NORX: Parallel and Scalable AEAD

Jean-Philippe Aumasson[1], Philipp Jovanovic[2], and Samuel Neves[3]

[1] Kudelski Security, Switzerland
jeanphilippe.aumasson@gmail.com
[2] University of Passau, Germany
jovanovic@fim.uni-passau.de
[3] University of Coimbra, Portugal
sneves@dei.uc.pt

Abstract. This paper introduces NORX, a novel authenticated encryption scheme supporting arbitrary parallelism degree and based on ARX primitives, yet not using modular additions. NORX has a unique parallel architecture based on the monkeyDuplex construction, with an original domain separation scheme for a simple processing of header, payload and trailer data. Furthermore, NORX specifies a dedicated datagram to facilitate interoperability and avoid users the trouble of defining custom encoding and signalling. 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. On a Haswell processor, a serial version of NORX runs at 2.51 cycles per byte. Simulations of a hardware architecture for 180 nm UMC ASIC give a throughput of approximately 10 Gbps at 125 MHz.

Keywords: authenticated encryption, stream cipher, cryptographic sponges.

1 Introduction

We introduce the NORX[1] family of authenticated ciphers, a candidate in the CAESAR competition. NORX uses a parallel and scalable architecture based on the monkeyDuplex construction [1,2], where the parallelism degree and tag size can be tuned arbitrarily. The NORX core is inspired by the ARX primitive ChaCha [3], however it replaces integer addition with the approximation $(a \oplus b) \oplus ((a \wedge b) \ll 1)$[2], with the aim to simplify differential cryptanalysis and improve hardware efficiency. Although, bitwise logic operations are frequently used in cryptographic primitives, we are not aware of any other algorithm using the above approximation of integer addition.

On a Haswell processor (Intel's latest microarchitecture), a *serial* version of NORX achieves 2.51 cycles per byte. For long messages ($\geq$ 4 KiB), our 4-wise parallel version is expected to be four times as fast when run on four cores (that is, more than 5 GiBps at 3.5 GHz).

[1] The name stems from "NO(T A)RX" and is pronounced like "norcks".
[2] Derived from the well-known identity $a + b = (a \oplus b) + (a \wedge b) \ll 1$ [4].

M. Kutyłowski and J. Vaidya (Eds.): ESORICS 2014, Part II, LNCS 8713, pp. 19–36, 2014.
© Springer International Publishing Switzerland 2014

In ASIC, NORX's fastest *serial* architecture is expected to achieve a throughput of about 10 Gbps at 125 MHz on a 180 nm technology. As for software implementations, the tunable parallelism allows NORX to reach even higher speeds.

We have not filed and are not aware of patents, patent applications, or other intellectual-property constraints relevant to use of the cipher. The source code of the reference implementation is published under a public domain-like licence (CC0 1.0), see [5].

Outline. Section 2 specifies the NORX family of AEAD schemes, as well as a datagram structure aiming to improve interoperability of NORX implementations. Section 3 describes the expected strength of NORX. Section 4 motivates the design decisions. Section 5 reports on software performance measurements and on preliminary results of a hardware performance evaluation. Finally, Section 6 presents preliminary cryptanalysis results.

2 Specification

2.1 Notations

Hexadecimal numbers are denoted in `typewriter` style, for example $\texttt{ab} = 171$. A *word* is either a 32-bit or 64-bit string, depending on the context. Data streams (as byte arrays) are parsed to word arrays in little-endian order. We denote by $a \parallel b$ the concatenation of strings a and b, by $|x|$ the bit length of x, and by $\mathsf{hw}(x)$ its Hamming weight. We use the standard notations $\neg$, $\wedge$, $\vee$ and $\oplus$ for bitwise negation, AND, OR and XOR; $x \ll n$ and $x \gg n$ for left- and right-shift; $x \lll n$, $x \ggg n$ for left- and right-rotation of x by n bits.

2.2 Generalities

NORX is parameterised by a *word size* of $W \in \{32, 64\}$ bits, a *number of rounds* $1 \leq R \leq 63$, a *parallelism degree* $0 \leq D \leq 255$, and a *tag size* $|A| \leq 10W$ bits. We denote a NORX instance by NORXW-R-D-$|A|$.

By default NORXW-R-D uses $|A| = 4W$. For example, NORX64-6-1 has $(W, R, D, |A|) = (64, 6, 1, 256)$.

Encryption Interface. NORX encryption takes as input a key K of $4W$ bits, a nonce N of $2W$ bits, and a message $M = H \parallel P \parallel T$ where, H is a *header*, P a *payload*, and T a *trailer*. $|H|$, $|P|$, and $|T|$ are allowed to be 0. NORX encryption produces a ciphertext C, with $|C| = |P|$, and an *authentication tag* A.

Decryption Interface. NORX decryption is similar to encryption: Besides K and N, it takes as input a message $M = H \parallel C \parallel T$, where H and T denote header and trailer, and C the *encrypted payload*, with $|H|$, $|C|$, and $|T|$ are again allowed to be 0. The last component of the input is an authentication tag A. Decryption either returns failure, upon failed verification of the tag, or produces a plaintext P of the same size as C if the tag verification succeeds.

2.3 Layout Overview

NORX relies on the monkeyDuplex construction [1,2], enhanced by the capability of parallel payload processing. The number of parallel encryption lanes L_i is defined by the parameter $0 \leq D \leq 255$. For the value $D = 1$, the NORX layout is similar to a standard sequential duplex construction, see Figure 1. For $D > 1$, the number of lanes is bounded by the latter value, e.g. for $D = 2$ see Figure 2. If $D = 0$ ("unbounded" parallelism), the number of lanes L_i is bounded by the size of the payload.

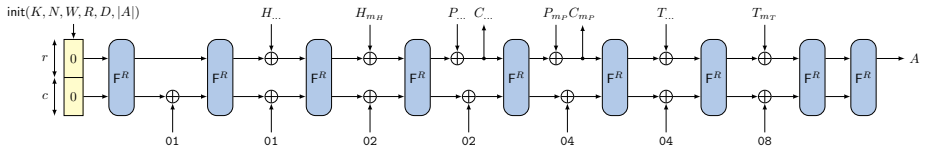

Fig. 1. Layout of NORX with parallelism $D = 1$

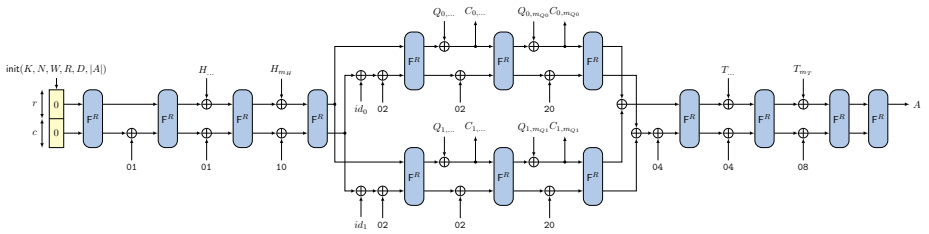

Fig. 2. Layout of NORX with parallelism $D = 2$

The round function F is a permutation of $b = r + c$ bits, where b is called the *width*, r the *rate* (or block length), and c the *capacity*. We call F a *round* and F^R denotes its R-fold iteration. The internal state S of NORX64 has $b = 640+384 = 1024$ bits and that of NORX32 has $b = 320+192 = 512$ bits. The state is viewed as a concatenation of 16 words, i.e. $S = s_0 \parallel \cdots \parallel s_{15}$, which are conceptually arranged in a 4×4 matrix, where $s_0, \ldots, s_9$ are called the *rate words*, used for data block injection, and $s_{10}, \ldots, s_{15}$ are called the *capacity words*, which remain untouched during absorbing and squeezing.

2.4 The Round Function F

F processes a state S by first transforming its columns with

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

and then transforming its diagonals with

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

Those two operations are called *column step* and *diagonal step*, as in BLAKE2 [6], and will be denoted by col and diag. The permutation G transforms four words a, b, c, d by computing (top-down, left-to-right):

$$\begin{array}{l} a \longleftarrow (a \oplus b) \oplus \big((a \wedge b) \ll 1\big) \\ d \longleftarrow (a \oplus d) \ggg r_0 \\ c \longleftarrow (c \oplus d) \oplus \big((c \wedge d) \ll 1\big) \\ b \longleftarrow (b \oplus c) \ggg r_1 \end{array} \quad \left| \quad \begin{array}{l} a \longleftarrow (a \oplus b) \oplus \big((a \wedge b) \ll 1\big) \\ d \longleftarrow (a \oplus d) \ggg r_2 \\ c \longleftarrow (c \oplus d) \oplus \big((c \wedge d) \ll 1\big) \\ b \longleftarrow (b \oplus c) \ggg r_3 \end{array} \right.$$

The rotation offsets (r_0, r_1, r_2, r_3) are $(8, 19, 40, 63)$ for NORX64, and $(8, 11, 16, 31)$ for NORX32. They were chosen as described in Section 4.

2.5 Encryption and Tag Generation

NORX encryption can be divided into three main phases: *initialisation*, *message processing*, and *tag generation*. Processing of a message $M = H \parallel P \parallel T$ is done in one to five steps: *header processing*, *branching* (for $D \neq 1$ only), *payload processing*, *merging* (for $D \neq 1$ only), and *trailer processing*. The number of steps depends on whether H, P, or T are empty or not, and whether $D = 1$ or not. NORX skips processing phases of empty message parts. For example, in the simplest case when $|H| = |T| = 0$, $|P| > 0$, and $D = 1$, message processing is done in one step, since only the payload P needs to be encrypted and authenticated.

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

Padding. NORX uses the *multi-rate padding* [2], a padding rule defined by $\mathsf{pad}_r : X \longmapsto X \parallel \mathtt{10}^q\mathtt{1}$ with bitstrings X and $\mathtt{10}^q\mathtt{1}$, and $q = (-|X| - 2) \bmod r$. This extends X to a multiple of the rate r and guarantees that the last block of $\mathsf{pad}_r(X)$ differs from the all-zero block $\mathtt{0}^r$. Note, that there are three special cases:

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

Domain Separation. NORX performs domain separation by XORing a *domain separation constant* to the least significant byte of s_{15} each time before the state is transformed by the permutation F^R. Distinct constants are used for the different phases of message processing, for tag generation, and in case of $D \neq 1$, for branching and merging steps. Table 1 gives the specification of those constants and Figures 1 and 2 illustrate their integration into the state of NORX.

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

Table 1. Domain separation constants

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

the new constant is 02. Then, as long as the next block is from the payload, 02 is used, and so on.

For the extra initial and final permutations no domain separation constants are used, which is equivalent to XORing 00 to s_{15}. Additionally, this allows NORX to skip unneeded processing phases, as already discussed above. For the special case $D \neq 1$ and $|P| = 0$ not only payload processing is skipped but also branching and merging phases.

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

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

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

   where u_0 to u_3 are as follows for NORX32 (left) and NORX64 (right):

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

   The other constants are computed by

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

   for $j \in \{0, 1\}$. See Section 4 for a discussion on these constants.
2. **Parameter Integration.** The parameters D, R, W and $|A|$ are integrated into the state S by XORing $(R \ll 26) \oplus (D \ll 18) \oplus (W \ll 10) \oplus |A|$[3] to s_{14} followed by an update of S with F^R.
3. **Finalisation.** A domain separation constant v, whose value is determined as shown above, is XORed into s_{15}. Subsequently, S is updated once more by F^R.

Note, that step 1 and part of step 2, namely integration of the parameters, are illustrated as $\mathsf{init}(K, N, W, R, D, |A|)$ in Figures 1 and 2.

[3] This layout is used to avoid XOR-collisions between the parameters.

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

1. **Header Processing.** If $|H| = 0$, this step is skipped, otherwise let $\mathsf{pad}_r(H) = H_0 \parallel \cdots \parallel H_{m_H-1}$ denote the padded header data, with r-bit sized header blocks $H_l = h_{l,0} \parallel \cdots \parallel h_{l,9}$ and $0 \leq l \leq m_H - 1$. Then H_l is processed by:
$$\begin{aligned} s_j &\longleftarrow s_j \oplus h_{l,j}, \quad \text{for} \quad 0 \leq j \leq 9 \\ s_{15} &\longleftarrow s_{15} \oplus v \\ S &\longleftarrow \mathsf{F}^R(S) \end{aligned}$$
2. **Branching.** This step is only performed if $D \neq 1$ and $|P| > 0$. In that case, NORX encrypts payload data on parallel *lanes* L_i, with $0 \leq i \leq D - 1$ if $D > 1$, or $0 \leq i \leq \lceil |P| \ / \ r \rceil - 1$ if $D = 0$. For each lane L_i, a copy $S_i = s_{i,0} \parallel \cdots \parallel s_{i,15}$ of the state S is created. The *lane number* i and the domain separation constant $v = \mathtt{02}$ are integrated into the least significant bytes of $s_{i,13} \parallel s_{i,14}$ and $s_{i,15}$, respectively. Finally each S_i is updated by F^R. That is, NORX does
$$\begin{aligned} S_i &\longleftarrow S \\ (s_{i,14}, s_{i,13}) &\longleftarrow (s_{i,14}, s_{i,13}) \oplus (\lfloor i \ / \ 2^W \rfloor, i \bmod 2^W) \\ s_{i,15} &\longleftarrow s_{i,15} \oplus v \\ S_i &\longleftarrow \mathsf{F}^R(S_i) \end{aligned}$$
3. **Payload Processing.** If $|P| = 0$, this step is skipped. Otherwise, payload data is padded using the multi-rate padding and then encrypted. Let $\mathsf{pad}_r(P) = P_0 \parallel \cdots \parallel P_{m_P-1}$, then we distinguish three cases:
   - $D = 1$: This is the standard case, which requires no further modifications.
   - $D > 1$: In this case, a fixed number of lanes L_i is available for payload encryption, with $0 \leq i \leq D - 1$. An r-bit sized block P_j, with $0 \leq j \leq m_P - 1$, is processed by lane L_i if $i \equiv j \bmod D$. In other words, the padded payload blocks are distributed through the lanes in a round-robin fashion.
   - $D = 0$: Here, the number of lanes L_i is determined by the number m_P of padded payload blocks. Each r-bit sized block is processed on its own lane, i.e. block P_i is encrypted on L_i, with $0 \leq i \leq m_P - 1$.

   The data encryption of a single block works equivalently for each value of D, hence we describe it only in a generic way. Again, let $\mathsf{pad}_r(P) = P_0 \parallel \cdots \parallel P_{m_P-1}$ be the padded payload data. To encrypt $P_l = p_{l,0} \parallel \cdots \parallel p_{l,9}$ and get a new ciphertext block $C_l = c_{l,0} \parallel \cdots \parallel c_{l,9}$ the following steps are executed
$$\begin{aligned} s_j &\longleftarrow s_j \oplus p_{l,j}, \text{ for } 0 \leq j \leq 9 \\ c_{l,j} &\longleftarrow s_j \\ s_{15} &\longleftarrow s_{15} \oplus v \\ S &\longleftarrow \mathsf{F}^R(S) \end{aligned}$$

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

4. **Merging.** This step is only performed if $D \neq 1$ and $|P| > 0$. After processing of P, the states S_i are merged back into a single state S. Then, a domain separation constant v is integrated, and S is updated by F^R:

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

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

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

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

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

2.6 Decryption and Tag Verification

NORX decryption mode is similar to the encryption mode. The only two differences are described below.

Message Processing. Processing header H and trailer T of $M = H \parallel C \parallel T$ is done in the same way as for encryption. Decryption of the encrypted payload C is achieved as follows:

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

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

Tag Verification. This step is executed after tag generation. Let A and A' denote the *received* and the *generated tag*. If $A = A'$, tag verification succeeds; otherwise it fails, the decrypted payload is discarded and an error is returned.

2.7 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. More specifically, we introduce two distinct types of datagrams, depending on whether the parameters of NORX are fixed or need to be signalled in the datagram header.

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*. The datagram for fixed parameters thus only needs to contain N, H, C, T, and A, as well as information to parse those elements. It is depicted in Appendix A.

We encode the byte length of H and T on 16 bits, allowing for headers and trailers of up to 64 KiB, a large enough value for most applications. The byte length of C 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[4]. Similarly, to frame check sequences in data link protocols, the tag is added as a *trailer of the datagram* specified. The data H, C, and T 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. The length of the datagram header is 28 bytes for NORX64 and 16 bytes for NORX32.

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

[4] Note that NORX is capable of (safely) processing much longer messages; those are just the maximum values when our proposed datagrams are used.

3 Expected Strength

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

1. **Unique Nonces.** Each key and nonce pair should not be used to process more than one message.
2. **Abort on Verification Failure.** If the tag verification fails, only an error is returned. In particular, the decrypted plaintext and the wrong 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 k-bit sized key of an ideal cipher.

Note that NORX restricts the number of messages processed with a given key: in [7] 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. NORX sets it to $e_{64} = 128$ for 64-bit and $e_{32} = 64$ for 32-bit, which corresponds in both cases to the size of the nonce. NORX has capacities of $c_{64} = 384$ (64-bit) and $c_{32} = 192$ (32-bit). Hence, security levels of at least $c_{64} - e_{64} - 1 = 384 - 128 - 1 = 255$ bits for NORX64 and $c_{32} - e_{32} - 1 = 192 - 64 - 1 = 127$ bits for NORX32 are expected (see [7]).

Moreover, [8] shows that the NORX mode of operation achieves security levels for authenticity and confidentiality of $\mathsf{min}\{2^{b/2}, 2^c, 2^{|K|}\}$ (recall that $|K| = |A|$), for all $0 \leq D \leq 255$, assuming an ideal underlying permutation F and a nonce respecting adversary.

4 Rationale

The Parallel Duplex Construction. The layout of NORX is based on the monkeyDuplex construction [1,2] enhanced by the capability of parallel payload processing. The *parallel duplex construction* is similar to the tree-hashing mode for sponge functions [9]. It allows NORX to take advantage of multi-core processors and enables high-throughput hardware implementations. Associated data can be authenticated as header and/or trailer data but only on a single lane. We felt that it is not worth the effort to enable processing of H and T in parallel, as they are usually short. The number of lanes is controlled by the parallelism degree $0 \leq D \leq 255$, which is a fixed instance parameter. Hence, two instances

with distinct D values cannot decrypt data from each other. Obviously, the same holds for differing W and R values.

To ensure that the payload blocks on parallel lanes are encrypted with distinct key streams, NORX injects a unique id into each of the lanes during the branching phase. 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 creation of the authentication tag.

The Permutations G and F. The function G of NORX is inspired by the quarterround function of the stream cipher ChaCha [3]. NORX adopts this core function almost one-to-one, with the only difference being the replacement of the integer addition by $z = (x \oplus y) \oplus \big((x \wedge y) \ll 1\big)$ with n-bit words x, y and z. This operation uses bitwise AND to introduce non-linearity and mimics integer addition of two bit strings x and y with a 1-bit carry propagation. Thus it provides, in addition to non-linearity, also a slight diffusion of bits. Clearly, G is invertible, and thus F is invertible as well.

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 strategy was previously proposed in [1] and has only minor effects on the overall performance, but increases the security of NORX. The minimal value of $R = 4$ is based on the following observations:

1. The best attacks on Salsa20 and ChaCha [10,11] break 8 and 7 rounds, respectively, which roughly corresponds to 4 and 3.5 rounds of the NORX core. However this is within a much stronger attack model than that provided by the duplex construction of NORX.
2. The preliminary cryptanalysis of NORX as presented in Section 6. The best differentials we were able to find belong to a class of high-probability truncated differentials over 1.5 rounds and a class of impossible differentials over 3.5 rounds. Despite the fact that those differentials cannot be used to mount an attack on NORX, it might be possible to find similar differentials, using more advanced cryptanalytic techniques, which could be used for an attack.

Choice of Constants. The values $u_0, \ldots, u_3$ correspond to the first digits of π. The other six constants $u_4, \ldots, u_9$ are derived iteratively from $u_0, \ldots, u_3$ as described in Section 2.5. Their purpose is to bring asymmetry during initialisation and to limit an attacker's freedom where he might inject differences.

The domain separation constants serve to separate the different processing phases of NORX, which is important for the indifferentiability proofs of the duplex construction [12,2,8]. 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, like slide attacks (see Section 6.3).

Choice of Rotation Offsets. The rotation offsets as used in F, see Section 2.4, provide a balance between security and efficiency. Their values were selected such that at least two out of four offsets are multiples of 8 and the remaining offsets are odd values of the form $8n \pm 1$ or $8n \pm 3$, with a preference for the first shape. The motivation behind those criteria is as follows: an offset which is a multiple of 8 preserves byte alignment and thus is much faster than an unaligned rotation on many architectures. Many 8-bit microcontrollers have only 1-bit shifts, so for example rotations by 5 bits are particularly expensive. Using aligned rotations, i.e. permutations of bytes, greatly improves 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 thus increase diffusion.

To find good rotation offsets and assess their diffusion properties, we used an automated search combined with a simple diffusion metric. The offsets we finally chose achieve full diffusion after F^2 and offer good performance.

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 [9] it has been proven that the multi-rate padding satisfies those properties. Moreover, it is simple to describe, easy to implement, very efficient and increases the complexity of certain kind of attacks, like slide attacks (see Section 6.3).

5 Performance

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

5.1 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).

NORX lends itself well to implementations taking advantage of SIMD extensions present in modern processors, such as AVX or NEON. The typical vectorized implementation of NORX, when $D = 1$, works in full rows on the 4×4 state, and computes column and diagonal steps of F in parallel.

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

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

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

This tweak saves up to 1 cycle per instruction sequence, of which there are 4 per G, at the cost of 1 extra instruction. In a sufficiently parallel architecture, this can save at least $4 \times 2 \times R$ cycles, which translates to $6.4R/W$ cycles per byte saved overall.

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 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`.

The top speed of NORX (for $D = 1$), in terms of bytes per second, was achieved by an AVX2 implementation of NORX64-4-1 on a Haswell CPU, listed in Table 2. For long messages (≥ 4 KiB), it achieves a throughput of about 1.39 GiBps (2.51 cycles per byte at 3.5 GHz). The overhead for short messages (≤ 64 bytes) is mainly due to the initialisation and finalisation rounds (see Figure 1). However, the cost per byte quickly decreases, and stabilizes for messages longer 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.

5.2 Hardware

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

A hardware architecture was designed, supporting parameters $W \in \{32, 64\}$, $R \in \{2, \ldots, 16\}$ and $D = 1$. It was synthesized with the Synopsys Design Compiler for an ASIC using 180 nm UMC technology. The implementation was targeted at high data throughput. The requirements in area amounted to about

Table 2. Software performance of NORX in cycles per byte

Intel Core i7-2630QM at 2.0 GHz						Intel Core i7-4770K at 3.5 GHz					
data length [bytes]		long	1536	576	64	data length [bytes]		long	1536	576	64
NORX64-6-1	Ref	7.69	9.08	11.54	37.75	NORX64-6-1	Ref	6.63	7.77	9.85	32.12
	AVX	4.94	5.90	7.52	24.81		AVX2	3.73	4.47	5.71	19.19
NORX64-4-1	Ref	5.28	6.24	7.94	26.00	NORX64-4-1	Ref	4.50	5.27	6.71	22.06
	AVX	3.28	3.91	5.03	16.69		AVX2	2.51	3.01	3.83	13.06
Intel Core i7-3667U at 2.0 GHz						Samsung Exynos 4412 Prime (Cortex-A9) at 1.7 GHz					
data length [bytes]		long	1536	576	64	data length [bytes]		long	1536	576	64
NORX64-6-1	Ref	7.04	8.32	10.59	34.87	NORX64-6-1	Ref	37.04	44.55	57.99	203.06
	AVX	5.04	6.03	7.71	25.44		NEON	13.17	16.76	23.10	94.56
NORX64-4-1	Ref	4.92	5.86	7.43	24.93	NORX64-4-1	Ref	26.56	32.21	42.35	152.25
	AVX	3.37	4.01	5.16	17.18		NEON	8.94	11.81	16.81	74.12

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 similar results as presented in [13] for ChaCha.

5.3 Comparison to AES-GCM

AES-GCM, the current de-facto standard for authenticated encryption, achieves very high speeds when the *AES New Instructions* (AES-NI) extension is available. Gueron reports 1.31 cpb for AES256-GCM on a Haswell processor [14]. In that case, NORX is only about half as fast as AES-GCM (the difference is around 1.2 cpb). The situation is different if AES-NI is not available, which is the case for the majority of platforms. We expect that NORX outperforms AES-GCM in these cases. For example, in [15] a constant-time implementation of AES128-GCM is presented, reaching 20.29 cpb on a Nehalem processor, while a vulnerable implementation reaches 10.12 cpb. These speeds are likely to be somewhat better on modern architectures, but certainly not below 3 cpb and especially not for constant-time implementations. On the other hand, NORX was designed to run in constant time, therefore such a protected implementation should have comparable performance to the results presented in Section 5.1.

6 Preliminary Cryptanalysis

This section presents preliminary results on the cryptanalysis of NORX. For a more thorough version, especially with respect to differential and rotational properties, we refer to [16].

6.1 Differential Cryptanalysis

We show how to construct high-probability differentials for the round function F^R when R is small. We focus on NORX64, but similar considerations hold for NORX32.

We consider a simple attack model where the initial state is 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. To find such simple differentials, we decomposed G into two functions G_1 and G_2, i.e. $\mathsf{G} = \mathsf{G}_2 \circ \mathsf{G}_1$, such that G_1 corresponds to the first part of G (i.e. up to the rotation $\ggg r_1$) and G_2 to the second. Then, we analysed the behaviour of G_1 on 1-bit input differences. Exploiting the fact that many differences are deleted by the shift $\ll 1$ when the active bit is in the MSB, we found three high-probability differentials of G with a low-weight output, as shown in Table 3. Extending those differentials to F delays the diffusion by one step. Input differences with other combinations of active MSBs lead to similar output differences, but we found none with a lower or equal Hamming weight as the above. Using the first differential of the above, we derived a truncated differential over 3 steps (i.e. $\mathsf{F}^{1.5}$) that 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 part. We expect that advanced search techniques are able to find better differentials for a higher number of rounds of F, e.g. where the sparse difference occurs in a later step than in the first.

Table 3. High-probability, low-weight differentials of G

Input / Output Difference of G	$\Pr(\cdot)$
8000000000000000, 8000000000000000, 8000000000000000, 0000000000000000 0000000000000000, 0000000000000001, 8000000000000000, 0000000000000000	1
0000000000000000, 8000000000000000, 8000000000000000, 8000000000000000 8000000000000000, 0000000001000001, 8000000000800000, 0000000000800000	2^{-1}
0000000000000000, 8000000000000000, 8000000000000000, 8000000000000000 8000000000000000, 0000000003000001, 8000000001800000, 0000000000800000	2^{-1}

Impossible Differentials. We show how to construct an impossible differential using the *miss-in-the-middle* approach. In forward direction we use a probability-1 truncated differential over 1.5 rounds with an input difference having active bits in the first 3 MSBs of the input to G in the first column of the state, see Table 3. We set (0000000000000000, 0000000000000000, 8000000000000000, 0000000000000000) as the difference in the third column in backward direction. Applying $\mathsf{F}^{1.5}$ to the state in forward direction and $\mathsf{F}^{-1.5} \circ \mathsf{col}^{-1}$ to the state in backward direction, results in a conflict in the 2nd bit of the 14th word. In forward direction this bit is always 1 and in backward direction it is always 0. We validated the impossible differential empirically in 2^{32} runs, starting in both directions from random states having the above differences. Equivalent impossible differentials can be constructed by varying the columns where the differences are injected. We were unable to construct an impossible differential for more than 3.5 rounds.

Remark. Neither the simple nor the impossible differentials can be used to attack NORX if the attacker is nonce-respecting: first of all the initialisation process prevents an attacker to set the required input difference in forward direction, i.e. active bits in 3 consecutive MSBs of a column. Once the initialisation is finished, the attacker could theoretically set those differences in the first or second column, but it would have no effect, as two states initialised with different nonces have a far too big distance from each other. Additionally the capacity part is completely unknown to the attacker.

6.2 Algebraic Cryptanalysis

Algebraic attacks on cryptographic algorithms discussed in the literature [17,18] 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.

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

Table 4. Properties of the ANF of G

	# polynomials by degree						#monomials			
	3	4	5	6	7	8	min	max	avg	median
32-bit	2	6	58	2	8	52	12	489	242	49.5
64-bit	2	6	122	2	8	116	12	489	253	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 of the whole state after a col or diag step. Due to memory constraints, we were unable to construct[5] 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 and due to a rapid degree growth and huge state sizes it is unlikely that algebraic attacks can be successfully used against the AEAD scheme.

6.3 Other Properties

Fixed Points. The G permutation and thus any iteration of the round function F have a trivial distinguisher: the fixed points $\mathsf{G}(0) = 0$ and $\mathsf{F}^R(0) = 0$. Nevertheless it, seems hard to exploit this property, as hitting the all-zero state is as

[5] Using SAGE [19] on a workstation with 64 GiB RAM.

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. Furthermore, we used the constraint solver STP [20] 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. Therefore, we find it unlikely that there are any other fixed points in NORX64 besides the zero-to-zero point.

Slide Attacks. Slide attacks try to exploit the symmetries that consist of the iteration of a number of identical rounds. To protect sponge constructions against slide attacks, two simple defenses can be found in the literature: [21] proposes to add a non-zero constant to the state just before applying the permutation and [22] recommends 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 defenses should hold for the former, too, and thus for NORX. With the domain separation and multi-rate padding both defensive mechanisms are already integrated into NORX.

Rotational Cryptanalysis. NORX includes several defenses against exploitable rotation-invariant behaviour: 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. The non-linear operation of NORX contains a non rotation-invariant bit-shift $\ll 1$, and finally, the duplex construction 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.

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. Finally, the authors also thank the anonymous reviewers for their efforts and for their very helpful comments regarding this paper.

References

1. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Permutation-based Encryption, Authentication and Authenticated Encryption. Presented at DIAC 2012, Stockholm, Sweden, July 05-06 (2012)

2. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 320–337. Springer, Heidelberg (2012)
3. Bernstein, D.J.: 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
4. Knuth, D.E.: The Art of Computer Programming. Combinatorial Algorithms, Part 1, vol. 4A. Addison-Wesley, Upper Saddle River (2011), http://www-cs-faculty.stanford.edu/~uno/taocp.html
5. Official website of NORX (2014), https://www.norx.io
6. Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: Simpler, Smaller, Fast as MD5. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 119–135. Springer, Heidelberg (2013)
7. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: On the Security of Keyed Sponge Constructions. Presented at SKEW 2011, Lyngby, Denmark, February 16-17 (2011), http://sponge.noekeon.org/SpongeKeyed.pdf
8. Jovanovic, P., Luykx, A., Mennink, B.: Beyond $2^{c/2}$ Security in Sponge-Based Authenticated Encryption Modes. Cryptology ePrint Archive, Report 2014/373 (2014), http://eprint.iacr.org/2014/373
9. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Cryptographic Sponge Functions (2008), http://sponge.noekeon.org/CSF-0.1.pdf
10. Aumasson, J.-P., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New Features of Latin Dances: Analysis of Salsa, ChaCha and Rumba. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 470–488. Springer, Heidelberg (2008)
11. Shi, Z., Zhang, B., Feng, D., Wu, W.: Improved Key Recovery Attacks on Reduced Round Salsa20 and ChaCha. In: Kwon, T., Lee, M.-K., Kwon, D. (eds.) ICISC 2012. LNCS, vol. 7839, pp. 337–351. Springer, Heidelberg (2013)
12. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008)
13. Henzen, L., Carbognani, F., Felber, N., Fichtner, W.: 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, pp. 1–5. IEEE (2008)
14. Gueron, S.: AES-GCM Software Performance on the Current High End CPUs as a Performance Baseline for CAESAR Competition Presented at DIAC 2013, Chicago, USA, August 11-13 (2013), http://2013.diac.cr.yp.to/slides/gueron.pdf.
15. Käsper, E., Schwabe, P.: Faster and Timing-Attack Resistant AES-GCM. Cryptology ePrint Archive, Report 2009/129 (2009), http://eprint.iacr.org/2009/129
16. Jovanovic, P., Neves, S., Aumasson, J.P.: Analysis of NORX. Cryptology ePrint Archive, Report 2014/317 (2014), http://eprint.iacr.org/2014/317
17. Aumasson, J.P., Dinur, I., Henzen, L., Meier, W., Shamir, A.: Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128. Cryptology ePrint Archive, Report 2009/218
18. Dinur, I., Shamir, A.: Cube Attacks on Tweakable Black Box Polynomials. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 278–299. Springer, Heidelberg (2009)
19. Stein, W.: Sage Mathematics Software. The Sage Development Team (2005–2013), http://sagemath.org
20. Ganesh, V., Govostes, R., Phang, K.Y., Soos, M., Schwartz, E.: STP — A Simple Theorem Prover (2006–2013), http://stp.github.io/stp

21. Gorski, M., Lucks, S., Peyrin, T.: Slide Attacks on a Class of Hash Functions. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 143–160. Springer, Heidelberg (2008)
22. Peyrin, T.: Security Analysis of Extended Sponge Functions. In: Presented at the ECRYPT Workshop Hash Functions in Cryptology: Theory and Practice, Leiden, The Netherlands (June 4, 2008), `http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Peyrin.pdf`

A Datagrams

Representations for the datagrams as introduced in Section 2.7.

– Fixed Parameters:

NORX64	header				payload			trailer
field	N	$\|H\|$	$\|T\|$	$\|C\|$	H	C	T	A
offsets [bytes]	0 – 15	16 – 17	18 – 19	20 – 27	28 – ??	?? – ??	?? – ??	?? – ??

NORX32	header				payload			trailer
field	N	$\|H\|$	$\|T\|$	$\|C\|$	H	C	T	A
offsets [bytes]	0 – 7	8 – 9	10 – 11	12 – 15	16 – ??	?? – ??	?? – ??	?? – ??

– Variable Parameters:

NORX64	header						payload			trailer
field	N	$\|H\|$	$\|T\|$	$\|C\|$	$W(1) \parallel R(7)$	D	H	C	T	A
offsets [bytes]	0 – 15	16 – 17	18 – 19	20 – 27	28	29	30 – ??	?? – ??	?? – ??	?? – ??

NORX32	header						payload			trailer
field	N	$\|H\|$	$\|T\|$	$\|C\|$	$W(1) \parallel R(7)$	D	H	C	T	A
offsets [bytes]	0 – 7	8 – 9	10 – 11	12 – 15	16	17	18 – ??	?? – ??	?? – ??	?? – ??

B Test Vectors

Test vectors for some instances of NORX are given below. More can be found on the official website [5].

– NORX64:

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

NORX64-4-1

```
C : 1B4DCCFF6779A2C3 865464C856BC4B0C DADBC58565E1690A 2CB12C0BE9D2F045
A : D0CE5276FDEC9F6E 33EE64CE5CCA3ABA 1187C05183464BD0 A0915ECA6FAF8757
```

NORX64-6-1

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

– NORX32:

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

NORX32-4-1

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

NORX32-6-1

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