NORX

v2.0

Designers & Submitters:

Jean-Philippe Aumasson

Philipp Jovanovic

Samuel Neves

https://norx.io

contact@norx.io

August 29, 2015

Contents

1 Introduction

The *Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR)* [3] invites cryptographers to submit authenticated encryption schemes supporting associated data (AEAD) [45], that offer advantages over AES-GCM [30, 42] 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 [18, 21], 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 [15], 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 $\texttt{ab} = 171$. 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 summarises basic notation used throughout the document.

Table 1.1: Notation used throughout the document

Symbol	Meaning
ε	The empty bitstring of length 0.
0^n	The all-zero bitstring of length n.
$\lvert x \rvert$	Length of bitstring x in bits.
$\lvert x \rvert_n$	Length of bitstring x in n-bit blocks.
$x \parallel y$	Concatenation of bitstrings x and y.
$\mathsf{hw}(x)$	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.
$\mathsf{left}_l(x)$	Truncation of bitstring x to its l left-most bits.
$\mathsf{right}_r(x)$	Truncation of bitstring x to its r right-most bits.

[1] The name stems from "NO(T A)RX" and is pronounced like "norcks".
[2] Derived from the well-known identity $a + b = (a \oplus b) + (a \wedge b) \ll 1$ [12, 40].

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 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 *round number* $1 \leq l \leq 63$,
- a *parallelism degree* $0 \leq p \leq 255$,
- a *tag size* of $t \leq 4w$ bits.

Encryption Mode

NORX encryption takes as input

- a *key* K of $k = 4w$ bits,
- a *nonce* N of $n = 2w$ bits,
- a *datagram* (A, M, Z) where
  - A is a *header*,
  - M is a *message*,
  - Z is a *trailer/footer*,

  and where any of A, M, Z can be the empty string (that is, of length 0).

NORX encryption produces as output

- a *ciphertext* (or *encrypted payload*) C of the same size as M,
- an *authentication tag* T of t bits.

In summary, NORX encryption $\mathcal{E}$ is specified as

$$\mathcal{E} : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^t$$

with

$$\mathcal{E}(K, N, A, M, Z) = (C, T)$$

where $|M| = |C|$.

Decryption Mode

NORX decryption takes as input

- a key K of $k = 4w$ bits,
- a nonce N of $n = 2w$ bits,
- a datagram (A, C, Z) where,
  - A is a header,
  - C is a ciphertext,
  - Z is a trailer,

  and where any of A, M, Z can be the empty string (that is, of length 0).
- an authentication tag T of t bits.

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

In summary, NORX decryption $\mathcal{D}$ is specified as

$$\mathcal{D} : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^t \rightarrow \{0,1\}^* \cup \{\perp\}$$

with

$$\mathcal{D}(K, N, A, C, Z, T) = \begin{cases} M & \text{if } T = T' \\ \perp & \text{if } T \neq T' \end{cases}$$

where T denotes the received authentication tag, T' the one computed on the recipient's side and $|M| = |C|$.

2.2 Instances

A NORX instance is a choice of values for the four parameters w, l, p, and t. Table 2.1 proposes five NORX instances for different use cases: 128- or 256-bit security, four or six rounds, and a version with four-wise parallelism. Table 2.1 also shows he corresponding nonce and key sizes n and k. The priority order of the recommended parameter sets from highest at the top to lowest at the bottom.

Table 2.1: NORX instances

Nr.	w	l	p	t	k	n
1.	64	4	1	256	256	128
2.	32	4	1	128	128	64
3.	64	6	1	256	256	128
4.	32	6	1	128	128	64
5.	64	4	4	256	256	128

We set the *default tag size* t for a given word size w to $t = 4w$, i.e. for $w = 32$ we get $t = 128$ and for $w = 64$ we get $t = 256$. A detailed discussion on the parameter combinations can be found in §4.2.

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

2.3 Layout Overview

NORX uses the monkeyDuplex construction [18, 21] enhanced with the capability to process payload in parallel. The number i of parallel encryption lanes L_i is controlled by the parameter $0 \leq p \leq 255$. For the value $p = 1$, the layout of NORX corresponds to a standard (sequential) duplex construction, see Fig. 2.1. For $p > 1$, the number of lanes L_i is bounded by the latter value, e.g. for $p = 2$ see Fig. 2.2. If $p = 0$, the number of lanes L_i is bounded by the size of the payload. In that case, the layout of NORX is similar to that of the PPAE construction [25].

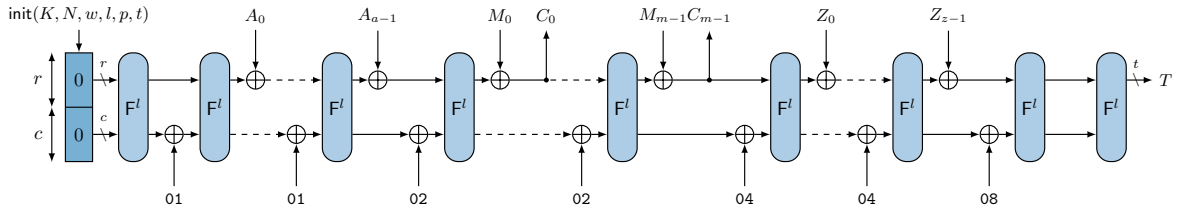


Figure 2.1: Layout of NORX for $p = 1$

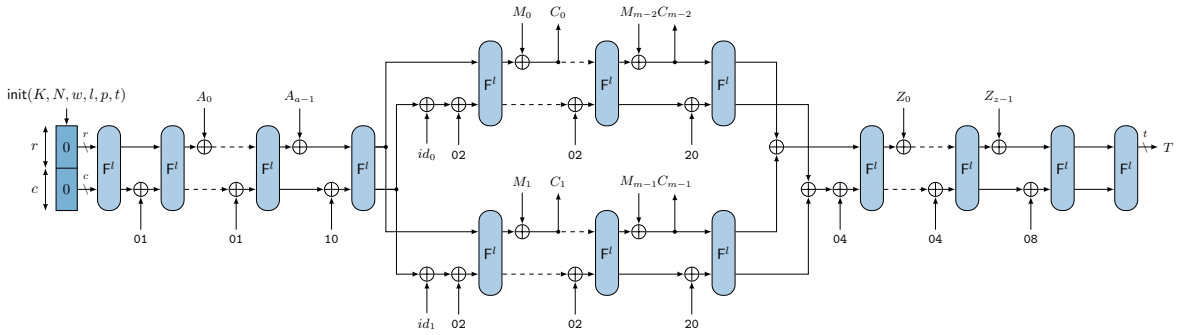


Figure 2.2: Layout of NORX for $p = 2$

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^l denotes its l-fold iteration. The organisation of the internal state S of NORX is as follows:

w	b	r	c
32	512	384	128
64	1024	768	256

The state is viewed as a concatenation of 16 words, i.e. $S = s_0 \parallel \cdots \parallel s_{15}$, where $s_0, \ldots, s_{11}$ are called the *rate words* (where data blocks are injected) $s_{12}, \ldots, s_{15}$ are called the *capacity words* (which remain untouched). Conceptually, the 16 state words are arranged in a 4×4 matrix:

$$S = \left(\begin{array}{cccc} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ \hline s_{12} & s_{13} & s_{14} & s_{15} \end{array}\right)$$

2.4 The Permutation F^l

The complete pseudocode for the NORX core permutation F^l is given in Fig. 2.4. A single NORX round F processes the 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 [11], and will be denoted by col and diag, respectively. An illustration of these operations is shown in Fig. 2.3.

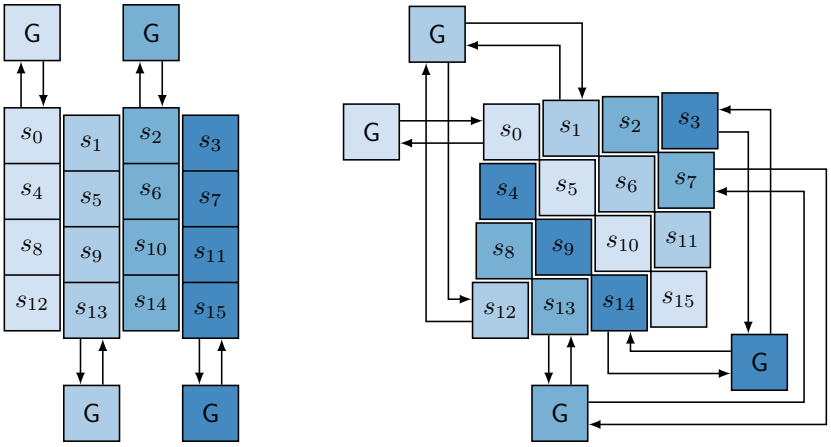


Figure 2.3: Column step and diagonal step of F

The G function uses *cyclic rotations* $\ggg$ and a *non-linear operation* H interchangeably to update its four input words a, b, c and d. The rotation offsets r_0, r_1, r_2, and r_3 for the cyclic rotations of 32- and 64-bit NORX are specified in Table 2.2.

2.5 The NORX Mode

The NORX mode is divided into a high-level and a low-level API discussed in §§2.5.1 and 2.5.2, respectively. The *high-level interface* consists of only two functions: AEADEnc and AEADDec.

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

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

Algorithm: $\mathsf{F}^l(S)$

1. **for** $i \in \{0, \ldots, l-1\}$ **do**
2. $\quad S \leftarrow \mathsf{diag}(\mathsf{col}(S))$
3. **end**
4. **return** S

Algorithm: $\mathsf{G}(a, b, c, d)$

1. $a \leftarrow \mathsf{H}(a, b)$
2. $d \leftarrow (a \oplus d) \ggg r_0$
3. $c \leftarrow \mathsf{H}(c, d)$
4. $b \leftarrow (b \oplus c) \ggg r_1$
5. $a \leftarrow \mathsf{H}(a, b)$
6. $d \leftarrow (a \oplus d) \ggg r_2$
7. $c \leftarrow \mathsf{H}(c, d)$
8. $b \leftarrow (b \oplus c) \ggg r_3$
9. **return** a, b, c, d

Algorithm: $\mathsf{col}(S)$

1. $s_0, s_4, s_8, s_{12} \leftarrow \mathsf{G}(s_0, s_4, s_8, s_{12})$
2. $s_1, s_5, s_9, s_{13} \leftarrow \mathsf{G}(s_1, s_5, s_9, s_{13})$
3. $s_2, s_6, s_{10}, s_{14} \leftarrow \mathsf{G}(s_2, s_6, s_{10}, s_{14})$
4. $s_3, s_7, s_{11}, s_{15} \leftarrow \mathsf{G}(s_3, s_7, s_{11}, s_{15})$
5. **return** S

Algorithm: $\mathsf{diag}(S)$

1. $s_0, s_5, s_{10}, s_{15} \leftarrow \mathsf{G}(s_0, s_5, s_{10}, s_{15})$
2. $s_1, s_6, s_{11}, s_{12} \leftarrow \mathsf{G}(s_1, s_6, s_{11}, s_{12})$
3. $s_2, s_7, s_8, s_{13} \leftarrow \mathsf{G}(s_2, s_7, s_8, s_{13})$
4. $s_3, s_4, s_9, s_{14} \leftarrow \mathsf{G}(s_3, s_4, s_9, s_{14})$
5. **return** S

Algorithm: $\mathsf{H}(x, y)$

1. **return** $(x \oplus y) \oplus ((x \wedge y) \ll 1)$

Figure 2.4: The NORX permutation F^l

These provide functionality for encryption and authentication of a message on the one hand and decryption and verification of an encrypted payload on the other. Both functions of course also support processing of associated data. The *low-level interface* defines the concrete implementation of padding, domain separation, absorption or encryption of data block sequences, tag generation, etc.

2.5.1 High-level Structure

The two high-level interface functions AEADEnc and AEADDec are depicted in Fig. 2.5.

2.5.2 Low-level Structure

The low-level functions of NORX are depicted in Fig. 2.6. Before going into the details of those methods, we first introduce the mechanisms for padding and domain separation which are required later on.

Padding

NORX adopts the so-called *multi-rate padding* [21]. This padding rule is defined by the map

$$\mathsf{pad}_r : X \longmapsto X \parallel 10^u 1$$

Algorithm: AEADEnc(K, N, A, M, Z)

1. $S \leftarrow \mathsf{initialise}(K, N)$
2. $S \leftarrow \mathsf{absorb}(S, A, 01)$
3. $\overline{S} \leftarrow \mathsf{branch}(S, |M|, 10)$
4. $\overline{S}, C \leftarrow \mathsf{encrypt}(\overline{S}, M, 02)$
5. $S \leftarrow \mathsf{merge}(\overline{S}, |M|, 20)$
6. $S \leftarrow \mathsf{absorb}(S, Z, 04)$
7. $S, T \leftarrow \mathsf{finalise}(S, 08)$
8. **return** C, T

Algorithm: AEADDec(K, N, A, C, Z, T)

1. $S \leftarrow \mathsf{initialise}(K, N)$
2. $S \leftarrow \mathsf{absorb}(S, A, 01)$
3. $\overline{S} \leftarrow \mathsf{branch}(S, |C|, 10)$
4. $\overline{S}, M \leftarrow \mathsf{decrypt}(\overline{S}, C, 02)$
5. $S \leftarrow \mathsf{merge}(\overline{S}, |C|, 20)$
6. $S \leftarrow \mathsf{absorb}(S, Z, 04)$
7. $S, T' \leftarrow \mathsf{finalise}(S, 08)$
8. **if** $T = T'$ **then return** M **else return** $\perp$ **end**

Figure 2.5: High-level interface functions of the NORX mode

where X is a bitstring and $u = (-|X| - 2) \bmod r$. If r and $|X|$ are divisible by 8 and X is viewed as a sequence of bytes, then the multi-rate padding can be written as

$$\mathsf{pad}_r : X \longmapsto X \parallel 01 \parallel 00^u \parallel 80$$

where $u = (-|X|_8 - 2) \bmod (r/8)$.

Domain Separation

NORX has a very simple and lightweight domain separation mechanism: it is performed by XORing a *domain separation constant* to the least significant byte of s_{15} each time before the state s is transformed by the permutation F^l. Distinct constants are used for the different algorithm phases, i.e. for the three different message processing stages, for tag generation, and in case of $p \neq 1$, for branching and merging steps. Table 2.3 gives the specification of those constants and Figs. 2.1 and 2.2 illustrate their integration into the state of NORX. Figs. 2.5 and 2.6 show their concrete usage.

Table 2.3: Domain separation constants

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

Initialisation

The method initialise sets up the $16w$-bit internal state $S = (s_0, \ldots, s_{15})$ of NORX by processing a $4w$-bit key $K = k_0 \parallel k_1 \parallel k_2 \parallel k_3$, a $2w$-bit nonce $N = n_0 \parallel n_1$, the instance parameters w, l, p, and t and some initialisation constants. These constants are given in Table 2.4 and can be derived by

$$(u_0, \ldots, u_{15}) = \mathsf{F}^2(0, \ldots, 15)$$

which allows on-the-fly computation if necessary. Note, however, that only $u_2, u_3, u_8, \ldots, u_{15}$ are actually used in initialise.

Algorithm: initialise(K, N)

$n_0 \parallel n_1 \leftarrow N$, s.t. $|n_i| = w$
$k_0 \parallel k_1 \parallel k_2 \parallel k_3 \leftarrow K$, s.t. $|k_i| = w$
$S \leftarrow (n_0, n_1, u_2, u_3, k_0, k_1, k_2, k_3, u_8, u_9, u_{10}, u_{11}, u_{12}, u_{13}, u_{14}, u_{15})$
$s_{12} \leftarrow s_{12} \oplus w$
$s_{13} \leftarrow s_{13} \oplus l$
$s_{14} \leftarrow s_{14} \oplus p$
$s_{15} \leftarrow s_{15} \oplus t$
$S \leftarrow \mathsf{F}^l(S)$
return S

Algorithm: encrypt$(\overline{S}, M, v)$

$C \leftarrow \varepsilon$
$M_0 \parallel \cdots \parallel M_{m-1} \leftarrow M$, s.t. $|M_i| = r, 0 \leq |M_{m-1}| < r$
if $|M| > 0$ **then**
  for $i \in \{0, \ldots, m-2\}$ **do**
    $j \leftarrow i \bmod |\overline{S}|_b$
    $\overline{s}_{j,15} \leftarrow \overline{s}_{j,15} \oplus v$
    $\overline{S}_j \leftarrow \mathsf{F}^l(\overline{S}_j)$
    $C_i \leftarrow \mathsf{left}_r(\overline{S}_j) \oplus M_i$
    $\overline{S}_j \leftarrow C_i \parallel \mathsf{right}_c(\overline{S}_j)$
  end
  $j \leftarrow (m-1) \bmod |\overline{S}|_b$
  $\overline{s}_{j,15} \leftarrow \overline{s}_{j,15} \oplus v$
  $\overline{S}_j \leftarrow \mathsf{F}^l(\overline{S}_j)$
  $C_{m-1} \leftarrow \mathsf{left}_{|M_{m-1}|}(\overline{S}_j) \oplus M_{m-1}$
  $\overline{S}_j \leftarrow \overline{S}_j \oplus (\mathsf{pad}_r(M_{m-1}) \parallel 0^c)$
  $C \leftarrow C_0 \parallel \cdots \parallel C_{m-1}$
end
return $\overline{S}, C$

Algorithm: decrypt$(\overline{S}, C, v)$

$M \leftarrow \varepsilon$
$C_0 \parallel \cdots \parallel C_{m-1} \leftarrow C$ s.t. $|C_i| = r, 0 \leq |C_{m-1}| < r$
if $|C| > 0$ **then**
  for $i \in \{0, \ldots, m-2\}$ **do**
    $j \leftarrow i \bmod |\overline{S}|_b$
    $\overline{s}_{j,15} \leftarrow \overline{s}_{j,15} \oplus v$
    $\overline{S}_j \leftarrow \mathsf{F}^l(\overline{S}_j)$
    $M_i \leftarrow \mathsf{left}_r(\overline{S}_j) \oplus C_i$
    $\overline{S}_j \leftarrow C_i \parallel \mathsf{right}_c(\overline{S}_j)$
  end
  $j \leftarrow (m-1) \bmod |\overline{S}|_b$
  $\overline{s}_{j,15} \leftarrow \overline{s}_{j,15} \oplus v$
  $\overline{S}_j \leftarrow \mathsf{F}^l(\overline{S}_j)$
  $M_{m-1} \leftarrow \mathsf{left}_{|C_{m-1}|}(\overline{S}_j) \oplus C_{m-1}$
  $\overline{S}_j \leftarrow \overline{S}_j \oplus (\mathsf{pad}_r(M_{m-1}) \parallel 0^c)$
  $M \leftarrow M_0 \parallel \cdots \parallel M_{m-1}$
end
return $\overline{S}, M$

Algorithm: absorb(S, X, v)

$X_0 \parallel \cdots \parallel X_{m-1} \leftarrow X$, s.t. $|X_i| = r, 0 \leq |X_{m-1}| < r$
if $|X| > 0$ **then**
  for $i \in \{0, \ldots, m-2\}$ **do**
    $s_{15} \leftarrow s_{15} \oplus v$
    $S \leftarrow \mathsf{F}^l(S)$
    $S \leftarrow S \oplus (X_i \parallel 0^c)$
  end
  $s_{15} \leftarrow s_{15} \oplus v$
  $S \leftarrow \mathsf{F}^l(S)$
  $S \leftarrow S \oplus (\mathsf{pad}_r(X_{m-1}) \parallel 0^c)$
end
return S

Algorithm: branch(S, m, v)

$\overline{S} \leftarrow 0^b$
if $p \neq 1$ and $m > 0$ **then**
  $s \leftarrow p$
  if $p = 0$ **then**
    $s \leftarrow \lceil m/r \rceil$
  end
  $\overline{S} = (\overline{S}_0, \ldots, \overline{S}_{s-1}) \leftarrow (0^b, \ldots, 0^b)$
  $s_{15} \leftarrow s_{15} \oplus v$
  $S \leftarrow \mathsf{F}^l(S)$
  for $i \in \{0, \ldots, s-1\}$ **do**
    $\overline{S}_i \leftarrow S$
    $\overline{s}_{i,13} \leftarrow \overline{s}_{i,13} \oplus i \bmod 2^w$
    $\overline{s}_{i,14} \leftarrow \overline{s}_{i,14} \oplus \lfloor i/2^w \rfloor)$
  end
else
  $\overline{S} \leftarrow S$
end
return $\overline{S}$

Algorithm: merge$(\overline{S}, m, v)$

$S \leftarrow 0^b$
if $p \neq 1$ and $m > 0$ **then**
  for $i \in \{0, \ldots, |\overline{S}|_b - 1\}$ **do**
    $\overline{s}_{i,15} \leftarrow \overline{s}_{i,15} \oplus v$
    $\overline{S}_i \leftarrow \mathsf{F}^l(\overline{S}_i)$
    $S \leftarrow S \oplus \overline{S}_i$
  end
else
  $S \leftarrow \overline{S}$
end
return S

Algorithm: finalise(S, v)

$s_{15} \leftarrow s_{15} \oplus v$
$S \leftarrow \mathsf{F}^l(\mathsf{F}^l(S))$
$T \leftarrow \mathsf{left}_t(S)$
return S, T

Figure 2.6: Low-level interface functions of the NORX mode

Table 2.4: Initialisation constants

w	32	64	w	32	64
u_0	0454EDAB	E4D324772B91DF79	u_8	A3D8D930	B15E641748DE5E6B
u_1	AC6851CC	3AEC9ABAAEB02CCB	u_9	3FA8B72C	AA95E955E10F8410
u_2	B707322F	9DFBA13DB4289311	u_{10}	ED84EB49	28D1034441A9DD40
u_3	A0C7C90D	EF9EB4BF5A97F2C8	u_{11}	EDCA4787	7F31BBF964E93BF5
u_4	99AB09AC	3F466E92C1532034	u_{12}	335463EB	B5E9E22493DFFB96
u_5	A643466D	E6E986626CC405C1	u_{13}	F994220B	B980C852479FAFBD
u_6	21C22362	ACE40F3B549184E1	u_{14}	BE0BF5C9	DA24516BF55EAFD4
u_7	1230C950	D9CFD35762614477	u_{15}	D7C49104	86026AE8536F1501

Data Absorption

The method absorb takes an arbitrary long bitstring X as input and absorbs it in blocks of r bits into the internal state thereby ensuring authenticity of X. If the last block is smaller then r bits, it is extended to the block size through pad_r. For domain separation the constant v is used. Data absorption is skipped entirely in case the input has length 0, i.e. if X corresponds to the empty bitstring ε.

In NORX the function absorb is used for authenticating associated data in the form of header data A using domain separation constant $v = 01$ and/or trailer data Z using domain separation constant $v = 04$. Refer to the high-level interface in Fig. 2.5 to see where and how absorb is used concretely in NORX.

Branching

If the parallelism degree $p \neq 1$ then branch is used to prepare parallel payload processing. branch is skipped entirely if either $p = 1$ or $|P| = 0$. The state S is extended to a multi-state vector $\overline{S}$ having either p elements if $p > 1$ or $\lceil |M|_r \rceil$ elements if $p = 0$. Note that in order to ensure that each lane produces a unique bitstream for encryption, a *lane number* i is integrated into state copy $\overline{S}_i$ (included into words $\overline{s}_{i,13}$ and $\overline{s}_{i,14}$) together with the domain separation constant $v = 10$.

Data Encryption and Decryption

The method encrypt (decrypt) takes an arbitrary long bitstring P (C) as input and encrypts (decrypts) it thereby producing the encrypted (decrypted) payload C (P). Since P is also absorbed into the state S, its authenticity is ensured as well. As in absorb, data is processed in r-bit blocks and the last block is padded using pad_r. Note that in the latter case only a truncated data block of the same size as the unpadded input block is extracted such that $|P| = |C|$ holds. The constant $v = 02$ is used for domain separation.

The different cases for p are handled as follows. For $p = 1$ the NORX mode corresponds to a regular sequential sponge construction and no special steps have to be taken for data encryption or decryption. For $p > 1$ a fixed number of p parallel lanes is available for data processing. Data blocks are rotated in a round-robin fashion across the states by assigning the i-th data block to state $i \bmod p$. In the last case, if $p = 0$, each data block is processed on its own separate lane.

Merging

The merge function is only executed if $p \neq 1$ and $|M| > 0$. After parallel-processing all payload data blocks, the states $\overline{S}_i$ are merged back into a single state S. The domain separation constant for merge is $v = 20$.

Finalisation

The finalise function generates an authentication tag T by first injecting the domain separation constant $v = 08$ then transforming S *twice* with the permutation F^l and finally extracting the t leftmost bits from $s_0 \parallel \cdots \parallel s_{11}$ which are returned as the tag T.

Tag Verification

Note that tag verification is not listed explicitly among the low-level interface functions in Fig. 2.6 but rather in Fig. 2.5, see the last step of AEADDec.

Tag verification consists of comparing the *received tag* T to the *generated tag* T'. If $T = T'$, tag verification succeeds; otherwise tag verification fails, the decrypted payload is discarded and an error $\perp$ 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.

2.6 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.

2.6.1 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, A, C, Z, and T, as well as information to parse those elements.

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

We encode the byte length of A and Z 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 2.5 and 2.6 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.

2.6.2 Variable Parameters

With *variable parameters*, the datagram needs to signal the values of w, l, and p. The header is thus extended to encode those values, as specified in Tables 2.7 and 2.8. To minimize bandwidth, w is encoded on one bit, supporting the two choices 32-bit ($w = 0$) and 64-bit ($w = 1$), l on 7 bits (with the MSB fixed at 0, i.e. supporting up to 63 rounds), and p on 8 bits (supporting parallelization degree up to 255). The datagram header is thus only 2 bytes longer than the header for fixed parameters.

[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.

Table 2.5: NORX32 datagram for fixed parameters (offsets are in bytes)

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

Table 2.6: NORX64 datagram for fixed parameters (offsets are in bytes)

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

Table 2.7: NORX32 datagram for variable parameters (offsets are in bytes)

Offset	**0**	**1**	**2**	**3**
0 **4**	Nonce N			
8	Header byte length $\|A\|$		Trailer byte length $\|Z\|$	
12	Encrypted payload byte length $\|C\|$			
16	$w(1)\|\|l(7)$	p		
20 ... **??**	Header A			
?? ... **??**	Encrypted payload C			
?? ... **??**	Trailer Z			
?? ... **??**	Tag T			

Table 2.8: NORX64 datagram for variable parameters (offsets are in bytes)

Offset	**0**	**1**	**2**	**3**
0 **4** **8** **12**	Nonce N			
16	Header byte length $\|A\|$		Trailer byte length $\|Z\|$	
20 **24**	Encrypted payload byte length $\|C\|$			
28	$w(1)\|\|l(7)$	p		
32 ... **??**	Header A			
?? ... **??**	Encrypted payload C			
?? ... **??**	Trailer Z			
?? ... **??**	Tag T			

3 Security Goals

We expect NORX with $l \geq 4$ rounds 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.

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

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

Note that NORX restricts the number of messages processed with a given key: in [17] 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.

3.0.2.1 Security Bounds for the Mode of Operation

Let $\Pi = (\mathcal{E}, \mathcal{D})$ denote NORX, with encryption function $\mathcal{E}$, decryption function $\mathcal{D}$, and based on an ideal underlying permutation p. Then the following privacy and authenticity security

bounds are satisfied

$$\mathbf{Adv}^{\mathsf{priv}}_{\Pi}(q_p, q_{\mathcal{E}}, \lambda_{\mathcal{E}}) \leq \frac{3(q_p + \sigma_{\mathcal{E}})^2}{2^{b+1}} + \left(\frac{8eq_p\sigma_{\mathcal{E}}}{2^b}\right)^{1/2} + \frac{rq_p}{2^c} + \frac{q_p + \sigma_{\mathcal{E}}}{2^k}$$

$$\begin{aligned}\mathbf{Adv}^{\mathsf{auth}}_{\Pi}(q_p, q_{\mathcal{E}}, \lambda_{\mathcal{E}}, q_{\mathcal{D}}, \lambda_{\mathcal{D}}) \leq{}& \frac{(q_p + \sigma_{\mathcal{E}} + \sigma_{\mathcal{D}})^2}{2^b} + \left(\frac{8eq_p\sigma_{\mathcal{E}}}{2^b}\right)^{1/2} + \frac{rq_p}{2^c} \\ &+ \frac{q_p + \sigma_{\mathcal{E}} + \sigma_{\mathcal{D}}}{2^k} + \frac{(q_p + \sigma_{\mathcal{E}} + \sigma_{\mathcal{D}})\sigma_{\mathcal{D}}}{2^c} + \frac{q_{\mathcal{D}}}{2^t}\end{aligned}$$

where r, c, b, k and t are rate, capacity, state, key and tag sizes, e is Euler's number, q_p are the number of permutation queries, $q_{\mathcal{E}}$ are the number of encryption queries of total length $\lambda_{\mathcal{E}}$ and $\sigma_{\mathcal{E}}$ is specified as follows:

$$\sigma_{\mathcal{E}} := \sum_{j=1}^{q_{\mathcal{E}}} \sigma_{\mathcal{E},j} \leq \begin{cases} 2\lambda_{\mathcal{E}} + 4q_{\mathcal{E}}, & \text{if } p = 0 \\ \lambda_{\mathcal{E}} + 3q_{\mathcal{E}}, & \text{if } p = 1 \\ \lambda_{\mathcal{E}} + (p+4)q_{\mathcal{E}}, & \text{if } p > 1 \end{cases}$$

The values $q_{\mathcal{D}}$, $\lambda_{\mathcal{D}}$ and $\sigma_{\mathcal{D}}$ for decryption $\mathcal{D}$ are specified analogously.

In summary, the NORX mode of operation achieves security levels of $\min\{2^{b/2}, 2^c, 2^k\}$ assuming an ideal underlying permutation p and, intuitively spoken, offers authenticity as long as it offers privacy. For more information see [36].

4 Features

NORX was designed for users, provides several features desirable for practical applications and offers a couple of advantages over AES-GCM [42]. 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 (i.e. 16 steps consisting of 8 column and 8 diagonal steps interleaved with each other) and of 4 rounds (i.e. 8 steps consisting of 4 column and 4 diagonal steps interleaved with each other) for the data processing part ensure 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 uses a default tag size 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 Chapter 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 [35].
- **Autonomy.** NORX requires no external primitive.
- **Diversity.** The cipher does not depend on AES instructions, thereby adding to the diversity among cryptographic algorithms.
- **Extensibility.** Thanks to the duplex construction and a simple, yet powerful domain separation scheme, NORX can be easily extended to support additional features, like secret message numbers, sessions, or forward secrecy without losing its security guarantees.

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

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

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 p. An implementation should rather be optimised for one set of parameters to gain higher efficiency.

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^l, 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^l.

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

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

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.

[2]Note that the 10 constants can be generated on-the-fly from $0, \ldots, 15$, see §2.5.2.

Parallelism

The core permutation F of NORX has a natural parallelism of 4 independent G applications. Additionally, NORX allows for greater parallelism levels using multiple lanes. Using the $p = 0$ mode, see Line 4, the internal parallelism level of NORX is effectively unbounded for long enough messages.

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 [22]. 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 [7].

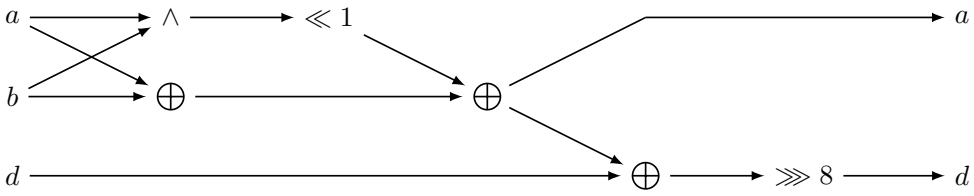
Avoiding Latency

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

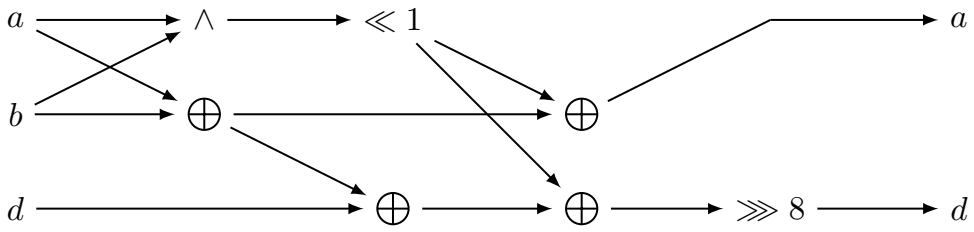
reduced latency:

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

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. Fig. 4.1). In a sufficiently parallel architecture, this can save at least $4 \times 2 \times l$ cycles, which translates to $6.4l/w$ 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 $p = 1$, 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 $p = 1$), 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 1.75 GiBps (1.99 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 Fig. 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.

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 $w \in \{32, 64\}$, $l \in \{2, \ldots, 16\}$ and $p = 1$. 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 [34].

Table 4.2: Software performance of NORX in cycles per byte

data length [byte]		long	4096	1536	576	64	8
Samsung Exynos 4412 Prime (Cortex-A9) at 1.7 GHz							
NORX32-4-1	Ref	16.72	18.03	20.52	27.92	109.48	771.88
	NEON	9.27	10.20	11.95	16.46	72.30	521.00
NORX64-4-1	Ref	15.60	17.91	22.02	32.42	148.55	1177.12
	NEON	7.13	8.40	10.61	16.25	82.12	648.88
BeagleBone Black Rev B (Cortex-A8) at 1.0 GHz							
NORX32-4-1	Ref	16.66	17.90	20.28	26.49	102.34	708.00
	NEON	9.49	10.52	12.36	17.92	75.62	550.12
NORX64-4-1	Ref	17.24	19.81	24.34	35.73	164.86	1317.50
	NEON	7.00	8.35	10.67	16.44	85.66	680.00
Intel Core i7-2630QM at 2.0 GHz							
NORX64-6-1	Ref	6.33	7.02	8.24	13.96	70.62	607.50
	AVX	4.02	4.42	5.14	6.90	63.75	204.00
NORX64-4-1	Ref	4.83	5,35	6.30	8.66	50.00	400.62
	AVX	2.68	2.96	3.45	4.66	17.18	137.5
Intel Core i7-3667U at 2.0 GHz							
NORX64-6-1	Ref	8.15	9.01	10.49	14.15	53.20	425.62
	AVX	5.04	5.56	6.45	8.65	32.19	255.00
NORX64-4-1	Ref	5.58	6,17	7.22	9.82	38.05	303.75
	AVX	3.37	3.72	4.35	5.84	22.11	174.38
Intel Core i7-4770K at 3.5 GHz							
NORX64-6-1	Ref	5.37	5.94	6.92	9.40	36.44	292.00
	AVX2	2.98	3.29	3.84	5.17	19.00	153.00
NORX64-4-1	Ref	3.98	4.39	5.11	6.97	27.19	217.00
	AVX2	1.99	2.20	2.58	3.49	12.94	104.50

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 [18, 21], but enhanced by the capability of parallel payload processing on multiple lanes (cf. Figs. 2.1 and 2.2). The *parallel duplex construction* is similar to the tree-hashing mode for sponge functions [20]. 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 A and Z in parallel, as they are usually rather short. The number of encryption lanes is controlled by the parallelism degree $0 \leq p \leq 255$, which is a fixed instance parameter. Hence two instances with distinct p values cannot decrypt data from each other. Obviously the same holds for differing w and l 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 $p \neq 1$, 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 [15], which itself is an advancement of the quarter-round function of the eSTREAM finalist Salsa20 [1, 16]. Variants of ChaCha's quarter-round function can be found for example in the SHA-3 finalist BLAKE [2, 10] and its successor BLAKE2 [11].

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. Fig. 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]

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

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

In NORX the integer additions is replaced by the following expression

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

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. Moving to simple bitwise logical operations facilitates hardware implementations. One way to realise G as a circuit is depicted in Fig. 5.2.

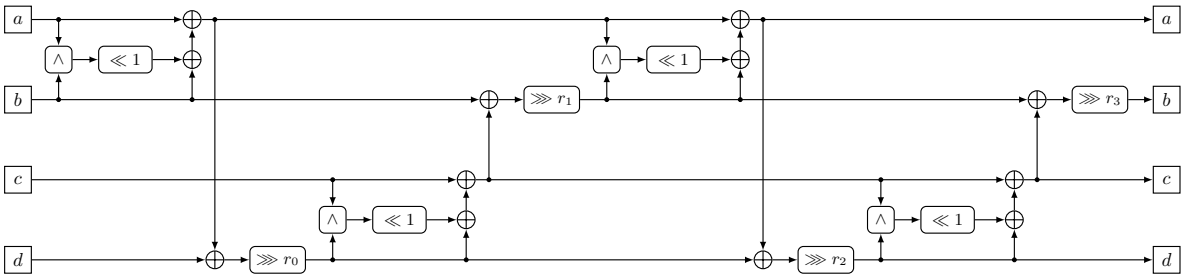


Figure 5.2: The G circuit

Bijectivity

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

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

[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 [11].

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:

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

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 [13]. 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 [5, 29], 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 [15].

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. $\mathsf{F}^l(S) = \mathsf{F}^l(S')$, after any number of rounds l. This characteristic of F is important for the duplex construction [21, 18] 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.

5.4 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 [18] and has only minor effects on the overall performance, but greatly increases the security of NORX. The minimal value of $l = 4$ is based on the following observations:

1. The best attacks on Salsa20 and ChaCha [9, 47, 49] 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 Chapter 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.

5.5 Selection of Constants

5.5.1 Initialisation

The initialisation constants are listed in Table 2.4 and are derived through

$$(u_0, \ldots, u_{15}) = \mathsf{F}^2(0, \ldots, 15)$$

as already mentioned in §2.5.2. This approach allows an on-the-fly computation, if necessary, and is meant to provide transparency in order to show that the values belong to the "nothing-up-my-sleeves" category, i.e. that they were selected in such a way that there is no possibility

[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 Chapter 6 on cryptanalysis for more details.

to hide a backdoor. The main purpose of the initialisation constants is to provide some asymmetry during initialisation and to limit the freedom where differences can be injected by an attacker.

5.5.2 Domain Separation

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 $p \neq 1$. 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, 21]. 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.

5.5.3 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 l denote a round number and let L and L_l 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 $X = \mathsf{F}^l(S) \oplus \mathsf{F}^l(S')$, where F denotes the round function of NORX.
3. Save $\mathsf{hw}(X)$ to L_l.

After the above loop is finished the test computes minimum, maximum, average and median values of the elements of L_l, saves the latter together with the offsets to L and resets L_l. 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 $l = 1$, achieve full diffusion after $l = 2$, and additionally offer good performance.

Table 5.1 lists the results of the test for 32- and 64-bit core functions with $l \leq 4$ 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. Fig. 5.3 shows how single bit changes in the word s_0 propagate through the NORX state over the course of 5 steps ($= \mathrm{F}^{2.5}$). 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.

Table 5.1: Diffusion statistics for NORX and ChaCha round functions

	NORX32				ChaCha (32-bit)			
l	min	max	avg	med	min	max	avg	med
1	83	280	179.22	181	73	294	182.19	185
2	194	307	256.02	256	199	312	255.99	256
3	198	312	255.99	256	204	313	255.98	256
4	201	307	255.99	256	200	314	255.98	256
	NORX64				ChaCha (64-bit)			
l	min	max	avg	med	min	max	avg	med
1	95	429	230.13	222	73	506	248.84	246
2	440	589	511.98	512	430	591	512.01	512
3	434	589	512.00	512	439	589	511.97	512
4	428	589	511.98	512	435	585	512.00	512

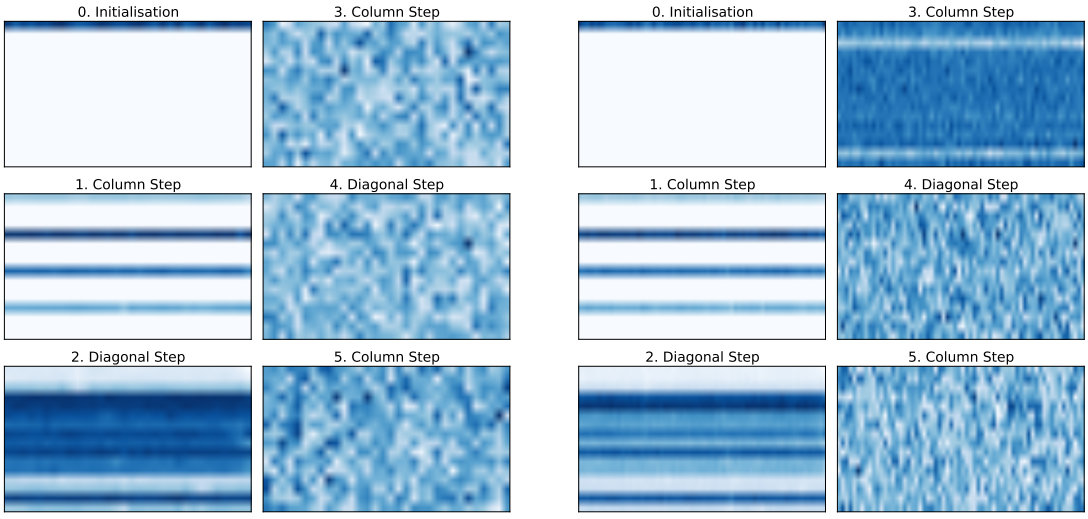


Figure 5.3: Visualisation of NORX diffusion

5.6 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 [20] 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.

5.7 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^l 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 $\alpha^0 := (\alpha_0^0, \ldots, \alpha_{m-1}^0), \ldots, \alpha^l := (\alpha_0^l, \ldots, \alpha_{m-1}^l)$ 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 $\mathrm{hw}(\alpha) = 1$ 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^l. 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 Fig. B.1 in the appendix for a visualisation of the relations between input and output differences of G_1.

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

	$\beta_0[j]$	$\beta_1[j]$	$\beta_2[j]$	$\beta_3[j]$	$\Pr(\alpha_v[i] \xrightarrow{G_1} \beta_w[j])$
$\alpha_0[i]$	$\alpha_0[i]$	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 6.2: Maximum Hamming weight of an output difference γ_w after one application of G on an input difference $\alpha_v[i]$

	$a_0[i]$	$a_1[i]$	$a_2[i]$	$a_3[i]$
max. $\mathsf{hw}(\gamma_w)$	102	115	34	39

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^l. 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:

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

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. $F^{1.5}$), 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 [39]. Later it was extended by Biham et al. in order to attack the block ciphers Skipjack [23] and IDEA [24]. 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 Fig. 6.1. A $*$ denotes a partially known and a $?$ an unknown entry. Our analysis

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

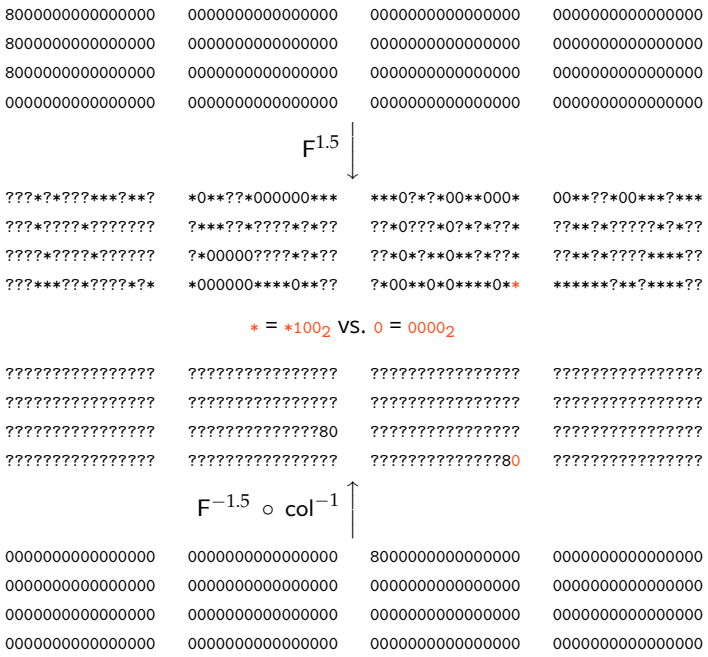


Figure 6.1: An impossible differential over 3.5 rounds of 64-bit F

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 initialisein Fig. 2.6). 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* [46] and that F^l provides maximum security for $l \geq 4$, 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 [6, 8, 28, 31] 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.

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

	#polynomials by degree						#monomials			
	3	4	5	6	7	8	min	max	avg	med
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

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 $\mathsf{F}^l(0) = 0$. 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 [48] on a workstation with 64 GiB RAM.

Furthermore, we used the constraint solver STP [32] 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. [26, 27] to cryptanalyse block ciphers. Later they were also extended to stream ciphers [44] and hash functions [33]. To protect sponge constructions against slide attacks two simple countermeasures can be found in the literature:

1. In [33] it is proposed to add a non-zero constant to the state just before applying the permutation.
2. In [43] 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 are added to the state just before the permutation F^l is applied and the multi-rate padding 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 [37] 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 [38] and Keccak [41].

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 [14].
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 Changelog

Changes from v1.1 to v2.0:

- Complete re-write of the spec aiming at more clarity and consistency.
- Renaming of variables:

type	old	new	type	old	new
word size	W	w	header	H	A
round number	R	l	payload	P	M
parallelism degree	D	p	trailer	T	Z
tag size	$\|A\|$	t	tag	A	T

- New derivation scheme for initialisation constants, see §2.5.2.
- New arrangement of the elements in the initial state, see initialise line 3 in Fig. 2.6.
- Simplified integration of the parameters w, l, p, and t during initialisation, see initialise lines 4–7 in Fig. 2.6.
- Increasing rate by $2w$ and decreasing capacity by the same amount. New rate+capacity: NORX64: $768 + 256$, NORX32: $384 + 128$.

Changes from v1.0 to v1.1:

- Branching: Added a missing -1 in $0 \leq i \leq \lceil |P|/r \rceil - 1$ for the case $p = 0$.
- Branching: Added a note that the value $\lfloor i/2^w \rfloor$, which is XORed to $s_{i,14}$, is only non-zero for very large messages.
- Payload Processing: In the parallel processing modes $p = 0$ and $p > 1$ full plaintext blocks P_i are added now directly to lane L_i for processing without padding. Only the last plaintext block P_{n-1} is padded.
- Chapter 3: Added security bounds for the NORX mode of operations from [36].
- §4.1: Added a remark concerning extensibility of the design.
- §4.3: Added software performance measurements for the Apple A7 chip and visualisations for all platforms.

8 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.

9 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.

10 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, 2014. http://competitions.cr.yp.to/caesar.html.

[4] Official website of NORX, 2014. https://www.norx.io.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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, 2008.

[10] 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.

[11] 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, 2013.

[12] 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.

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

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

[15] 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.

[16] 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, 2008.

[17] 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.

[18] 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.

[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. Cryptographic Sponge Functions, January 2011. http://sponge.noekeon.org/CSF-0.1.pdf.

[21] 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, 2011.

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

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

[29] 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.

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

[31] 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.

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

[33] 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.

[34] 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.

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

[36] Philipp Jovanovic, Atul Luykx, and Bart Mennink. Beyond $2^{c/2}$ Security in Sponge-Based Authenticated Encryption Modes. Cryptology ePrint Archive, Report 2014/373, 2014. http://eprint.iacr.org/2014/373.

[37] 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, 2010.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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, 2012.

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

[49] 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

A.1 Traces for F

To verify correctness of an F implementation we suggest to use the derivation of the NORX initialisation constants:

$$(u_0, \ldots, u_{15}) = \mathsf{F}^2(0, \ldots, 15)$$

The values of $u_0, \ldots, u_{15}$ are given in Table 2.4.

A.2 Full AEAD Computations

We assume that the following input data is given for NORX32 and NORX64, respectively:

NORX32		
type	data	#bytes
K :	00 01 ... 0E 0F	16
N :	F0 E0 ... 90 80	8
A :	00 01 ... 7E 7F	128
M :	00 01 ... 7E 7F	128
Z :	00 01 ... 7E 7F	128

NORX64		
type	data	#bytes
K :	00 01 ... 1E 1F	32
N :	F0 E0 ... 10 00	16
A :	00 01 ... 7E 7F	128
M :	00 01 ... 7E 7F	128
Z :	00 01 ... 7E 7F	128

The test vectors for the five proposed instances are given on the following pages. Intermediate values are snapshots of the state after initialisation (Fig. 2.5, line 1), after header processing (Fig. 2.5, line 2), after message encryption (Fig. 2.5, line 5), after trailer processing (Fig. 2.5, line 6), and after finalisation (Fig. 2.5, line 7).
For more tests we refer to the NORX software package available at [4]. The above test vectors were generated using `utils/debug.c`. See also `utils/check.c`, `utils/genkat.c`, and the various `kat.h` files for a more comprehensive test-framework.

NORX32-4-1

State after initialisation (Fig. 2.5, line 1)

```
2A878673  EE1C2B8C  94E41462  E6F99787
1B90CE6A  A18CD777  DA58C1A7  44A3BC15
91D3A9EC  235BBCE1  97C2FDF8  329735F2
4CEEFCCC  8737252B  D73B823B  4F53B645
```

State after header processing (Fig. 2.5, line 2)

```
C59FD03E  5785B42B  B6CBF54C  90004EDA
BB7D6D72  F9989DC6  E2A06D50  43A2C3AF
FDAAE2F5  FE32E309  5221792A  D9F96EAB
0C55EEC7  F3EF5FAC  E7FA58E1  22FD6D9C
```

State after message encryption (Fig. 2.5, line 5)

```
FEDBC5E6  142FB141  8334D280  1E116527
EFDE094C  1697E39F  7C21D218  1E92774B
D2095D7D  D979E560  82CF4349  EC30DD09
128F82BE  6544BFB8  EF1A52AC  5E5F1FED
```

State after trailer processing (Fig. 2.5, line 6)

```
BD51E53D  E1491C17  B0F2A15C  79A47756
DC954A9D  5FECCB91  6DF4BD69  8D971403
67C7B5F5  E6AE4FA1  2ABDC3D0  4B8CC3A4
8A926377  89F06163  E19986A1  EBBC5255
```

State after finalisation (Fig. 2.5, line 7)

```
1E131078  1EAB2EEA  235DA05D  99CBE3D4
02A9B200  F4398B31  6A8AFB2A  F20519CC
30374067  DAD9E66E  E73F2A1B  7B0B6C47
83719333  33E14AAC  C25B534A  F90DA3DA
```

Ciphertext and authentication tag

```
C :  F4 AF C8 E6 6D 2D 80 DE 0A 7F 71 9C 89 96 24 C9
     AD 89 6E C7 C6 17 39 D5 37 6D 06 48 C7 BC B2 04
     E5 7D B0 5C 6F 83 B3 FF 43 15 E8 A4 EF 2F 2C 85
     5F 21 EA 4C 51 AC 6D E5 75 77 3B A5 48 F3 6E 63
     6A 13 B9 79 D9 53 BB 91 29 8E A4 A6 E2 AA 27 40
     29 91 E0 DA 54 19 97 82 54 07 B2 F1 24 41 DE 3A
     E6 C5 DB FE 41 B1 2F 14 80 D2 34 83 27 65 11 1E
     4C 09 DE EF 9F E3 97 16 18 D2 21 7C 4B 77 92 1E

T :  78 10 13 1E EA 2E AB 1E 5D A0 5D 23 D4 E3 CB 99
```

NORX32-6-1

State after initialisation (Fig. 2.5, line 1)

```
49B4A0AD  45A8BD64  C9491C37  DA94AD3A
0A56F675  5A269CED  19DACB30  5A090476
B2BE631F  31C56682  F64767A3  65A5AE10
74DCEAB6  773F71A2  BBAEB53B  90FCEE05
```

State after header processing (Fig. 2.5, line 2)

```
D74A8393  6D0AC471  CC7A1E12  26C3729F
6978169B  08AB1D5C  B76FDD3A  4AF0DDCB
98B794C2  48D3E784  D64663CA  3B4AB8DE
AEF849D6  D2A7A723  0607C246  2A84390B
```

State after message encryption (Fig. 2.5, line 5)

```
1075D283  AC1F9114  15849073  43998991
FCAD7129  E6094691  6C509F88  3C4F7D24
5888E757  40E53C0F  01E81B2D  EA81FF9B
BD5560E0  37573B47  0BFBB5C5  E4249955
```

State after trailer processing (Fig. 2.5, line 6)

```
D6156EB9  0D1ADD96  5EA107A6  A291EEA4
6B9B9DA1  BED38408  E806FECD  AD0DF9BB
3102AB45  A12FA1F6  40CEF800  C171D4C3
F83CEDA8  466675EE  B09D8343  AB6C631A
```

State after finalisation (Fig. 2.5, line 7)

```
B6D1AC7A  6DA7DC80  4A62B573  0C148012
CD3CA96E  3E325DF0  55376AA9  2F958B5E
F42D34D7  F40ADD06  1FF77253  1D3349E6
6DB79321  ABF43F26  7B8C3CF2  BC87ADED
```

Ciphertext and authentication tag

```
C :  1A 59 BD 08 C4 09 90 97 ED 27 04 A8 7B 2F 8B 48
     E1 65 99 14 60 99 4F 3D 57 9B 1B 13 76 99 24 4E
     8C CB 0B 8E 15 F0 F0 3C 18 87 71 9C 03 A2 5A B3
     41 3D C3 B1 9D 85 D3 15 11 50 74 EE 84 51 95 BF
     EB A5 06 A1 8E 6A A5 A4 C8 8F C8 4B 0D 3F 53 5A
     7F B5 74 0F 16 6B 5B 9A 23 E8 78 F5 CD 55 5E 81
     83 D2 75 10 14 91 1F AC 73 90 84 15 91 89 99 43
     29 71 AD FC 91 46 09 E6 88 9F 50 6C 24 7D 4F 3C

T :  7A AC D1 B6 80 DC A7 6D 73 B5 62 4A 12 80 14 0C
```

NORX64-4-1

State after initialisation (Fig. 2.5, line 1)

```
E5F46094878A8B67  9D91985BC08A6DE0  7A257A96202FB03D  C3C8A1B9544950E4
A7DD2BB9DF0113E3  4CB4BA1AF1E08AE7  A11D70BE7A68DD9C  5F29B9540E0824A7
FA14572EC030682C  B3DEF83462277111  61079FBEEAE3DC47  379021175E684E47
48CBBAA6C5D79358  3EE1E2CB3BC69E6A  1825F80DF73CEC32  350D2B8123057B3C
```

State after header processing (Fig. 2.5, line 2)

```
26D6F79DAD83DD7E  CD0F5DDCC0669C87  7E6A4D9C4B7C73C5  6B9A809124B1E25A
94E9EDAF09B0B175  91C4D4FA12952C0A  4B8D3496625035FE  C0C605B6F658980E
2F3189200C6853B4  FCFF65878A7EB5BE  0B8F39E049E8DAAC  C8D39DFEA5F2B656
05D912C5460E50AE  329B8AAB29CFC0C4  F08C6BBB97C3EC98  5ECCDE8FDAF538E2
```

State after message encryption (Fig. 2.5, line 5)

```
F0B100A7748AECE2  99F9897C8FA0B3E1  D6152280E3A5234E  27148FD2F4FE5D71
7882AF4F2E417F1E  A6046C51725CF7EB  AD8AC693FBD2C107  BF2521681E512C0C
8CF120F5A42AE6F1  1BAF40D42F743C2A  A6A7D33AD16C0849  E381B8E46492A2B4
70DA708DE96D3A93  2F5EAC58BA0F53EF  9B1549ADD5DA2E0B  705DDB394B4101B3
```

State after trailer processing (Fig. 2.5, line 6)

```
77CF22AA996E7AC0  F505ACD994C83361  EE03E257C3EDACF7  AE5EA67159B7A126
6A38F48BD4EB0E77  09234C73133FCBDD  C8AA9235006E8D11  BB97EFEDF3550863
78750C1F4C49AC98  24E819297BA25825  E8189DBCADCD1F28  CAAFC91275D8719D
E4EA456D4954CD43  29CA6AEBEB98C24A  50983C29330FFB43  C2BC8DEFA8D9608F
```

State after finalisation (Fig. 2.5, line 7)

```
D16E157B3D5B1215  F88F48E821887734  EA37332A570DADA2  305766B464BB8443
07B68AEC3B1B4A8B  5FDF4F14249B8A92  BCD466C42F52871C  527989DBD1876236
D9EC97715DE78B2A  D9FE3FC995FE4638  E2C6BE970A0CED73  98292302A546DA7F
01F5400F30C79B7B  7DFF06AC557B048C  D91008CAD6885BFE  97EDE7D253393BF6
```

Ciphertext and authentication tag

```
C :  62 10 4A C4 97 FF 39 F9 51 60 A6 9E 07 3F 89 89
     40 55 F4 75 FA 8E 08 96 8E EC AD 33 9C BB 03 30
     9A A7 43 72 B4 C4 FB 3C E1 3C A2 C7 48 49 DB A1
     DA B1 69 D0 21 C1 D5 16 F6 61 A6 D2 1F C5 B9 37
     E4 59 B2 9D C4 DC FA 78 41 0C 7D 64 29 9A 30 9B
     9D B1 72 3C EF 74 21 E8 67 08 3D 26 83 39 B5 24
     E2 EC 8A 74 A7 00 B1 F0 E1 B3 A0 8F 7C 89 F9 99
     4E 23 A5 E3 80 22 15 D6 71 5D FE F4 D2 8F 14 27

T :  15 12 5B 3D 7B 15 6E D1 34 77 88 21 E8 48 8F F8
     A2 AD 0D 57 2A 33 37 EA 43 84 BB 64 B4 66 57 30
```

NORX64-6-1

State after initialisation (Fig. 2.5, line 1)

```
76FBD39C51C13C93  52757303FF789049  2BF64476E08BEAB3  7853121D4956D973
51CEC6828811E368  0814909AA24F5904  0F5B60F5A93774B0  6902478A97469300
ACEB8ABE024A000F  21971BBAC8696E13  2B75B95601458C00  7D0B1CFB1BE8C43B
318992BCBAE05D6A  6EE527E385FD3305  9219026A5BF673C4  41F58E8B64064E08
```

State after header processing (Fig. 2.5, line 2)

```
3935320CAC83A708  885681ED01EBC63A  055ACE3D2A35B24B  1E7B4321B816E51B
2D16F729A73E9C1D  B482ACAA96005B25  F44CDB6C6042C47D  D8142A91D9F90940
2439D8D89D8CAC25  E94E6341950A1083  655775BF551195A8  0E5939FA8F1CE9CD
C9B0B9D02EDFCC32  2E8E4EBD80047E51  6116C9853EFF4D6F  2BE9E97060FF36BE
```

State after message encryption (Fig. 2.5, line 5)

```
A0BC72BC1DCB2403  FF6107394AF0E50E  F3BD7276381906F9  F61F84D8396FB31C
BD723C8927B712FD  2344C7FD033CE0D4  FDE150253CED10B6  F4C42A8303AFC52D
5125F7A82BE38DCC  FAE61D10BE5F8055  2154210104E3B2B2  C14D13D536548CD9
D47BBE8AFA679DE1  AEB82B8A33B7F824  49B5CD9AD7A09059  1F9E8EF60FB85C51
```

State after trailer processing (Fig. 2.5, line 6)

```
D80DF9F3C207166E  893333B35BB7154A  42AB7295AE0B79DB  90D146E31BA8D93B
78C9AAAC4434B284  FD0AD55007C6FBAE  08ACE9E640708176  D65DE5AD9DD4B687
DC3680BA6D6D7F29  F043599666D3321A  B24D01130ACBFEA4  9E5F4E4AD2A0642D
08FA43BCBBAE23B9  D4EE30E13F385752  058CEF9D316E3B8C  C9D8DA2797AF2D5E
```

State after finalisation (Fig. 2.5, line 7)

```
F5F912AC9877E37D  6DA8385714926C74  77A3D1989F00590C  F2388309731AF4D4
3EC35DA675CD5E01  E9D8F8FE07CC4467  314F49CA4998D7EA  67025BCA91C7470B
1B874F4FDFACDC07  0F0EB5FBA371BC73  520D19766BE5187F  7F97635256E74DA2
CE5A62B68D57E346  A00D3648E908217B  C9119CE74284E01E  61F81A7D92EC8E34
```

Ciphertext and authentication tag

```
C :  7E B0 E2 87 73 2C 75 9C 96 09 94 43 77 4C B6 58
     5D FD F0 E0 D6 A4 BF AB DA 9E 7F 01 34 78 83 60
     96 5B F1 F6 21 1D B6 E8 58 B7 6A A4 4B 5B 08 16
     CF 27 87 7C 81 AF C7 97 C4 72 02 79 0E 38 C8 8C
     4C 2F 0F 71 34 45 26 84 C6 6E 37 8B B9 67 2C AA
     24 5F 23 E0 5E 00 7F C9 15 CC 68 AC 49 56 35 74
     03 24 CB 1D BC 72 BC A0 0E E5 F0 4A 39 07 61 FF
     F9 06 19 38 76 72 BD F3 1C B3 6F 39 D8 84 1F F6

T :  7D E3 77 98 AC 12 F9 F5 74 6C 92 14 57 38 A8 6D
     0C 59 00 9F 98 D1 A3 77 D4 F4 1A 73 09 83 38 F2
```

NORX64-4-4

State after initialisation (Fig. 2.5, line 1)

```
79279318B635A587  55E53B1A445B5D6D  171E0A5CCAD10A00  4D1526E3063086FA
925E3EC06806B51C  B26465410FE38799  E12481B790CE26E1  8D1EE2B1FA9EC185
656BB6DF707B4E1E  A93F8917B2365053  99301AB1AD25771E  1A975A794358B833
8D9D473B25191370  E7711D44BD741FAA  8870F157F97AEF97  FFB036A735876101
```

State after header processing (Fig. 2.5, line 2)

```
FB7F76C31462B4F9  269D1488DEE4BD81  C828CF596BEB3BD3  B336468127606B8F
E6077A88658532CF  C44FF43E2A0C07C5  F6541DF4A5A06734  88962773B9093E7E
AA76A9908B726649  EDEA871536D1919D  EE5FB237FBDF3E98  F5FF8F327A3ED9E5
F10BF1DE42674CA4  D8A1B9CE0518328D  59BFD2D870B1BAD9  008677FC94D34927
```

State after message encryption (Fig. 2.5, line 5)

```
A2069B67D13B46DF  4119B60417031BC0  F23445E71EC32151  030720B7934359CB
6368500DA0D046A5  E320CDD0CEBF33BD  B494EEEA4AF8908F  E5448DA7E3ABE1F9
976C6EEF22230EBA  AA9BAF73EC5135D7  FDAA60AA4952CC18  CA67DF94A4D24FFD
C33F4C91106983FD  A44F9FF3DDBCF30E  6D0FC1FA51710C81  0C6630B86954CFD1
```

State after trailer processing (Fig. 2.5, line 6)

```
4367F1E179F7913D  E6EB2721923C82DF  AD2CC1E801A27252  6DDB37F0B3C42D08
14552E6AE052F88A  75339EB2A021994E  CD2153908AED5B00  EE06BE0321E13D8F
BD8768435F3E9D1A  0BD9DB9FF05E952B  46B09EFBE7DDF449  A8E4CC2224EDAF4C
11EC7B31D3C9F76F  3373C9023B69D494  0BE0E45A1A5B1BB8  18A21284F2A40BC1
```

State after finalisation (Fig. 2.5, line 7)

```
2487F4E9FD78F953  5FEFF0053BC4A243  751A08512ED9B0AE  64A0DB077C939198
4B24D3930F8CD4F8  67F43EA25223F4C0  3D1257445E1B0D08  0460E5E01846B2FF
5A9DCF3D677793D2  46741E5DF0B5E61B  A287DD12AE3EE8FC  4281D43264E0ACCE
4793F2F1FDC4DB99  CCF0E20D140E89D8  7C965BFB5E12CEDB  B1DB82BB7A8D4D0B
```

Ciphertext and authentication tag

```
C :  83 DB 70 24 8B 3A 2C A6 4F EE 48 72 6A C4 7B 99
     93 EF 41 16 1A 6F 35 40 7A 4A D6 15 C4 28 43 C7
     63 7A 55 D4 D8 67 D5 BC 71 1A FF 8F DB E6 C5 4E
     26 8B D4 91 01 46 6A 9E F0 FA 53 01 03 43 6B 33
     03 65 D2 C9 81 B7 91 33 74 8F 87 31 22 54 60 7F
     34 5E DF 51 5A AA D0 E0 B1 99 D9 9E D2 1B 4B 29
     4D 40 6D 5A 05 CB C1 28 E7 26 C2 AC 12 F8 EA C4
     CC 74 B9 07 13 A9 96 8E 01 51 AA BC D2 1A 04 72

T :  53 F9 78 FD E9 F4 87 24 43 A2 C4 3B 05 F0 EF 5F
     AE B0 D9 2E 51 08 1A 75 98 91 93 7C 07 DB A0 64
```

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.

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

	Inverse NORX32				Inverse ChaCha (32-bit)			
l	min	max	avg	med	min	max	avg	med
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)			
l	min	max	avg	med	min	max	avg	med
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

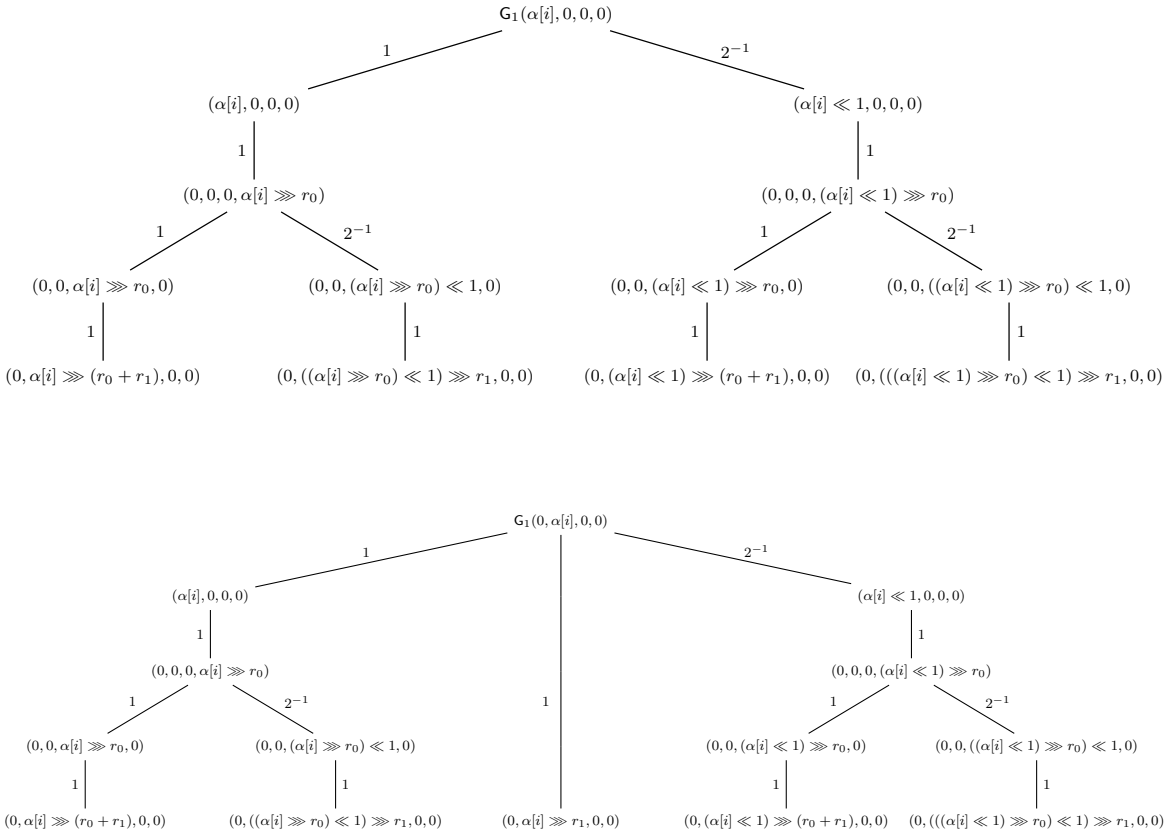
B.2 Addenda to cryptanalysis

B.2.1 Visualisation of differentials for G_1

Fig. 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

Fig. 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.



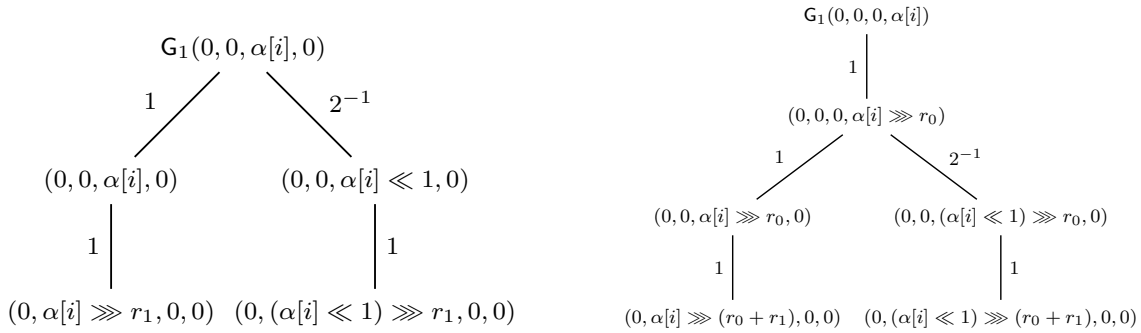


Figure B.1: Relations of the G_1 output differences

????	????	????	??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