \texttt{AFBA11EEC8CE43A4} & \texttt{A6BC58426BDAB6AC} & \texttt{C9FA0754D15A38A6} & \texttt{61B7C093B862D551} \\ \texttt{B7A8A66A9227EE06} & \texttt{17BEF1A5F98B7250} & \texttt{CCAA13033F5ADCD3} & \texttt{15CBCEF3A8A993B5} \\ \texttt{2E321403DA39690B} & \texttt{D805E663071507B0} & \texttt{6D7EBAA185FF9F07} & \texttt{64071C2C7A0205EA} \end{pmatrix}$$

$$\mathsf{F}^7(S) = \begin{pmatrix} \texttt{BF643FF50F9B521B} & \texttt{D6ECDEF9B9AC18B0} & \texttt{29C44312EB0ED72A} & \texttt{6AA97E4B4BF39E0A} \\ \texttt{A957D54C2B38DF1B} & \texttt{23E4928A7504F6B8} & \texttt{6CFEE0C2D418DC84} & \texttt{10464EB477E6D548} \\ \texttt{18A96DABB8BBC145} & \texttt{406A6EE1C806F1E4} & \texttt{A54BD0A7B7291B4A} & \texttt{27BC2F8593DD77BE} \\ \texttt{3BE8FF6116D7AFB0} & \texttt{4D78AEB59B3A9C25} & \texttt{9F03C664A44601DC} & \texttt{DDBE9B34DA020E59} \end{pmatrix}$$

$$\mathsf{F}^8(S) = \begin{pmatrix} \texttt{F51507DD9E95189F} & \texttt{AB5E0B1641FAD08F} & \texttt{09B7BF70943B60DE} & \texttt{E35D03636672DACD} \\ \texttt{1D013C731A134DCD} & \texttt{850FC95D9CA677C8} & \texttt{48D78D3658CBE8D0} & \texttt{3898A93514FBF49D} \\ \texttt{8849E2B60F59D433} & \texttt{A1C7E702A391D4B9} & \texttt{C0057990DE07D3EE} & \texttt{6BBF9A8B0E6CB108} \\ \texttt{7DE67998BA91A9CE} & \texttt{68F2B4BC4B8F6A52} & \texttt{4EFE2C5711E64647} & \texttt{27173B06EFB20807} \end{pmatrix}$$

A.3 Full AEAD computations

Unless stated otherwise, intermediate values are snapshots of the state after the final permutation of a given phase. For example, for $D = 1$ the *end of the header processing* denotes the state after final permutation F^R in the header phase or in other words, it is the state before the first payload data block is absorbed. This corresponds to the state after the third application of F^R in Figure 2.1, assuming that two header blocks are processed.

A.3.1 Values for NORX32

We assume that the following input data is given:

K :	00112233	44556677	8899AABB	CCDDEEFF
N :	FFFFFFFF	FFFFFFFF		
H :	10000002	30000004		
P :	80000007	60000005	40000003	20000001
T :	null			

Padded header and payload

$\mathsf{pad}_{320}(H)$:	10000002	30000004	00000001	00000000
	00000000	00000000	00000000	00000000
	00000000	80000000		
$\mathsf{pad}_{320}(P)$:	80000007	60000005	40000003	20000001
	00000001	00000000	00000000	00000000
	00000000	80000000		

Basic state setup

243F6A88	FFFFFFFF	FFFFFFFF	85A308D3
00112233	44556677	8899AABB	CCDDEEFF
13198A2E	03707344	254F537A	38531D48
839C6E83	F97A3AE5	8C91D88C	11EAFB59

Note, that the basic state is the same for all of the following instances.

NORX32-4-1

End of initialisation

45079318	15859046	2D54327F	05340C5E
25968B63	D63C2815	4E56477B	67296814
74E8F429	063B7AC4	CE0B7244	F1ED4AAE
7A07FD03	143815BE	C62D6471	79949917

End of header processing

9F8F35CA	AAFA2A3D	324C1414	028732CB
428AEE2C	C6CB8013	DD7DB211	334C8138
A204652A	2AD0B71F	693ACC09	22A8BF7A
2E0239D6	31112F7F	8B542A96	009A5230

End of payload processing

```
7702CA8A  E8BA5210  FD9B73AD  C0443A0D
82B2A588  C1220339  5E40B34C  0F29A284
E1F07668  9A8124DF  529A07FA  46750AB6
4985A971  9A42DF66  7FC2E9B4  68D4D56D
```

Ciphertext and authentication tag

```
C :  1F8F35CD  CAFA2A38  724C1417  228732CA
A :  7702CA8A  E8BA5210  FD9B73AD  C0443A0D
```

NORX32-6-1

End of initialisation

```
3520362B  EFEEB49F  CD8C723D  6895CCFF
6A328C16  EA2E58C3  95324CE4  067097EF
B27A6CCE  85A6D78F  5C48D4A7  BA30CF41
ADDB0BA0  6840050D  6F9519FA  F984872C
```

End of header processing

```
598EDABD  45C18DDC  E0CA4C35  D73309C7
52B0EEED  6C68E782  21F94224  ECC9AE6D
405C64B9  8C7F1F06  16288D77  8F8A701F
6EF702CB  ECE8A30C  9958980B  A9C2C80B
```

End of payload processing

```
69872EE5  3DAC068C  E8D6D8B3  0A3D2099
172CC220  D60C8413  2E44AB22  B84CBD3B
30F66F19  789B5878  B96BFA8D  7A09BAD1
D6218207  5DFA310F  745DD644  7E3CE144
```

Ciphertext and authentication tag

```
C :  D98EDABA  25C18DD9  A0CA4C36  F73309C6
A :  69872EE5  3DAC068C  E8D6D8B3  0A3D2099
```

A.3.2 Values for NORX64

We assume that the following input data is given:

```
K :  0011223344556677   8899AABBCCDDEEFF   FFEEDDCCBBAA9988   7766554433221100
N :  FFFFFFFFFFFFFFFF   FFFFFFFFFFFFFFFF
H :  1000000000000002   3000000000000004
P :  8000000000000007   6000000000000005   4000000000000003   2000000000000001
T :        null
```

Padded header and payload data

$\mathrm{pad}_{640}(H)$:

```
1000000000000002    3000000000000004    0000000000000001    0000000000000000
0000000000000000    0000000000000000    0000000000000000    0000000000000000
0000000000000000    8000000000000000
```

$\mathrm{pad}_{640}(P)$:

```
8000000000000007    6000000000000005    4000000000000003    2000000000000001
0000000000000001    0000000000000000    0000000000000000    0000000000000000
0000000000000000    8000000000000000
```

Basic state setup

```
243F6A8885A308D3    FFFFFFFFFFFFFFFF    FFFFFFFFFFFFFFFF    13198A2E03707344
0011223344556677    8899AABBCCDDEEFF    FFEEDDCCBBAA9988    7766554433221100
A4093822299F31D0    082EFA98EC4E6C89    AE8858DC339325A1    670A134EE52D7FA6
C4316D80CD967541    D21DFBF8B630B762    375A18D261E7F892    343D1F187D92285B
```

Note, that the basic state is the same for all of the following instances.

NORX64-4-1

End of Initialisation

```
D022DC01B3866148    89FCA8C1843BF785    30A1ECB49A02AADB    1000C747A44124FB
4E672A9AAF5E921E    16420D6BBA6B9F4E    73CEA9B7998DBF50    9F651146DA367E54
305CB8FED9C0ED60    F722C6F74FCF070B    C245AFF1112C7437    74DFEA28BA0416A9
30A18E5406072314    5D2E526BF85AC19C    85B14AD2AC1C9386    5AB3B60CFCB12F6F
```

End of header processing

```
9B4DCCFF6779A2C4    E65464C856BC4B09    9ADBC58565E16909    0CB12C0BE9D2F044
D0C80C37C3918211    8C5C7C6B40047B0E    C2603917D10C587B    DEFAD74C772BC0B2
D170DF446F28760F    E505EB2E805DA8A7    8BF3FB43E7C2B420    2203D8187F422421
71B1B43879475337    D02C2522F65982A8    05EEDE3E0800601B    2028AC58C98320EC
```

End of payload processing

```
D0CE5276FDEC9F6E    33EE64CE5CCA3ABA    1187C05183464BD0    A0915ECA6FAF8757
A3E9C1D118136885    AB8538FF8C277EE1    E2E41174CAB5FBEA    20B8A68439999754
041372C291B970B3    E07C5BBEF0086609    7933168208A35AE3    552BF5B67660BB00
5F632E79247130C8    D56F0A370A5E8E6A    EDFA24DD54998983    11ECB808957B8F4B
```

Ciphertext and authentication tag

C :

```
1B4DCCFF6779A2C3    865464C856BC4B0C    DADBC58565E1690A    2CB12C0BE9D2F045
```

A :

```
D0CE5276FDEC9F6E    33EE64CE5CCA3ABA    1187C05183464BD0    A0915ECA6FAF8757
```

NORX64-6-1

End of Initialisation

```
20F8D2782DDF6C31    DD69927096686A90    F8CBD5150CDBA438    0EBE20BA1E4DCB1D
1A294BB54962722F    565C8AF32629FDC1    8B428024526F8A61    82967A5203570FE4
B1F72B4736F4CED7    BD266AA84E2C095B    C5C16C9F0BEAD689    18A71CD9A6DE70C6
41BF1B9A3C074F0A    F577BDE52AB57012    4B20530598D0CD33    1E8D98F07FFE8723
```

End of header processing

```
F223675B69C7A933    7EBAB65233E8DC20    EB660E1BF0F3FEEB    51BE33115B333D6C
5D275000BC021125    20EC5593FE81C7F8    6CC5432F5B3E879A    6E1F8413890FD70E
6F7C347636684E70    F14CECB5241C7A1A    35B7ADD82A64187C    B4A91E5E8880A867
E01438E7B2BD3A0F    014CBAC9B987229F    EC5A0A9A719CC814    FD18778BF21FDD38
```

End of payload processing

```
A05D644CCD2C5887    31DE2501AE4FE789    5C153D99943D29A4    98353A0E38D58A93
31CDF7F6E18DFF65    5510B9B54B7C3C18    4598EAFF89AF00B0    82400B0CE2D0E303
67F11561CFB79DD0    06EC3CF4172108FB    9A70B294DC4B00F3    69D44FC00F5EC17E
E3D7597BD0C27D89    39D0BF81B1D7F23E    E914593F1725E80B    0377D9F0A6C4E8B9
```

Ciphertext and authentication tag

```
C :   7223675B69C7A934    1EBAB65233E8DC25    AB660E1BF0F3FEE8    71BE33115B333D6D
A :   A05D644CCD2C5887    31DE2501AE4FE789    5C153D99943D29A4    98353A0E38D58A93
```

NORX64-4-4

Due to the small size of the payload, only lane L_0 is processing actual payload data. Lanes L_1, L_2 and L_3 are processing only padding blocks.

End of initialisation

```
1A89ABA695E14478    2A8948869113AAC0    980E79B43FE58D71    571062C847AF564C
0E6F5777A9E7BD51    B2A95AE36EEE66D9    0B558CB3D4C54D73    DFB0D84743496F37
0903BCFB370F92EE    A49E9E405F37734F    16D82D55E403FC2D    76FDE340B5A5DAF9
BF69D495C9A57C2B    12D15F1738391169    A4D4AE1365987627    FB13B8D9D7DB27C2
```

End of header processing

```
80F9216CD3ECFA69    F2BE76817E8258AB    8CFD7A31EDA2A966    5BF8CBBA4D36791B
136964C7FEE1E9BE    3BD8C9DAF1B8D602    7273E544EF7DE324    227565DF743A2506
1DEA0740ED1671C0    CD6C06D59AAAF9F5    E3035BBBDB4E0CD4    BDAED547402B66E3
921233DFEB162B82    7BE4484009B4BDF2    E35FBC654F676627    FDF37CFC2BDF9AAA
```

End of payload processing L_0

```
4591F9318DFC4A11    32731D622AA600EA    FDED62FD8EC05F5B    2836B1B28EED8EDC
9902A74D7AEC5511    54A6F4ADA38CA811    3BA317627300EAB8    C121C8B978667D8C
25837DF4190AE9E3    ED6A5E21A7E317E1    EEE71FAE8F27EEFD    87449FBCE89CFB7E
8CD2EB4BF5F504D8    F0C3920A3A8ECD8C    90EE2EA98A42515D    B00F0E05E4B67B60
```

End of payload processing L_1

```
BF4B336606F0579B    E839B6688C9AA4E1    AEBB453DDCD47DCA    25F2445949CCDE12
C38FADBFDC029397    0999A856C0ED59D2    9834BF8A824B224C    E5FD5F6B2229A3F4
D213AD1685D54A46    213DF3E20DCBE743    68DD8ACCBE00F900    25710A57E2F8F94F
3B8491313BE60C80    D88B0207C7354964    9C1AD7E81F3BC47E    2DBBC44A824451ED
```

End of payload processing L_2

```
145F870F7727DD13    4A7F6217BF3420E3    D17E70DD11025563    8FDBA2EF5956FA12
388F662997591452    6C34B916B5162351    FACA66C4DD8BA81B    D330A4A5291D91A7
30548C2748542FB4    00CA0A7BA290FEFA    B4C6D022569E91C7    C38B3D835417C1AE
C78EFD3AC729A4C7    C6F532A6F400BF8F    F07CEC6CF249DF70    8FDE1A2E304F8241
```

End of payload processing L_3

```
19A6559E6072DF53    A96A64AD9FF61438    014748A082167AAA    A5AAF4EF434DEC68
56CB062E01F6C0EC    4BD58AAE85F731F5    A16A54D371ABA6F2    B3EF8E28E0332385
5E115E4AB87479D5    C8E84A36D0D37A65    56E722D16455E58F    D9EE43C73F258110
77199C8FF845950A    794D65C385DD9E6B    A2E06A749A1D6BCC    19255666CB1C4D16
```

End of merging

```
118D01C54E8C0A9F    C2402B4DC1951A6F    F7911EFBF3942998    A3CB7559461D838D
A0E95A27F9A5FDB8    22F3308F4DA936AC    55E30CFBC038992E    2AAD744A49A6797D
8431377093D07662    EE8734D52BC81B3A    E56D85EDC6CA2D8A    3D709885E508675B
68D0C0C0D0550D2E    12E9AEF86CF6AC31    6B7F7E81D7B8F57A    865046F538D68BB3
```

Ciphertext and authentication tag

```
C :  4591F9318DFC4A11    32731D622AA600EA    FDED62FD8EC05F5B    2836B1B28EED8EDC
A :  256284AD4965C563    85E32C4244ABCC3A    3E6B49CB02DB6AF6    E885462500F37A1E
```

B Miscellaneous

B.1 Diffusion statistics for inverse round functions

Table B.1 shows the diffusion statistics of the inverse round functions of NORX and ChaCha.

	Inverse NORX32				Inverse ChaCha (32-bit)			
R	min	max	avg	median	min	max	avg	median
1	17	162	49.444	47	17	126	44.776	44
2	160	306	247.737	248	164	304	244.982	246
3	202	307	255.991	256	203	310	255.994	256
4	202	315	256.018	256	200	311	256.022	256
	Inverse NORX64				Inverse ChaCha (64-bit)			
R	min	max	avg	median	min	max	avg	median
1	17	203	51.346	49	17	142	46.129	45
2	262	568	433.742	435	194	543	382.667	383
3	440	593	511.995	512	440	591	511.964	512
4	435	585	512.011	512	433	596	511.991	512

Table B.1: Diffusion statistics for inverse NORX and ChaCha round functions.

B.2 Addenda to cryptanalysis

B.2.1 Visualisation of differentials for G_1

Figure B.1 depicts the relations of the output differences of G_1 for input differences α_i with one active bit. The probability of an output difference in the tree can be computed by multiplying the values on the edges of the path leading from the root to the particular node.

B.2.2 Impossible differential cryptanalysis

Figure B.2 shows the bit representations of the output differences of the impossible differential over 3.5 rounds of NORX64, which was presented in §§6.1.4. The upper matrix illustrates the difference in forward direction and the lower matrix the one in backward direction. Each row corresponds to one of the 64-bit words of the state (denoted in little-endian), beginning with s_0 for the first row and ending with s_{15} for the last row. The conflict occurs in the 2nd bit of the 14th word.

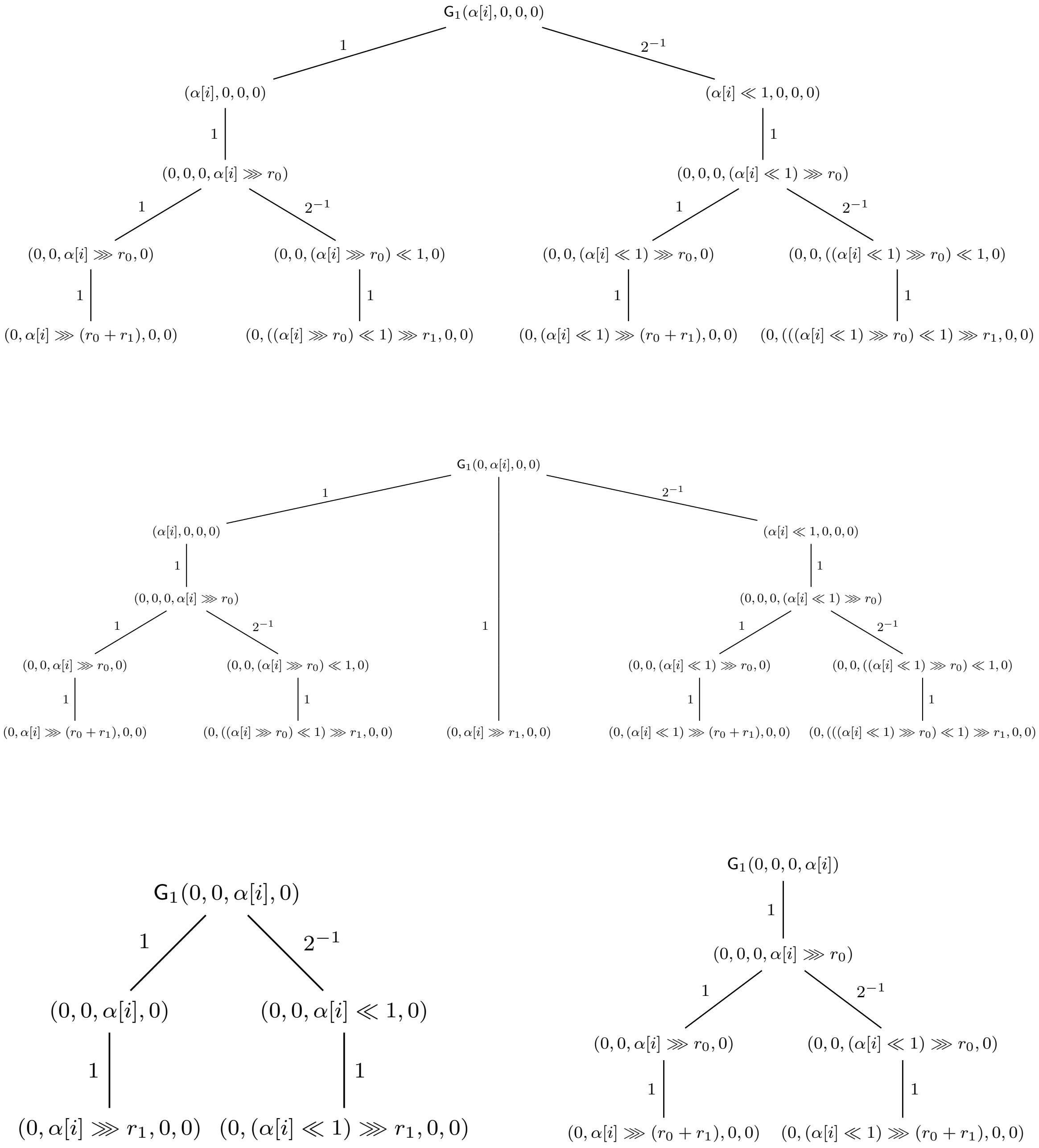

$\mathsf{G}_1(\alpha[i],0,0,0)$
1
2^{-1}
$(\alpha[i],0,0,0)$
$(\alpha[i] \lll 1,0,0,0)$
1
1
$(0,0,0,\alpha[i] \ggg r_0)$
$(0,0,0,(\alpha[i] \lll 1) \ggg r_0)$
1
2^{-1}
1
2^{-1}
$(0,0,\alpha[i] \ggg r_0,0)$
$(0,0,(\alpha[i] \ggg r_0) \lll 1,0)$
$(0,0,(\alpha[i] \lll 1) \ggg r_0,0)$
$(0,0,((\alpha[i] \lll 1) \ggg r_0) \lll 1,0)$
1
1
1
1
$(0,\alpha[i] \ggg (r_0+r_1),0,0)$
$(0,((\alpha[i] \ggg r_0) \lll 1) \ggg r_1,0,0)$
$(0,(\alpha[i] \lll 1) \ggg (r_0+r_1),0,0)$
$(0,(((\alpha[i] \lll 1) \ggg r_0) \lll 1) \ggg r_1,0,0)$
$\mathsf{G}_1(0,\alpha[i],0,0)$
1
2^{-1}
$(\alpha[i],0,0,0)$
$(\alpha[i] \lll 1,0,0,0)$
1
1
$(0,0,0,\alpha[i] \ggg r_0)$
$(0,0,0,(\alpha[i] \lll 1) \ggg r_0)$
1
2^{-1}
1
1
2^{-1}
$(0,0,\alpha[i] \ggg r_0,0)$
$(0,0,(\alpha[i] \ggg r_0) \lll 1,0)$
$(0,0,(\alpha[i] \lll 1) \ggg r_0,0)$
$(0,0,((\alpha[i] \lll 1) \ggg r_0) \lll 1,0)$
1
1
1
1
$(0,\alpha[i] \ggg (r_0+r_1),0,0)$
$(0,((\alpha[i] \ggg r_0) \lll 1) \ggg r_1,0,0)$
$(0,\alpha[i] \ggg r_1,0,0)$
$(0,(\alpha[i] \lll 1) \ggg (r_0+r_1),0,0)$
$(0,(((\alpha[i] \lll 1) \ggg r_0) \lll 1) \ggg r_1,0,0)$
$\mathsf{G}_1(0,0,\alpha[i],0)$
1
2^{-1}
$(0,0,\alpha[i],0)$
$(0,0,\alpha[i] \lll 1,0)$
1
1
$(0,\alpha[i] \ggg r_1,0,0)$
$(0,(\alpha[i] \lll 1) \ggg r_1,0,0)$
$\mathsf{G}_1(0,0,0,\alpha[i])$
1
$(0,0,0,\alpha[i] \ggg r_0)$
1
2^{-1}
$(0,0,\alpha[i] \ggg r_0,0)$
$(0,0,(\alpha[i] \lll 1) \ggg r_0,0)$
1
1
$(0,\alpha[i] \ggg (r_0+r_1),0,0)$
$(0,(\alpha[i] \lll 1) \ggg (r_0+r_1),0,0)$

Figure B.1: Relations of the G_1 output differences.

```
????  ????  ????  ??1?  ????  0???  ????  ????  ????  ??10  0???  ??1?  ????  1???  ?00?  ????
??10  0000  0???  ?10?  ????  ????  ??10  0000  0000  0000  0000  0000  0000  0???  ??10  00??
??10  000?  ??10  0000  ????  ?100  ????  ?100  0000  0000  00??  ??10  0000  0000  0000  00??
0000  0000  00??  ?10?  ????  ????  ??10  0000  0000  ?100  0???  ?00?  ????  ?1??  ??1?  ???0
????  ????  ????  ?10?  ????  ????  ????  ????  0???  ????  ????  ????  ????  ????  ????  ????
????  ??10  0???  100?  ????  ????  ?10?  ????  ????  ????  ????  ??1?  ????  1???  ????  ????
????  ????  ?000  0000  ????  ????  ????  ??10  0000  ????  ?0??  ????  ?00?  ????  ????  10??
????  ????  ?1??  000?  ????  1???  ????  ????  ????  ????  ????  ??0?  ????  0???  ????  ????
????  ????  ????  ??00  ????  ????  ????  ????  ?1??  ????  ????  ????  ????  ????  ????  ????
????  ???1  0000  0000  0000  0000  0000  ????  ????  ????  ????  ???1  ????  ?1??  ????  ????
????  ????  ??10  0000  0???  ????  ???1  0??1  0000  0???  ??0?  ????  ??00  ????  ????  ?100
????  ????  ??1?  ?100  ????  ?0??  ????  ????  ????  ????  ??1?  ???0  0???  ?1??  ????  ????
????  ????  ????  ??10  0???  ??0?  ????  ????  ?00?  ????  ????  ????  ????  ??0?  ????  0???
??10  0000  0000  0000  0000  0000  0000  0???  ??10  00??  ???1  0000  0???  ?10?  ????  ????
????  ?100  0000  0000  00??  ???1  0000  00?1  0000  00??  ??00  000?  ??00  0000  ????  ?100
??10  00??  ??10  ?100  0???  ?00?  ????  ?1??  ??1?  ????  ??10  ???0  00??  ?10?  ????  ????
```

vs.

```
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  1000  0000
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  1000  0000
????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????  ????
```

Figure B.2: Bit representation of a 3.5-round impossible differential for 64-bit F.