Pipeline Oriented Implementation of NORX for ARM Processors

Luan Cardoso dos Santos[1], **Julio López**[1]

[1]Laboratory of Security and Cryptography - LASCA
Institute of Computation - UNICAMP
Av. Albert Einstein, 1251, Room 84
Cidade Universitária Zeferino Vaz
13083-852 Campinas SP Brazil

luan@lasca.ic.unicamp.br, jlopez@ic.unicamp.br

Abstract. *NORX is a family of authenticated encryption algorithms that advanced to the third-round of the ongoing CAESAR competition for authenticated encryption schemes. In this work, we investigate the use of pipeline optimizations on ARM platforms to accelerate the execution of NORX. We also provide benchmarks of our implementation using NEON instructions. The results of our implementation show a speed improvement up to 48% compared to the state-of-art implementation on Cortex-A ARMv8 and ARMv7 processors.*

Resumo. *NORX é uma família de algoritmos de cifração autenticada que participa da terceira fase do CAESAR, competição para esquemas de cifração autenticada. Nesse trabalho, investigamos o uso de optimizações de* pipeline *em plataformas ARM de forma a acelerar a execução do NORX. Também mostramos tempos da nossa implementação usando instruções NEON. Nossos resultados mostram melhoria de até 48% na velocidade de execução comparado com implementações estado-da-arte em processadores Cortex A ARMv8 e em processadores ARMv7.*

1. Introduction

Authenticated Encryption algorithms (AE) are symmetric-key cryptographic schemes where the main objective is to provide simultaneously confidentiality, integrity, and authentication. In an intuitive form, confidentiality means that an adversary with access to the ciphertext and nonce cannot recover any information of the plaintext beyond its length, and the ciphertext itself is indistinguishable from random bits. Similarly, authenticity guarantees that a ciphertext cannot be manipulated to generate a valid authentication for any given message.

NORX [1] is an authenticated encryption scheme currently participating in the CAESAR [2] competition, which has the objective of choosing an authenticated encryption algorithms that offer advantages over the AES-GCM algorithm [3], as candidates for a future standard in authenticated encryption.

We show software optimization techniques for the NORX family of algorithms, targeting ARM processors. We choose ARM processors due to their widespread usage in consumer electronics, such as IOT devices, smartphones, embedded devices, and gadgets.

In this section, we will provide some background on the concepts used throughout the paper regarding AE, cryptographic competitions, and sponge functions. In Section 2, we will give a brief description of the NORX algorithm, and in Section 3 we will show the main characteristics of the target architecture. Section 4 will describe our implementation techniques and in Section 5, we will present benchmarks and discuss the results.A final conclusion is given in Section 6.

1.1. AEAD Algorithms

An authenticated encryption scheme is an algorithm that uses a secret key and a public nonce to process a plaintext and generate a ciphertext and an authentication tag. Furthermore, an AE scheme can also receive extra data that is authenticated together with the plaintext. In that mode of operation, this scheme is called Authenticated Encryption with Additional Data (AEAD). Such a scheme is useful, for example, to encrypt the body of a message, while keeping the receiving address in plain form, and authenticating the whole. This way, the recipient of a message can guarantee that public data was not modified by a third party. A basic block diagram of an authenticated encryption algorithm is shown in Figure 1.

Formally an AEAD scheme is defined by the tuple $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ and the associated sets $\mathrm{Nonce} = \{0,1\}^n$, $\mathrm{Header} \subset \{0,1\}^*$ and $\mathrm{Message} \subseteq \{0,1\}^*$. The Message set must satisfy the membership test $M \in \mathrm{Message} \Rightarrow M' \in \mathrm{Message}$ for any M' with the same length of M. The keyspace $\mathcal{K}$ is a non-empty finite set of strings. The encryption algorithm $\mathcal{E}$ is a deterministic algorithm that receives as input strings $K \in \mathcal{K}$, $N \in \mathrm{Nonce}$, $H \in \mathrm{Header}$ and $M \in \mathrm{Message}$. The encryption algorithm returns a string $\mathcal{C} = \mathcal{E}_K^{N,H}(M) = \mathcal{E}_K(N, H, M)$. The decryption algorithm $\mathcal{D}$ is a deterministic algorithm that receives as input the strings $K \in \mathcal{K}$, $N \in \mathrm{Nonce}$, $H \in \mathrm{Header}$ and $\mathcal{C} \in \{0,1\}^*$ and returns $\mathcal{D}_K^{N,H}(\mathcal{C}) = \mathcal{D}_K(N, H, C)$, that is either a string from the set of possible messages, or a symbol $\perp$ meaning that the set of ciphertext, nounce and key is invalid. Beyond that, it is required that $\mathcal{D}_K^N(\mathcal{E}_K^N(M)) = M$ for all $K \in \mathcal{K}$, $N \in \mathrm{Nounce}$ and $M \in \mathrm{Message}$, and that $|\mathcal{E}_K^{N,H}| = l(|M|)$ for some linear-time length function l [4].

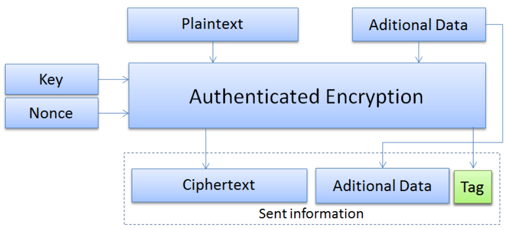


Figure 1. Basic block design of an AEAD, where ciphertext and authentication tag are produced by processing plaintext, additional data, key, and nonce.

1.2. The CAESAR competition

The algorithm considered in this paper is a competitor of the third round of CAESAR –Competition for Authenticated Encryption: Security, Applicability, and Robustness. Following the footsteps of previous competitions, such as the AES, eSTREAM, and SHA, CAESAR aims to select a portfolio of authenticated ciphers that offer advantages over

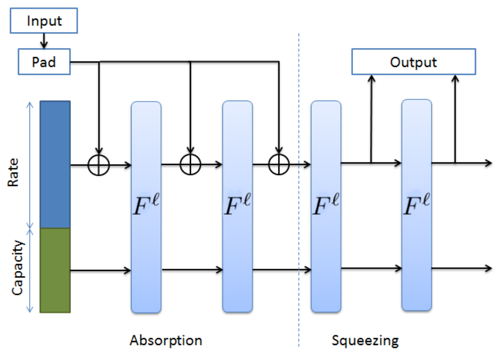


Figure 2. The basic design of a sponge function, showing the absorption and squeeze processes [7].

NIST's AES-GCM and that are suitable for widespread adoption. The AES competition is regarded as one of the responsible for promoting an improvement in the scientific knowledge about block ciphers. Similarly, eSTREAM [5] and SHA-3 [6] promoted research in the areas of stream ciphers and hash functions, and it is expected that the CAESAR competition brings the same impact in the research area of authenticated ciphers [2].

1.3. Sponge function

A cryptographic sponge function, introduced as a primitive for authenticated encryption in [7] and as a general cryptographic function in [8][9], is an algorithm with a finite internal state that receives as input a string of any length and produces as output a string of any desired length. Sponge functions can be used to create various cryptographic primitives, such as hash functions, MACs, stream ciphers, pseudorandom number generators and authenticated encryption schemes. A sponge function can be imagined as a real-world sponge, where data is absorbed and then squeezed from it.

A sponge is based on three main components: A state S of b bits, subdivided into *rate* and *capacity* sections of respectively r and c bits; a round permutation function F^l of b bits with a round number l defined in terms of a permutation F of b bits as the l-fold iteration $F^l(S) = F(F(...F(S)))$ which is used to transform the state in each round; and a padding rule P for the input. A sponge works by initializing the state value and "absorbing" r bits from the padded input and transforming the state with $F^l(S)$. After that, the sponge is ready to be "squeezed", removing up to r bits before needing to evaluate $F^l(S)$ again. Figure 2 illustrates the operation of a sponge [7]. An example of a practical use of sponge functions in cryptographic primitives is the SHA-3[6] hash algorithm, that uses a 1600-bit sponge.

2. The NORX AEAD family of algorithms

NORX is an AEAD scheme created by Jean-Philippe Aumasson, Philipp Jovanovic and Samuel Neves [10], supporting associated data in the form of both headers and trailers. NORX also supports arbitrary parallelism and is optimized for efficient hardware and software implementations, with a SIMD-friendly construction, no secret array indexing, and only bitwise operations. ARX primitives are thoroughly used, without modular additions, and is based on the monkey-duplex construction. NORX's core permutation

Table 1. The five instances of NORX

Instance name	w	l	p	t	k	n
NORX64-4-1	64	4	1	256	256	128
NORX32-4-1	32	4	1	128	128	64
NORX64-6-1	64	6	1	256	256	128
NORX32-6-1	32	6	1	128	128	64
NORX64-4-4	64	4	4	256	256	128

function is based on ChaCha's permutation [11], with the integer addition $(a + b)$ replaced by the approximation[1] $a \oplus b \oplus (a \wedge b) \ll 1$, which in turn –according to the design team of NORX– simplifies cryptanalysis and improves hardware efficiency [10].

The NORX family of algorithms is parametrized by the word size in bits w; a round number ℓ with $1 \leq \ell \leq 63$; a parallelism degree p with $0 \leq p \leq 255$ (where $p = 0$ defines arbitrary parallelism) and a tag t. Regarding the key length, NORX32 uses a 128-bit key, NORX64 a 256-bit key, while NORX16 and NORX08 use a 96-bit and an 80-bit key respectively. The 32 and 64-bit versions of NORX also use an $n = 2w$ bits nonce; the 8-bit and 16-bit variants have a nonce of $n = 32$ bits. On the CAESAR submission, Aumasson et al. [10] propose five instances of NORX for different uses cases. They are listed in Table 1, from the highest recommendation at the top to the lowest. The naming convention for a specific instance of the algorithm is NORXw-l-p-t, with w, l, p and t being the instance parameters. When the tag length is the default $t = 4w$, then the notation is shortened as NORXw-l-p.

NORX parametrized with $w = 32$ bits is adequate for lightweight applications and resource-constrained environments, requiring small hardware area and small ROM size for software implementations. On the other hand, the instances with $w = 64$ bits are adequate for high-performance and high-security applications, being efficient in both 64-bit and 32-bit CPUs[2]. Requirements for ASIC implementations are about 64 kGE, and at most 64 bytes of ROM for the initialization constants. It is also possible to implement NORX using only one byte plus the sponge size of data in RAM [10].

In the following sections, we will show in more details the specification of NORX, as given in [10].

2.1. NORX's mode of operation

NORX follows a duplexed sponge layout, as shown in Figure 3. NORX's construction allows parallel processing of the payload, defined by p. For serial processing, with $p = 1$, the layout of NORX is that of a standard duplexed sponge. For a value $p > 1$, the number of parallel processing lanes is given by the value of p; for example, Figure 3 illustrates the case of $p = 2$. For $p = 0$, the number of processing lanes is bounded by the size of the payload itself, making the layout of NORX similar to that of the PPAE construction [12].

[1]This approximation is derived from the identity $a + b = (a \oplus b) + (a \wedge b) \ll 1$.

[2]A draft of these use-cases can be found in the CAESAR mailing list at the address `https://groups.google.com/forum/#!topic/crypto-competitions/DLv193SPSDc`.

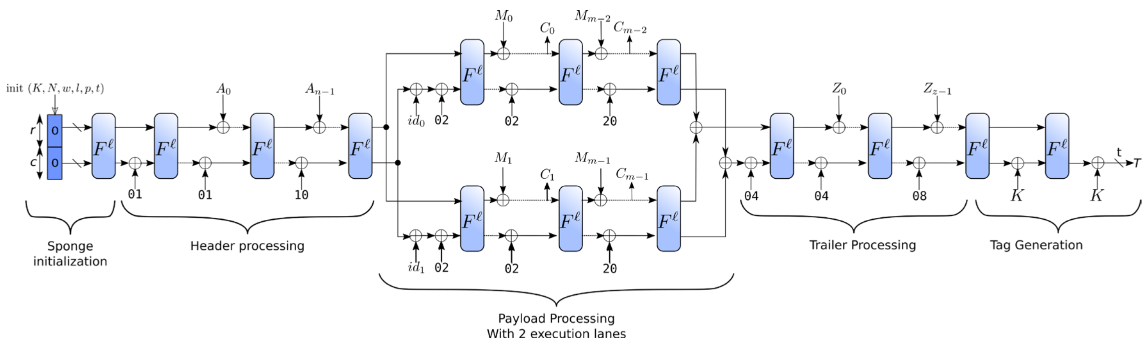


Figure 3. The layout of NORX with parallelism degree $p = 2$. Notice that the sponge is divided into multiple execution lanes in the payload processing step. Those lanes can be computed in parallel, as there is no data dependency amongst them. When $p = 1$ a single lane is executed, making the payload processing similar to header and trailer processing. Based on a figure from [10].

2.2. NORX permutation function

NORX's core is the permutation function $F^\ell()$, applied to the NORX internal state S, with ℓ being the number of rounds. The state is a concatenation of 16 w-bit words in the form $S = s_0 \parallel \cdots \parallel s_{15}$, where the words $s_0, \cdots, s_{11}$ are called the *rate words*, where data is injected and extracted from, and the remaining words $s_{12}, \cdots, s_{15}$ are called *capacity words*. Conceptually, the state can be viewed as a 4×4 matrix:

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

A single permutation $F()$ processes the state S by applying the G function to the matrix's columns and then diagonals. The G function is described in Algorithm 1, and the permutation F is specified in Algorithm 2.

Algorithm 1 NORX G permutation function

```
1:  Function G
2:      input: $a, b, c, d$                                ▷ Four words of the State
3:      $a \leftarrow (a \oplus b) \oplus ((a \wedge b) \lll 1)$
4:      $d \leftarrow (a \oplus d) \ggg r_0$
5:      $c \leftarrow (c \oplus d) \oplus ((c \wedge d) \lll 1)$
6:      $b \leftarrow (c \oplus b) \ggg r_1$
7:      $a \leftarrow (a \oplus b) \oplus ((a \wedge b) \lll 1)$
8:      $d \leftarrow (a \oplus d) \ggg r_2$
9:      $c \leftarrow (c \oplus d) \oplus ((c \wedge d) \lll 1)$
10:     $b \leftarrow (c \oplus b) \ggg r_3$
11:     output: $a, b, c, d$
12: end Function
```

NORX's encryption and decryption primitives can be described by Algorithm 3 and 4, where `header`, `branch`, `payload`, `merge`, `trailer` and `tag` are domain

Algorithm 2 NORX F round function

```
1: **Function** $F$
2:     **input:** $S, G()$                                ▷ Norx State $s_0 \cdots s_{15}$ and $G()$ function
3:     /* Processing the columns */
4:     $s_0, s_4, s_8, s_{12} \leftarrow G(s_0, s_4, s_8, s_{12})$
5:     $s_1, s_5, s_9, s_{13} \leftarrow G(s_1, s_5, s_9, s_{13})$
6:     $s_2, s_6, s_{10}, s_{14} \leftarrow G(s_2, s_6, s_{10}, s_{14})$
7:     $s_3, s_7, s_{11}, s_{15} \leftarrow G(s_3, s_7, s_{11}, s_{15})$
8:     /* Processing the diagonals */
9:     $s_0, s_5, s_{10}, s_{15} \leftarrow G(s_0, s_5, s_{10}, s_{15})$
10:    $s_1, s_6, s_{11}, s_{12} \leftarrow G(s_1, s_6, s_{11}, s_{12})$
11:    $s_2, s_7, s_8, s_{13} \leftarrow G(s_2, s_7, s_8, s_{13})$
12:    $s_3, s_4, s_9, s_{14} \leftarrow G(s_3, s_4, s_9, s_{14})$
13:    **output:** $S$
14: **end Function**
```

separation constants; K is the key, N is the nonce, A is the additional data to be processed before the plaintext, `Msg` is the plaintext, Z is the additional data to be processed after the plaintext, `Tag` is the authentication tag, and `Cipher` is the ciphertext. For more details on the algorithm's description, the reader is invited to read chapter 2 of [10].

Algorithm 3 NORX AEAD encryption

```
1: **Function** ENCRYPT($K, N, A, Msg, Z$)
2:     $S \leftarrow$ initialise($K, N$)
3:     $S \leftarrow$ absorb($S, A$, header )
4:     $\bar{S} \leftarrow$ branch($S, |Msg|$, branch )
5:     $\bar{S}, Cipher \leftarrow$ encrypt($\bar{S}, M$, payload )
6:     $S \leftarrow$ merge($\bar{S}, |M|$, merge )
7:     $S \leftarrow$ absorb($S, Z$, trailer )
8:     $S, Tag \leftarrow$ finalise($S$, tag )
9:     **return** $Cipher, Tag$
10: **end Function**
```

3. Platforms - ARM processors

In this section, we will briefly describe the target architecture of this work. The ARM –Advanced RISC Machine– architecture is a mainly 32-bit architecture owned by the British company ARM Holdings. The ARM architecture was introduced in 1985, and with more than 86 billion chips produced up to 2016, it has a big share of the consumer and embedded processor market [13].

The 32-bit ARM processors feature a Load/Store architecture without support for unaligned memory accesses, uniform 16×32-bit registers, and mostly a single clock cycle execution. Beyond that, the processors also feature conditional execution for most instructions and a 32-bit barrel shifter that can be used without affecting the performance with most arithmetic instructions and addresses calculations. The architecture can be divided into three main family lines: Cortex-M, Cortex-A, and Cortex-R. The Cortex-M

Algorithm 4 NORX AEAD Decryption

```
1: Function DECRYPT($K, N, A, Cipher, Z, T$)
2:     $S \leftarrow$ initialise($K, N$)
3:     $S \leftarrow$ absorb($S, A,$ header )
4:     $\bar{S} \leftarrow$ branch($S, |C|,$ branch )
5:     $\bar{S}, Msg \leftarrow$ decrypt($\bar{S}, C,$ payload )
6:     $S \leftarrow$ merge($\bar{S}, |C|,$ merge )
7:     $S \leftarrow$ absorb($S, Z,$ trailer )
8:     $S, Tag' \leftarrow$ finalise($S,$ tag )
9:     if $Tag' == T$ then return $Msg, Tag$
10:    else
11:        return $\perp$                                ▷ Symbol for failed decryption
12:    end if
13: end Function
```

cores are the simplest ones, with a focus on embedded systems with a low footprint and low energy requirements. The Cortex-A cores are more powerful, with the focus on power efficiency and they are deployed in a wide range of products. Lastly, the Cortex-R cores are suitable for high-performance real-time systems, where a high reliability is needed [14].

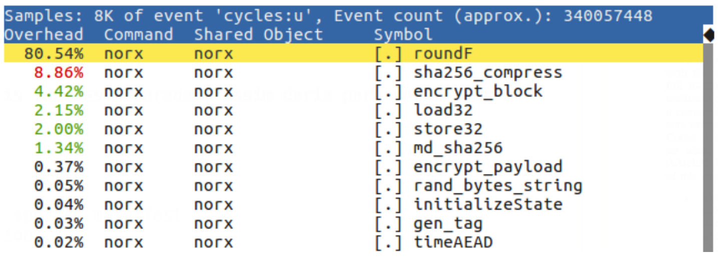
In this paper, the software implementations focus on the Cortex-A cores, namely Cortex-A7, A15, and A53, mainly for their large use in consumer electronics such as smartphones and tablets. Benchmarks of the implementations were also carried on the embedded processors containing Cortex-M4, M3, and M0 cores, as a way to evaluate the optimization impact on simpler processors.

The main characteristics of the target cores of this work are as follows [14]:

- Cortex-A7: Currently the most power efficient ARMv7-A core, with over a billion shipped units in production. The processor is capable of 40-bit physical addressing and has an eight-stage in-order pipeline. The A7 core is compatible with higher performance cores such as the Cortex-A15 and A17 for use with the big.LITTLE technology, where high-performance cores are combined with highly efficient cores in a heterogeneous computation approach.
- Cortex-A15: A high-performance ARMv7-A core, well suited to consumer items such as smartphones and embedded applications. As with other processors of the same line, it is capable of 40-bit physical addressing. It also features a 15 stage pipeline for integer calculations.
- Cortex-A53: An ARMv8-A core capable of seamlessly running both 32-bit and 64-bit code, and is made as an efficient 64-bit core for a low area and power footprint. Like the Cortex-A7, it is capable of being deployed together with high-end CPUs for chips with heterogeneous cores. The Cortex-A53 uses an efficient eight-stage in-order pipeline.

4. Implementation and Techniques

In the next sections, we discuss the optimization techniques applied to NORX, in order to obtain better performance in comparison to the state-of-art implementation [15]. Profiling the code to identify hotspots of interest to optimize was the first step in this work. For that, the Linux tool `perf` was used to analyze the code. Results are shown in Figure 4. Notice that the call for `sha256_compress`, responsible for 8.86% of overhead, is not a part of NORX. Instead, it is used to generate pseudorandom inputs for the algorithm benchmark, and should not be considered into optimization efforts.



```
Samples: 8K of event 'cycles:u', Event count (approx.): 340057448
Overhead  Command  Shared Object       Symbol
  80.54%  norx     norx                [.] roundF
   8.86%  norx     norx                [.] sha256_compress
   4.42%  norx     norx                [.] encrypt_block
   2.15%  norx     norx                [.] load32
   2.00%  norx     norx                [.] store32
   1.34%  norx     norx                [.] md_sha256
   0.37%  norx     norx                [.] encrypt_payload
   0.05%  norx     norx                [.] rand_bytes_string
   0.04%  norx     norx                [.] initializeState
   0.03%  norx     norx                [.] gen_tag
   0.02%  norx     norx                [.] timeAEAD
```

Figure 4. The result of the analysis of NORX, showing that the round function is responsible for about 70% of overhead. Tests carried out on an Odriod XU4 device, with a Cortex-A15 core.

4.1. Improving the use of the processor's pipeline

The function G is executed on the columns and then on the diagonals of the 4×4 matrix representation of NORX State, using Algorithm 5 for each column and diagonal. Since there is no dependency between each column and diagonal, the function G can be rewritten in a way that each step of G is executed right after the other, for each column or diagonal. This way, the execution can be arranged in groups of two columns or four columns. This approach allows a better use of the pipeline, improving the execution performance. Figure 5 illustrates the idea behind both approaches to optimize the function G.

Algorithm 5 NORX original implementation of G function

```
Function G
    input: $a, b, c, d$                                   ▷ Four words of the State
    $a = (a \oplus b) \oplus ((a \wedge b) \ll 1)$
    $d = d \oplus a$
    $d = \mathtt{ROR}(d, \mathtt{r0})$
    $c = (c \oplus d) \oplus ((c \wedge d) \ll 1)$
    $b = b \oplus c$
    $b = \mathtt{ROR}(b, \mathtt{r1})$
    $a = (a \oplus b) \oplus ((a \wedge b) \ll 1)$
    $d = d \oplus a$
    $d = \mathtt{ROR}(d, \mathtt{r2})$
    $c = (c \oplus d) \oplus ((c \wedge d) \ll 1)$
    $b = b \oplus c$
    $b = \mathtt{ROR}(b, \mathtt{r3})$
    output: $a, b, c, d$
end Function
Function F
    input: $S, \mathtt{G}()$                ▷ Whole Norx State and permutation function.
    $(s_0, s_4, s_8, s_{12}) \leftarrow \mathtt{G}(s_0, s_4, s_8, s_{12})$                ▷ column step
    $(s_1, s_5, s_9, s_{13}) \leftarrow \mathtt{G}(s_1, s_5, s_9, s_{13})$
```

```
21:     $(s_2, s_6, s_{10}, s_{14}) \leftarrow \mathtt{G}(s_2, s_6, s_{10}, s_{14})$
22:     $(s_3, s_7, s_{11}, s_{15}) \leftarrow \mathtt{G}(s_3, s_7, s_{11}, s_{15})$
23:     $(s_0, s_5, s_{10}, s_{15}) \leftarrow \mathtt{G}(s_0, s_5, s_{10}, s_{15})$        ▷ diagonal step
24:     $(s_1, s_6, s_{11}, s_{12}) \leftarrow \mathtt{G}(s_1, s_6, s_{11}, s_{12})$
25:     $(s_2, s_7, s_8, s_{13}) \leftarrow \mathtt{G}(s_2, s_7, s_8, s_{13})$
26:     $(s_3, s_4, s_9, s_{14}) \leftarrow \mathtt{G}(s_3, s_4, s_9, s_{14})$
27:     output: $S$
28: end Function
```

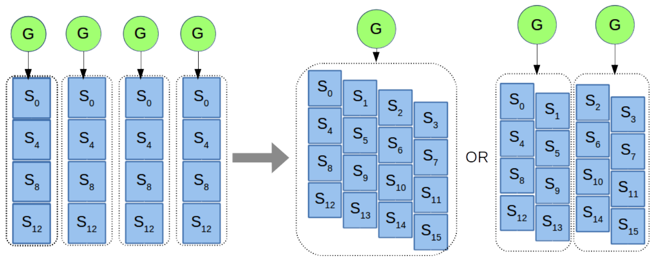


Figure 5. Illustration of two possible ways to reinterpret the G function; on the left, the original one. On the center, the 4-way pipeline, and on the right, the two-way pipeline. Notice that it is only shown the first column transformation.

In this way, the optimized code of function G for a 4-way pipeline is described in Algorithm 6 and the same function, optimized for a 2-way pipeline is described in Algorithm 7. In those algorithms `ROR(a, r)` is the bitwise right rotation of `a` by `r` bits; `ROR({a, b, c, d}, r)` is each word in the tuple `{a,b,c,d}` rotated by `r` bits and `r1`, `r2`, `r3`, `r4` are NORX rotation constants. Notice that, in order to execute a complete $F()$ function, $G4()$ must be called twice, with a different argument order in the second call to execute the diagonal step. Similarly, $G2()$ must be called a total of four times in order to execute the column and diagonal steps.

Algorithm 6 The function G with 4-way pipeline optimization

```
1: Function G4
2:     input: $S$                                                ▷ Whole Norx State
3:     $s_0 = (s_0 \oplus s_4) \oplus ((s_0 \wedge s_4) \ll 1)$
4:     $s_1 = (s_1 \oplus s_5) \oplus ((s_1 \wedge s_5) \ll 1)$
5:     $s_2 = (s_2 \oplus s_6) \oplus ((s_2 \wedge s_6) \ll 1)$
6:     $s_3 = (s_3 \oplus s_7) \oplus ((s_3 \wedge s_7) \ll 1)$
7:     $s_{12} = s_{12} \oplus s_0$ ; $s_{13} = s_{13} \oplus s_1$
8:     $s_{14} = s_{14} \oplus s_2$ ; $s_{15} = s_{15} \oplus s_3$
9:     $\{s_{12}, s_{13}, s_{14}, s_{15}\} = \mathtt{ROR}(\{s_{12}, s_{13}, s_{14}, s_{15}\}, \mathtt{r0})$
10:    $s_8 = (s_8 \oplus s_{12}) \oplus ((s_8 \wedge s_{12}) \ll 1)$
11:    $s_9 = (s_9 \oplus s_{13}) \oplus ((s_9 \wedge s_{13}) \ll 1)$
12:    $s_{10} = (s_{10} \oplus s_{14}) \oplus ((s_{10} \wedge s_{14}) \ll 1)$
13:    $s_{11} = (s_{11} \oplus s_{15}) \oplus ((s_{11} \wedge s_{15}) \ll 1)$
14:    $s_{12} = s_{12} \oplus s_0$ ; $s_{13} = s_{13} \oplus s_1$
15:    $s_{14} = s_{14} \oplus s_2$ ; $s_{15} = s_{15} \oplus s_3$
16:    $\{s_4, s_5, s_6, s_7\} = \mathtt{ROR}(\{s_4, s_5, s_6, s_7\}, \mathtt{r1})$
17:    $s_0 = (s_0 \oplus s_4) \oplus ((s_0 \wedge s_4) \ll 1)$
```

```
18:     $s_1 = (s_1 \oplus s_5) \oplus ((s_1 \wedge s_5) \ll 1)$
19:     $s_2 = (s_2 \oplus s_6) \oplus ((s_2 \wedge s_6) \ll 1)$
20:     $s_3 = (s_3 \oplus s_7) \oplus ((s_3 \wedge s_7) \ll 1)$
21:     $s_{12} = s_{12} \oplus s_0$ ; $s_{13} = s_{13} \oplus s_1$
22:     $s_{14} = s_{14} \oplus s_2$ ; $s_{15} = s_{15} \oplus s_3$
23:     $\{s_{12}, s_{13}, s_{14}, s_{15}\} = \texttt{ROR}(\{s_{12}, s_{13}, s_{14}, s_{15}\}, \texttt{r2})$
24:     $s_8 = (s_8 \oplus s_{12}) \oplus ((s_8 \wedge s_{12}) \ll 1)$
25:     $s_9 = (s_9 \oplus s_{13}) \oplus ((s_9 \wedge s_{13}) \ll 1)$
26:     $s_{10} = (s_{10} \oplus s_{14}) \oplus ((s_{10} \wedge s_{14}) \ll 1)$
27:     $s_{11} = (s_{11} \oplus s_{15}) \oplus ((s_{11} \wedge s_{15}) \ll 1)$
28:     $s_{12} = s_{12} \oplus s_0$ ; $s_{13} = s_{13} \oplus s_1$
29:     $s_{14} = s_{14} \oplus s_2$ ; $s_{15} = s_{15} \oplus s_3$
30:     $\{s_4, s_5, s_6, s_7\} = \texttt{ROR}(\{s_4, s_5, s_6, s_7\}, \texttt{r3})$
31:     output: $S$
32: end Function
```

Algorithm 7 The function G with 2-way pipeline optimization

```
 1: Function GH
 2:     input: $s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13}$
 3:                                   ▷ Either two columns or diagonals of State.
 4:     $s_0 = (s_0 \oplus s_4) \oplus ((s_0 \wedge s_4) \ll 1)$
 5:     $s_1 = (s_1 \oplus s_5) \oplus ((s_1 \wedge s_5) \ll 1)$
 6:     $s_{12} = s_{12} \oplus s_0$
 7:     $s_{13} = s_{13} \oplus s_1$
 8:     $s_{12} = \texttt{ROR}(s_{12}, \texttt{r0})$
 9:     $s_{13} = \texttt{ROR}(s_{13}, \texttt{r0})$
10:     $s_8 = (s_8 \oplus s_{12}) \oplus ((s_8 \wedge s_{12}) \ll 1)$
11:     $s_9 = (s_9 \oplus s_{13}) \oplus ((s_9 \wedge s_{13}) \ll 1)$
12:     $s_4 = s_4 \oplus s_8$
13:     $s_5 = s_5 \oplus s_9$
14:     $s_4 = \texttt{ROR}(s_4, \texttt{r1})$
15:     $s_5 = \texttt{ROR}(s_5, \texttt{r1})$
16:     $s_0 = (s_0 \oplus s_4) \oplus ((s_0 \wedge s_4) \ll 1)$
17:     $s_1 = (s_1 \oplus s_5) \oplus ((s_1 \wedge s_5) \ll 1)$
18:     $s_{12} = s_{12} \oplus s_0$
19:     $s_{13} = s_{13} \oplus s_1$
20:     $s_{12} = \texttt{ROR}(s_{12}, \texttt{r2})$
21:     $s_{13} = \texttt{ROR}(s_{13}, \texttt{r2})$
22:     $s_8 = (s_8 \oplus s_{12}) \oplus ((s_8 \wedge s_{12}) \ll 1)$
23:     $s_9 = (s_9 \oplus s_{13}) \oplus ((s_9 \wedge s_{13}) \ll 1)$
24:     $s_4 = s_4 \oplus s_8$
25:     $s_5 = s_5 \oplus s_9$
26:     $s_4 = \texttt{ROR}(s_4, \texttt{r3})$
27:     $s_5 = \texttt{ROR}(s_5, \texttt{r3})$
28:     output: $s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13}$
29: end Function
30: Function G2
```

```
31:     input: $S, GH()$                    ▷ Whole Norx State and the 2-col permutation
32:     $(s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13}) \leftarrow GH(s_0, s_1, s_4, s_5, s_8, s_9, s_{12}, s_{13})$
33:     $(s_2, s_3, s_6, s_7, s_{10}, s_{11}, s_{14}, s_{15}) \leftarrow GH(s_2, s_3, s_6, s_7, s_{10}, s_{11}, s_{14}, s_{15})$
34:     output: $S$
35: end Function
```

In algorithm 7, the even numbered lines execute $G()$ on a line or column of the internal state, while the odd numbered lines execute the same instruction on a independent line or column of the state. This allows issue of independent `xor` instructions, such as in lines 6 and 7. Similarly, the same will happen with `and` and `lsh` instructions. The same idea is applied further on 6, but instead of only issuing instructions for two independent sets of state words, the whole state is operated at once. This allows the code to issue most of instructions in sets of four, without dependencies amongst them. This is specially useful for processors with a deep pipeline.

Regarding the security of those optimizations, we used Flowtracker to analyze the behavior of the algorithm, and the code runs in constant time. Beyond that, NORX is resistant against side-channel attacks by design, with cryptanalysis regarding differential, algebraic, fixed-point, slide, and rotational attacks, [10]. Furthermore, there are no table-lookups, no branching, or loops dependent on secret data. For implementation correctness, we compared our outputs with the reference algorithm, using the same set of input data. We verified internal consistency using encryption-decryption of random sets of plaintext, nonce, and keys.

4.2. Code improvements

A few minor improvements were also applied on the code. The ones with a positive impact on the code performance were:

- Extensive use of preprocessor macros and function inlining, which avoids overhead while still keeping code readability.
- Avoid the use of temporary variables whenever possible, doing most of the encryption, decryption, and additional data processing in place.
- Using a prefix operation instead of a postfix one on loop counters yields small improvements, more visible on Cortex-M based processors.
- Initialize the sponge using constants instead of calculating it as $F^2(0 \parallel 1 \parallel 2 \parallel \cdots \parallel 15)$, where each number j is represented as using w bits.
- Where possible, concatenate shift and rotate operations together with arithmetic operations, as to allow the use of the target processor's barrel shifter, making the shift operation free.

Other approaches were tested, such as replacing `memcpy()` calls with loops, manually unrolling loops and changing memory alignment. Those did not impact the performance in any significant way, resulting in negligible variations in cycle count.

5. Benchmarks and results

The benchmarks were carried out on an Odroid XU4 device running Arch Linux for the Cortex-A7 and Cortex-A15 cores, and on an Odroid-C2 device for the Cortex-A53, running the same OS. The code was compiled using GCC 6.3.1. Each test consists of the encryption of random data from `/dev/urandom` with lengths between 128 bytes to 1 megabyte in powers of two increments. Tests were also carried on an Arduino Zero, with a Cortex-M0 core; an Arduino Due with a Cortex-M3 core and a Teensy 3.2 with a Cortex-M4 core. On the Cortex M4, M3, and M0 devices, the codes were compiled using `arm-none-eabi-gcc 4.8.3`. The compilation flags used were: `-O3 -Wall -Wextra -std=c99 -fno-schedule-insns -fomit-frame-pointer -Wno-old-style-declaration -funroll-loops -fpeel-loops` We consider that the cycle counter on the target processors show consistent values, and it is adequate to compare our implementations with the reference ones. For ARM-v7-A architecture, the following inline assembler code is used to enable the access to performance counter, and return the value of the performance registers which can be used to measure elapsed clock cycles between calls of the instruction `asm("mrc p15, 0, %0, c9, c13, 0" : "=r"(value));`. For AArch64 compatible processors, the following code is used: `asm ("mrs %0, pmccntr_el0" : "=r" (r));` .

Lastly, these calls are done before and after the call to the encryption or decryption function, and with the difference between the two measures, the average cycle per byte is measured, and the median of multiple measurements is reported in this work. This methodology is similar to the one used by SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives)[16].

Table 2. Cycles per byte for NORX encryption. Plaintext length of 256KiB on the 32bit processors. The best result for each plataform and cypher is in bold type.

		Ref. code	4x pipe	2x pipe	Speedup
NORX 3261	Cortex A7	29.45	29.70	**24.72**	16%
	Cortex A15	17.77	**14.23**	15.16	20%
NORX 6461	Cortex A7	48.52	50.09	**46.65**	4%
	Cortex A15	33.83	**26.76**	28.33	21%
NORX 3264	Cortex A7	28.46	33.74	**26.50**	7%
	Cortex A15	16.88	**15.26**	15.37	10%

Table 3. Cycles per byte for NORX encryption on the 64-bit platform. Plaint text length of 256KiB. The best result for each plataform and cypher is in bold type.

	Ref.	4x pipe	2x pipe	Speedup
NORX 3261	19.55	**10.94**	12.27	44%
NORX 6461	10.29	**5.84**	6.58	43%
NORX 3264	19.42	**12.08**	13.06	38%

In Table 2, we show the results for the 32-bit processors, namely Cortex A7 and A15; in Table 3 we show the results for the 64-bit Cortex A53 processor; lastly, in Table 4, we show the results on Cortex-M processors. We choose to show the average CBP –cycle per byte– with an input length of 256KiB since it better dilutes the overhead values, without risking overflow on the cycle counting registers, except in the Cortex-M based processors, due to memory constraints.

For the 32-bit variant of NORX, a $4\times$ pipeline implementation is faster than the reference code in up to 20% on a 32-bit ARM and 44% on the 64-bit Cortex-A53. Our optimized implementation is faster than the reference NEON implementation: the $2\times$ pipeline implementation is 12% faster than the reference code on the Cortex A7 core; the

Table 4. Perfomance of NORX3261 (cycles per byte) on 32-bit Cortex-M architecture.

Cortex model	Size	No pipeline optimizations	Ref. code	4x pipe	2x pipe
M0	8KiB	99.52	100.12	111.84	99.96
M3	32KiB	49.96	50.49	67.21	66.26
M4	16KiB	49.96	50.49	47.28	66.26

$4\times$ pipeline implementation is 22% faster on the Cortex A15. While NORX has a SIMD friendly construction, with the internal state fitting in four 128-bit NEON registers, there are two extra transformations needed in each application of the function G in order to align the words between the column and diagonal steps. This transformation requires three extra pairs of SIMD load and store instructions, two `vext.8` instructions, and a `vwsp` instruction. We believe that this, together with the extra cost needed to transfer data from the NEON registers back to the ARM registers every round, coupled with the optimal usage of the pipeline makes our solution better than using SIMD instructions for these cores.

For the 64-bit variant of NORX, a $2\times$ pipeline is better suited for the Cortex-A7 processor, and a $4\times$ pipeline for the Cortex-A15 processor, due to the differences in pipeline length. With SIMD instructions being adequate for larger volume of data, NORX6461 on the 32-bit platform shows better performance using SIMD instructions, mainly due to the 64-bit word rotations being expensive using the 32-bit ARM registers, in comparison to the neon approach, where the rotations of two words can be done at the same time in the 128-bit register. For Cortex A53, both pipeline implementations show satisfactory results, being 43% faster than the reference code. In relation to a NEON implementation, the $4\times$ pipeline implementation is 39% faster and the $2\times$ pipeline implementation is 31% faster. Similar to NORX3261, the presence of a native 64-bit register and a deep pipeline with 8 stages makes a pipeline oriented approach superior to the SIMD alternative.

The multisponge approach, NORX3264, shows similar behavior to that of the 32-bit single-sponge algorithm. Table 2 and Table 3 show the execution times for a single thread implementation running on a single core. The multisponge algorithm is better suited for a multithread implementation, with each thread being responsible for a instance of the internal state. Our tests show that such an approach can result in up to 76% speedup, in relation to the single thread implementation.

6. Conclusions

This work shows how a pipeline oriented optimization can yield significant performance improvements on an authenticated encryption algorithm, specifically NORX, outperforming NEON vectorial code in some situations, while at the same time using only portable C code. These optimizations also result in little to no performance penalties on smaller Cortex-M cores. We believe that these techniques can also be applied to other algorithms that use similar constructions, resulting in a better performance with little drawbacks.

7. Aknowledgements

This work was supported by LG Electronics via the ”Crypto for IOT” project. The second author was partially supported by a research productivity grant from CNPq. We thank Diego Aranha for helping with the benchmark platforms.

References

[1] J. Aumasson, P. Jovanovic, and S. Neves, "NORX v3.0," norx.io/data/norx.pdf, Sep. 2016. [Online]. Available: norx.io/data/norx.pdf

[2] C. CAESAR, "Competition for authenticated encryption: Security, applicability, and robustness," http://competitions.cr.yp.to, Apr. 2013. [Online]. Available: http://competitions.cr.yp.to

[3] M. J. Dworkin, "Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac," *Special Publication (NIST SP)-800-38D*, 2007.

[4] P. Rogaway, "Authenticated-encryption with associated-data," in *ACM Conference on Computer and Communications Security*. ACM, 2002, pp. 98–107.

[5] M. Videau, "estream," in *Encyclopedia of Cryptography and Security (2nd Ed.)*. Springer, 2011, pp. 426–427.

[6] B. Preneel, "AHS competition/sha-3," in *Encyclopedia of Cryptography and Security (2nd Ed.)*. Springer, 2011, pp. 27–29.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Duplexing the sponge: single-pass authenticated encryption and other applications," *IACR Cryptology ePrint Archive*, vol. 2011, p. 499, 2011.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Sponge functions," in *ECRYPT hash workshop*, vol. 2007. Citeseer, 2007.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "On the indifferentiability of the sponge construction," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 4965. Springer, 2008, pp. 181–197.

[10] J. Aumasson, P. Jovanovic, and S. Neves, "NORX: parallel and scalable AEAD," in *ESORICS (2)*, ser. Lecture Notes in Computer Science, vol. 8713. Springer, 2014, pp. 19–36.

[11] D. J. Bernstein, "Chacha, a variant of salsa20," in *Workshop Record of SASC*, vol. 8, 2008.

[12] A. Biryukov and D. Khovratovich, "PAEQ: parallelizable permutation-based authenticated encryption," in *ISC*, ser. Lecture Notes in Computer Science, vol. 8783. Springer, 2014, pp. 72–89.

[13] A. Holdings, "Arm: Media fact sheet," https://www.arm.com/-/media/arm-com/news/ARM-media-fact-sheet-2016.pdf?la=en, Sep. 2016. [Online]. Available: https://www.arm.com/-/media/arm-com/news/ARM-media-fact-sheet-2016.pdf?la=en

[14] ——, "Processors cortex-a," http://www.arm.com/products/processors/cortex-a, Mar. 2017. [Online]. Available: http://www.arm.com/products/processors/cortex-a

[15] J. Aumasson, P. Jovanovic, and S. Neves, "Norx reference implementations (software)," https://github.com/norx/norx, 2015.

[16] D. J. Bernstein, "Supercop: System for unified performance evaluation related to cryptographic operations and primitives," 2009.