Bachelor thesis
Computer Science

Radboud University

Optimizing NORX for Atmel 8-bit AVR microcontrollers

Author:
Leon Botros
s4160894

First supervisor/assessor:
Dr. P. Schwabe
p.schwabe@cs.ru.nl

Second assessor:
Dr. L. Batina
lejla@cs.ru.nl

March 13, 2015

Abstract

This thesis presents the first results of NORX, an authenticated encryption algorithm, on Atmel 8-bit AVR microcontrollers. Even though NORX was mainly designed with 64-bit CPUs in mind, the designers claim that NORX is also compatible with smaller architectures, for instance 8-bit architectures such as the AVR. We show that by implementing the core parts of NORX in AVR Assembly a significant speedup is achieved over the reference implementation. The implementation of NORX32 which provides 128-bit security and uses 4 rounds of permutation requires 146 cycles/byte, whereas the reference implementation takes 393 cycles/bytes. Despite the fact that focus was on primarily on speed, code size also decreased drastically.

Contents

Chapter 1

Introduction

Today's standard for authenticated encryption is dominated by AES-GCM. While this standard is trusted by the research community as secure, there are some practical disadvantages. One way of finding alternatives is through public competitions. The submissions are evaluated by a committee of cryptographers from different institutes all over the world. Throughout these competitions the cryptographers get a lot of feedback, e.g., a security evaluation, so they can fix vulnerabilities and update their submission. Others can also prove that an algorithm is broken or has serious design flaws, after which the submission is either withdrawn or discarded. This way of designing new algorithms is seen by many a tremendous boost to the research community. Upon withstanding several rounds of critical analyses, confidence in the security of a cipher is increased significantly.

In a competition conducted by the US Government's National Institute of Standards and Technology (NIST), the submission Rijndael [1] by Daemen and Rijmen was selected to replace the Data Encryption Standard (DES) with Advanced Encryption Standard (AES). Rijndael was selected because it offered the best performance across a lot of platforms.

Another example of development through competition is SHA-3 [2]. Even though SHA-2 has not been broken yet, NIST held a competition to find an alternative in case it was broken, what many believe, was just a matter of time. This competition was won by Keccak [3] (actually a subset of Keccak, since it is a family of sponge functions which can be used for a variety of things) which was developed by Bertoni, Daemen, Van Assche and Peeters.

CAESAR [4], not to be confused with the famous politician from ancient Rome, is an acronym which stands for Competition for Authenticated Encryption: Security, Applicability, and Robustness. The goal of this competi-

tion is to find authenticated ciphers which offer advantages over the current standards (like, for example, AES-GCM).

NORX [5] is an authenticated encryption cipher designed by Aumasson, Jovanovic and Neves. It is one of many submissions for the CAESAR competition. The designers included software implementations on 32- and 64-bit processors.

The designers' C reference implementation, which is part of the submission to CAESAR, can be compiled by for example AVR-GCC, a C compiler for AVRs, but produces big and relatively slow machine code. A way to solve this issue is to create a platform-specific implementation. There are already some optimized versions for CPUs supporting AVX and AVX2 (Intel and AMD CPUs) and NEON-enabled ARMs (Smartphones).

Even though NORX is originally designed with 64-bit architectures in mind, the designers claim that it should also perform well on smaller architectures, e.g., 8-bit AVRs. To implement an optimized version for the 8-bit AVR platform and see whether this claim is true is the goal of this thesis.

Chapter 2

Preliminaries

In this chapter we look at both the algorithm and the target platform. In Section 2.1 we take a look at how exactly NORX works. My goal here was to leave out (small) details that do not matter for optimizing for speed.

Programming for 8-bit devices is a bit different from programming for 32- and 64-bit machines, especially on assembly level. In Section 2.2 we take a look at what we need to know about 8-bit AVRs in order to optimize NORX for this platform.

2.1 NORX

2.1.1 AE(AD)

Authenticated encryption (AE) schemes form a class of symmetric cryptographic algorithms. The goal of this class is to be able to send out a confidential message that is also authenticated. General input and output of these algorithms is shown in Figure 2.1.

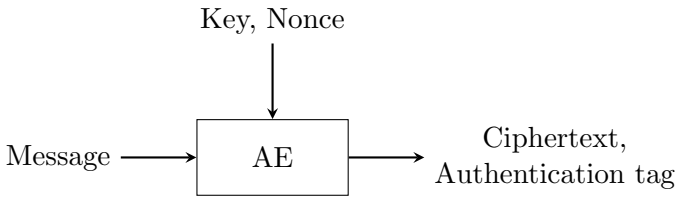

Figure 2.1: Authenticated encryption.

Authenticated encryption with associated data (AEAD) is different from

authenticated encryption in one aspect: The idea is to send out a message in such a way that part of it is confidential, part of it is in the clear and all of it is authenticated. This is very useful if you want to send along a message header containing routing information for datagrams in a network protocol. The header is not confidential and is sent in the clear but the whole message is authenticated. An overview is shown in Figure 2.2.

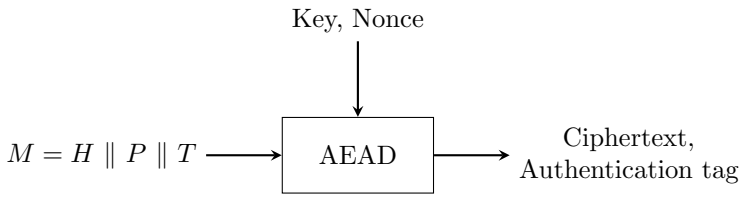

Figure 2.2: Authenticated encryption with associated data.

Since this is symmetric (or secret-key) cryptography, both parties have the shared secret key which they use for both encryption and decryption. This key is created using a key exchange protocol such as Diffie-Hellman [6]. The nonce is a public pseudo-random generated number. The designers of NORX state that NORX is moderately resistant against reuse of nonces as long as the header data is unique, but fresh nonces are strongly recommended. The authentication tag, also called a message authentication code (MAC), is then used to authenticate the (whole) message.

2.1.2 Encryption

What follows is a short overview of NORX encryption. A full detailed overview can be found in the NORX specification on the NORX website [5].

Symbol	Description
$a \parallel b$	Concatenation of bit strings a and b.
$\lvert x \rvert$	Size of bit string x.
$\neg, \wedge, \vee, \oplus$	Bitwise negation, AND, OR and XOR.
$x \ll n, x \gg n$	Bit shifts left/right of bit string x by n bits.
$x \lll n, x \ggg n$	Bit rotations left/right of bit string x by n bits.
$\longleftarrow$	Variable assignment.

Figure 2.3: Symbols & Operations used in NORX.

Instances

A NORX instance is parameterized by

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

A NORX instance is denoted by `NORXW-R-D-|A|`. In this thesis we focus on `NORX32-4-1`, but many of the optimizations can be used for other instances. Since we did not denote a tag size, the default tag size of $4W$ bits is used.

Parameters

Input

- a key K of $4W$ bits (128 bits for NORX32, 256 bits for NORX64)
- a nonce N of $2W$ bits
- a message $M = H \parallel P \parallel T$ where

  H is a header,

  P is the payload,

  T is a trailer.

Output

- a cipher text of size $|P|$ (encrypted payload of same size as P)
- an authentication tag, default size is $4W$ bits

The monkeyDuplex construction

NORX is an AEAD scheme. NORX is not based on a block cipher, rather it uses a fixed permutation to transform the internal state of the algorithm. This idea is called permutation-based encryption and was first proposed in 2012 by Bertoni, Daemen, Van Assche and Peeters [7].

NORX uses a monkeyDuplex construction, an adaptation of the so-called duplex construction. The duplex construction is a duplexed version of a sponge construction. Sponge constructions consist of an internal state, a padding function (so that the input can be absorbed by the state) and a

state permutation. They are conveniently called sponges because they behave like sponges (absorb and squeeze). Sponges can absorb big amounts of data and squeeze out a fixed-size output. A duplexed sponge construction alternates absorbing and squeezing, this way an output is produced for every block of padded input. This property makes it very useful for authenticated encryption. Authenticated encryption requires one call to the permutation per message block while allowing arbitrarily long input (and output) sizes. Sponge- and duplex constructions are not only used for authenticated encryption. They can also be used for hashing, pseudo-random-number generation and key derivation. An example would be the SHA-3 hash-function-competition winner Keccak [3]. NORX extends the construction by adding in more lanes to add parallelism, hence the parallelism degree (number of parallel lanes) parameter D. An overview of NORX's monkeyDuplex construction can be seen in Figure 2.4.

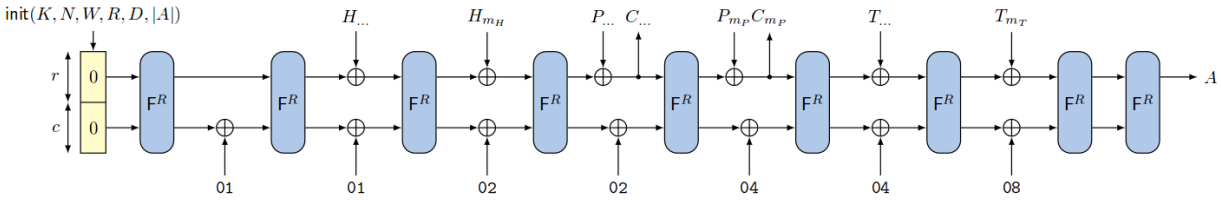

Figure 2.4: Layout of NORX for $D = 1$.

NORX State

A NORX state S consists of 16 W-bit-sized words, arranged in a 4×4 matrix:

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

The entries $s_0, \ldots, s_9$ are called the *rate words* which will hold the message data. The entries $s_{10}, \ldots, s_{15}$ are called *capacity words*.

State initialization

The NORX state is initialized using the key $K = k_0 \parallel k_1, \parallel k_2 \parallel k_3$, the nonce $N = n_0 \parallel n_1$ and constants u_0 through u_9:

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

The constants can be found in the NORX specification. The NORX parameters (W, R, D and $|A|$) are integrated in state S followed by R iterations of the round function F, which is a fixed permutation on the state S. We will get to the details of this function later on.

$$s_{14} \longleftarrow s_{14} \oplus v$$
$$S \longleftarrow \mathsf{F}^R(S)$$
$$\textbf{where } v = (R \ll 26) \oplus (D \ll 18) \oplus (W \ll 10) \oplus |A|$$

Similarly, a domain separation constant is integrated into state S:

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

This time around, v is a domain separation constant. NORX performs domain separation by XORing the domain separation constant v of the next step with the least significant byte of s_{15}. This is done each time before the state is tranformed by the NORX permutation $F^R(S)$. How to calculate v for each phase can be found in the NORX specification, but it does not matter for optimizations described in this thesis. This concludes the initialization.

Padding

A padding function appends bits to the input message so that the length of the padded input is a multiple of the rate r. These r-bit blocks can now effectively be absorbed in the rate words. NORX uses multi-rate padding [8]. The padding rule is pretty simple:

$$pad_r : X \mapsto X \parallel 10^q1$$
$$\textbf{where } q = (-|X| - 2) \bmod r$$

This mapping pads the bit string X to a multiple of the rate r (hence the name) and the last block is not an all-zero block (0^r).

Message processing

Message processing consists of the following steps:

1. Header processing
2. Branching (Only if $D \neq 1$)
3. Payload processing
4. Merging (Only if $D \neq 1$)
5. Trailer processing

Since we focus on `NORX32-4-1`, we can ignore the branching and merging steps. Message (header, payload and optionally a trailer) blocks are injected (by XORing) into the rate words $s_0, \ldots, s_9$. Processing payload blocks also outputs a block of the encrypted cipher text, whereas header and trailer processing does not; as can be seen in Figure 2.4. This is because the header and trailer are meant to be sent in the clear. In this figure you can also see that a domain separation constant is integrated in the capacity words. By using this duplex construction, a message of arbitrary length is still processed in a single pass of the algorithm. If a different parallelism degree $D > 1$ is used, message processing is done in D different lanes which improves efficiency on multi-core platforms.

Header processing If there is no header ($|H| = 0$), this step is skipped. If there is a header, it is padded to a multiple of r bits using the multi-rate padding previously mentioned. Let $\mathsf{pad}_\mathsf{r}(H) = H_0 \parallel \ldots \parallel H_{m_H-1}$ denote the padded header. Let H_l be such an r-bit sized header block with $0 \leq l \leq m_H - 1$. Since rate $r = 10$ (and capacity $c = 6$) the header blocks consist of 10 bits $h_{l,0} \parallel \ldots \parallel h_{l,9}$. Each header block is "injected" into the rate words as follows:

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

Payload processing Let $\mathsf{pad}_\mathsf{r}(P) = P_0 \parallel \ldots \parallel P_{m_p-1}$ denote the padded payload. A payload block $P_l = p_{l,0} \parallel \ldots \parallel p_{l,9}$ is encrypted as follows:

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

The encrypted payload block $c_l = c_{l,0} \parallel \ldots \parallel c_{l,9}$ is the result of encrypting payload block p_l. The last block P_{m_p-1} creates a truncated ciphertext block so that $|C|$ is equal to $|P|$.

Trailer processing Trailer processing is similar to header processing.

The round function F

The round function F uses G (details later) to transform a NORX state S. G is a function which tranforms four W-sized words. The function F is a permutation of $b = r+c$ bits, where b is called the *width*, r the *rate*, and c the *capacity*. The equation $b = r + c$ expresses a trade off between security and efficiency. The rate determines the efficiency and the capacity determines the security strength. First, the words in the columns in S are transformed as follows:

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}

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

Second, the words in the diagonals of S are transformed as follows:

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}

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

The application of R rounds of F on NORX state S is denoted by $\mathsf{F}^R(S)$. This application is also called a NORX permutation.

The function G

G is a function which takes four W-bit-sized words a, b, c, d and transforms them as follows:

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

The rotation offset constants r_0, r_1, r_2, r_3 in NORX32 are set to 8, 11, 16, 31, respectively. This function is inspired by the quarter-round function of ChaCha [9], an improved version of the stream cipher Salsa20 [10] from the eSTREAM software portfolio [11]. The design principle of ChaCha's quarterround is sometimes named ARX (add-rotate-xor). NORX replaces the addition with a logical and, hence the name NORX (short for NOTARX). Together, F and G are at the core of NORX. The computationally expensive part of NORX is obviously the R calls to the round function F for each block of r bits. The following chapters describe how I optimized this part for the Atmel AVR ATmega family of microcontrollers.

2.1.3 Decryption

Decryption is very similar to encryption. See the full specification.

2.1.4 Tag generation

Tag generation is performed after the whole message is processed by the duplex construction. The state S is transformed one last time using $F^R(S)$ and

then extracting the $|A|$ least significant bits from the rate words $s_0, \ldots, s_9$ yields the tag:

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

If the default tag size of $4W$ is used, this is equivalent to $A \longleftarrow s_0 \parallel s_1 \parallel s_2 \parallel s_3$.

2.1.5 Tag verification

Tag verification is performed by compared the received tag A to the tag A' which is generated after decryption. If $A = A'$, the tag verification succeeds; otherwise it fails. The NORX specification also states that the decrypted payload should be securely erased if tag verification fails and that tag verification should not leak information regarding the compared strings.

2.2 8-bit Atmel AVR Microcontrollers

In this chapter we give necessary background of the target platform. In order to understand how to optimize for 8-bit AVRs we first need to have an understanding of their architecture.

2.2.1 Architecture and instruction set

The AVR is a simple RISC (Reduced Instruction Set Computing) microcontroller. A microcontroller is in fact a small computer. It has a CPU, memory (a mix of SRAM, flash and EEPROM) and some I/O-pins. The microcontroller uses the flash memory to store programs. AVRs use a two-stage single-level pipeline, which means that while an instruction is executed the next instruction is fetched. A general overview provided in the ATmega2560 Data sheet by Atmel [12] of this Harvard architecture is shown in Figure 2.5.

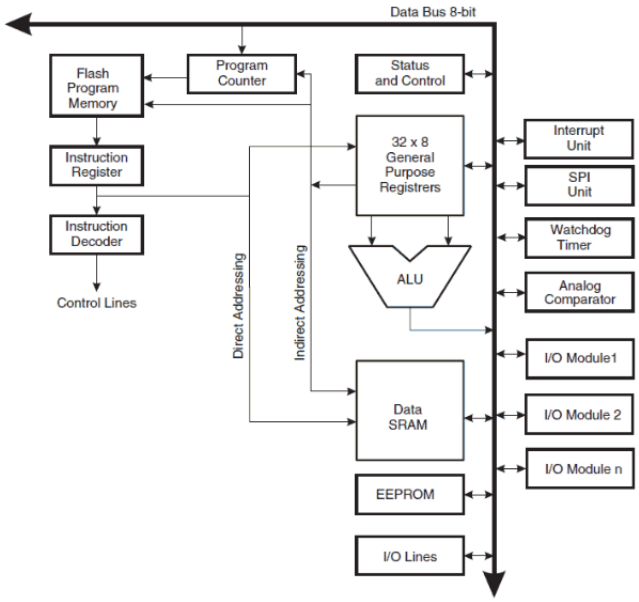

Figure 2.5: 8-bit AVR Harvard Architecture.

Most instructions are executed in a single clock cycle. Memory operations take two cycles to complete. There are three main groups of 8-bit (there are actually some 32-bit AVRs on the market too but those are not nearly as popular as the 8-bit ones) AVR microcontrollers to distinguish:

Series	Memory (kB)	I/O-pins
ATtiny	$1-8$	$6-32$
ATmega	$4-256$	$28-100$
ATxmega	$16-384$	$44-100$

Table 2.1: An overview of the AVR microcontroller family.

In general, the **ATtiny** series is used for small applications since they are small in size and do not consume a lot of power. The **ATmega** series is by far the most popular of AVRs, they have a nice amount of memory and are very suitable for medium to complex applications. The **ATxmega** series is used for more complex applications that require more memory and speed. Also, the higher-end versions offer more features than the lower-end ones. In this thesis I used an ATmega2560 on an Arduino Mega development board. The maximum clock frequency is 16 MHz, it has 256kB of flash and 8kB of SRAM. A full data sheet of this device is provided by Atmel [12].

The AVR has 32 8-bit registers available, R0-R31. Memory addresses are 16 bits. Registers R26-R31 can be used as pointers towards a memory location. Since addresses in the SRAM are 16-bits, two registers are required. These pointers are called X (R26:R27), Y (R28:R29) and Z (R30:R31).

The full instruction set manual can be found on Atmel's website [13]. Some important instructions are listed in Table 2.2 and some arithmetic operations used by NORX in Table 2.3. Note that when using arithmetic instructions with two operands, the first of the input operands gets overwritten by the outcome.

Instruction	Example	Description	Cycles [1]
MOV	MOV R25, R24	Copies the content of R24 into R25.	1
MOVW	MOVW R25, R17	Copies a register pair (R17/R18 into R25/R26).	1
LD	LD R25, X(+)	Loads content of memory location that X points to in R25 (and increments X afterwards).	2
ST	ST X, R25	Stores R25 at memory location X.	2
LDD	LDD R25, X + n	Loads what is at memory location X + n in R25, does not change X.	2
STD	STD X + n, R25	Stores R25 at memory location X + n.	2
PUSH	PUSH R25	Pushes R25 onto the stack.	2
POP	POP R25	Pops the top of the stack into R25.	2
RCALL	RCALL subroutine	Relative call to a subroutine.	3
RET	RET	Returns to the calling code.	4

Table 2.2: Frequently used general instructions.

Instruction	Description	Cycles
EOR	Exclusive OR.	1
AND	Bitwise AND.	1
LSL/LSR	Logical Shift Left/Right, inserts 0 and sets the carry flag.	1
ROL/ROR	Rotate Left/Right, inserts carry and sets the carry flag.	1

Table 2.3: Arithmetic instructions used in NORX.

Again, some of the higher-end series offer more instructions than the lower-end ones. In my implementation the load and store with displacement instructions (STD and LDD) are used. These only work on pointers X and Z, **not** on Y. Unfortunately these instructions are not available on ATtiny devices. This means that the main goal is of this thesis is to produce an implementation for the ATmega series. Nevertheless, the implementation can easily be adapted to run on the ATtiny family. Also, it is worth noting that some instructions take more cycles on the ATxmega family.

[1] Listed are the cycles on ATmega. Some instructions take more cycles on ATxmega.

2.2.2 AVR-GCC

To compile C code for the AVR, AVR-GCC version 4.8.2 is used. Unless noted otherwise the following flags are used:

```
-Wall -mmcu=atmega2560 -O3 -DF_CPU=16000000
```

When using AVR-GCC we must keep in mind that AVR-GCC uses registers too and some require special attention:

Registers	How to use
R0	Temporary register, can be changed by C code.
R1	Zero register, can be used but must be cleared afterwards.
R2-R17	Call-saved registers. Calling C subroutines leaves them unchanged. Assembler subroutines that use these registers must restore them.
R18-R27	Call-used registers. Calling C subroutines can change them. Assembler subroutines can use these registers freely, no restore required.
R28-R29	Y pointer is used by AVR-GCC as a frame pointer. Needs to be restored if changed.
R30-R31	Call-used registers. Same usage as R18-R27.

Arguments of a subroutine are allocated in R25-R8. If there are more arguments, they are pushed on to the stack. Return values can be put in R24 (8-bit), R25-R25 (16-bit), R25-R22 (32-bit) or R25-R18 (64-bit).

2.2.3 Timers

Measuring execution speed of code on the AVR is done using timers.

The ATmega2560 has two 8-bit timers (`timer0` and `timer2`) and four 16-bit timers (`timer1`, `timer3`, `timer4` and `timer5`). Timers have an internal counter register (TCNT) that can be set to scale off the system clock using a prescaler. A prescaler can be set by setting some bits in the timer control register (TCCR).

Without a prescaler, the counter is incremented at the same rate as the system clock (16 MHz for the ATmega2560). Using a prescaler slows this down. For example, when the prescaler is set to 8, the counter register of the timer is incremented every 8 system clock cycles. Now the counter is incremented at a rate of $16/8 = 2$ MHz. The prescaler can be set to 8, 64, 256 or 1024.

The 8-bit timer counts from 0 to 255 (0x00 to 0xFF). The 16-bit timer counts from 0 to 65535 (0x0000 to 0xFFFF). After that, they overflow (go back to 0).

The timer interrupt mask (TIMSK) is a register that enables a timer to cause an overflow interrupt. When a timer overflows, the overflow flag is set and the timer overflow interrupt flag (TIFR) is set. Normal execution will be interrupted and the processor will execute the code of the interrupt service routine (ISR). Global interrupts need to be enabled in order use this.

My supervisor gave me a function which can be used to count CPU cycles on the AVR. It uses `timer0` and `timer1` to make a 24-bit timer. `timer0` is set to prescale directly off the system clock. When `timer0` overflows, no interrupt is caused because this bit is not set in TIMSK. `timer1` is set with a prescaler of 256 (2^8). This means that `timer1` overflows every $2^{16} \cdot 2^8 = 2^{24}$ system clock ticks. When this timer overflows, an interrupt service routine is executed. In this routine 2^{24} is added to a 64-bit unsigned long long called `ticks`. To get the amount of clock ticks from the last overflow until the time that the function is called, the counter values TCNT0 and TCNT1 are read. Since `timer1` uses a prescaler of 256, the value in the counter register has to be shifted 8 bits to the left (multiply by 2^8). The values are then bitwise ORed (because of overlapping bits) with `ticks` and the resulting value is returned. The function itself takes some cycles too, so there is some overhead. However, this overhead is constant and can be measured using the function without any code in between.

Chapter 3

NORX on AVR ATmega

3.1 Optimizing G

Intuitively, the NORX permutation (F^R) seemed like a good place to start when trying to optimize NORX on the AVR. This permutation is called in every part of NORX. Every message block requires one NORX permutation. So the longer the message, the more permutations.

After deciding that optimizing the round function F would be a good idea to start, I started at the smallest building block of this function, which is G (page 11).

G uses 4 operations which can be seen in Table 3.1. The table also shows how many cycles are needed to perform the operation on the AVR. Note that these operations work on 32-bit words, since the NORX state consists of sixteen 32-bit words in `NORX32-4-1`.

An exclusive-or operation takes 4 cycles on the AVR since it has to XOR the 4 bytes separately, taking 1 cycle each. The same applies to the bitwise-and operation. A bit shift is performed by first doing one logical shift (logical shifts insert a zero) followed by 3 rotate through carry instructions. A 32-bit rotation costs a little more on the AVR, because we have to set the carry flag first. First a byte is copied into a temporary register, then either a rotation or a shift is performed on this register. Now that the carry flag is set, we can rotate each of the 4 bytes. In total this takes 6 cycles.

Table 3.1 also shows how many of these operations are used in NORX. The rotation offsets (8, 11, 16, 31) are carefully chosen by the designers to reduce cost on 8-bit architectures without a barrel shifter, such as the AVR ATmega. If, for example, we want to shift a 32-bit word 11 bits left, we can do this by rotating left 3 bits and then renaming the registers. In general

we never have to rotate more than to the closest multiple of 8, either left or right. With these offsets we only have to do 4 rotations (3 bits right for the offset 11 and 1 bit left for the offset 31).

	XOR	AND	Shift	Rotation	Total
Operations on the AVR	12	4	4	4	24
Cycles on AVR	4	4	4	6	
Total cycles on the AVR	48	16	16	24	104

Table 3.1: Operations & Cycles used for G on the AVR.

So, all arithmetic operations for one G-permutation take 104 cycles. This does not take into account reading or writing memory. If we assume that we need to read all words a, b, c, d this will cost $4 \cdot 4 \cdot 2 = 32$ cycles. Writing them back costs 32 cycles too. This brings the total to 168 cycles. Plus, to calculate

$$a \longleftarrow (a \oplus b) \oplus ((a \wedge b) \ll 1)$$

for example we need to keep a copy of either a or b in a temporary register. Since both MOV and MOVW instructions cost 1 cycle, MOVW is used where possible. Keeping a copy of a word costs 2 cycles this way (not counting the use of an extra register). We have to do this 4 times in G. This brings the total to 176 cycles.

G takes four 32-bit words as inputs. Ideally, this would fit into the registers and all calculations can be done without having to do slow memory operations before writing the final result back. Unfortunately, we only have 32 8-bit registers of which 18 have to be restored after use. Using call-saved registers is not different in cycles from storing/loading intermediate results, since whatever was in there needs to be pushed onto the stack to be restored later. Push and pop operations are operations that work on memory too and are no different in cycles than a regular load or store.

Eight registers (R25-R18) are required to hold the function arguments, four 16-bit addresses of the words. This does not leave a whole lot of registers to use for calculations. Therefore, when I implemented G in AVR assembly, I was forced to store and load after every step of G. Using this G only showed a very little performance increase over the reference implementation. A further, much larger, speedup comes from inlining multiple calls to G as explained in the following section.

3.2 Optimizing F^R

Unrolling G and creating a loop around F, and thus making one big subroutine for F^R, has four advantages:

1. Removing function-call overhead. In the reference implementation, F was called R times in a loop from C code and inside F, G was called 8 times. Depending on the message size this would result in a lot of function calls. And every time it would do exactly the same. A function call pushes a stack frame on the stack containing the return address, local variables and possibly function parameters. When returning to the calling function, these values are popped. Stack operations (PUSH and POP) take 2 cycles each and we want to avoid these when trying to keep the cost low.

2. Allows (efficient) usage of call-saved registers. In the scope of G it was not worth it to use call-saved registers (registers that require restoring at the end of a subroutine). In the scope of F we can save the call-saved registers that are used to do all 8 G calculations and then restore them afterwards.

3. Memory addressing. G has 4 arguments (pointers to a, b, c, d) which take 8 registers in total to store. When instead one subroutine for F is used we have one pointer to the first byte of the NORX state $S = s_0, \ldots, s_{15}$. When stored in pointer register pair Z we can access s_n using the load and store with displacement instructions (STD and LDD). The four bytes of s_n are at memory locations $\mathrm{Z}+4n+0, \ldots, \mathrm{Z}+4n+3$. The displacement when using these instructions can go up to 64, which is exactly the size of the NORX state in `NORX32-4-1`. Unfortunately, this means that the idea of using LDD and STD cannot directly be implemented for NORX instances using a word size of 64 bits, since the NORX state is 128 bytes in that case. Two pointers would probably be required.

4. One word can be kept in registers. After last the column step, $\mathsf{G}(s_3, s_7, s_{11}, s_{15})$, we have to do the first diagonal step, $\mathsf{G}(s_0, s_5, s_{10}, s_{15})$. Instead of storing and loading s_{15}, we can keep these 4 bytes in the registers. This saves 4 stores and 4 loads which is equivalent to 16 cycles per round. The bytes of d are rotated right 3 times at this stage, so some renaming is required.

The main disadvantage to this approach is that it creates AVR assembly code that repeats for most parts except for some addresses, which increases the size of the code.

3.3 Implementation details

3.3.1 Implementation of F^R

The AVR assembly subroutine that I implemented takes 2 arguments. The first one is a pointer to the NORX state. The second one is the number of rounds. By parameterizing the number of rounds the subroutine is also suitable for NORX instances that use a different number of rounds. An overview of how the registers are used in this subroutine can be seen in Table 3.2.

R0	R1	R2:5	R6:9	R10:12	R13	R14:17	R18:21	R22:25	R26:29	R30:31
t_1	t_2	s_0	s_1	s_2	r	a	b	c	d	state ptr

Table 3.2: Register usage in F.

We use two temporary registers so we can use the MOVW instruction instead of MOV. Register R13 holds the number of rounds left. Registers R2 through R12 (registers that were left over) are used to hold 11 bytes of the state in the registers. This way, we only have to load and store them once for all rounds. Note that only 3 of 4 bytes of s_2 are stored.

A basic form of F^R in AVR assembly can be found in Figure 3.1. G is left unimplemented in this overview.

```
F:
    push   r2
    push   r3
    push   r4
    push   r5
    push   r6
    push   r7
    push   r8
    push   r9
    push   r10
    push   r11
    push   r12
    push   r13
    push   r14
    push   r15
    push   r16
    push   r17
    push   r28
    push   r29

    movw   r30, r24
    mov    r13, r22

    ldd    r2, Z + 0
    ldd    r3, Z + 1
    ldd    r4, Z + 2
    ldd    r5, Z + 3
    ldd    r6, Z + 4
    ldd    r7, Z + 5
    ldd    r8, Z + 6
    ldd    r9, Z + 7
    ldd    r10, Z + 8
    ldd    r11, Z + 9
    ldd    r12, Z + 10

round:
    ; G(s0,s4,s8,s12)
    ; G(s1,s5,s9,s13)
    ; G(s2,s6,s10,s14)
    ; G(s3,s7,s11,s15)
    ; G(s0,s5,s10,s15)
    ; G(s1,s6,s11,s12)
    ; G(s2,s7,s8,s13)
    ; G(s3,s4,s9,s14)
    dec    r13
    breq   end
    rjmp   round

end:
    std    Z + 0, r2
    std    Z + 1, r3
    std    Z + 2, r4
    std    Z + 3, r
    std    Z + 4, r6
    std    Z + 5, r7
    std    Z + 6, r8
    std    Z + 7, r9
    std    Z + 8, r10
    std    Z + 9, r11
    std    Z + 10, r12

    clr    r1
    pop    r29
    pop    r28
    pop    r17
    pop    r16
    pop    r15
    pop    r14
    pop    r13
    pop    r12
    pop    r11
    pop    r10
    pop    r9
    pop    r8
    pop    r7
    pop    r6
    pop    r5
    pop    r4
    pop    r3
    pop    r2
    ret
```

Figure 3.1: F^{R} in AVR Assembly.

First, all call-saved registers are saved. The address of the state is stored in Z, the number of rounds is stored in R13. The first 11 bytes of the state are loaded in R2:R12 as previously mentioned. In one round we do all 8 G calculations, afterwards R13 is decreased by one. If R13 is now equal to zero we are done and a zero flag is set. The branch-if-equal (BREQ) jumps if the zero flag is set to the label *end* where the registers are restored again. R1, the zero register, is cleared (set to 0). If the zero flag is not set, it does

not jump but instead just executes the next line. A relative jump (RJMP) is used to jump to *round* because the code for all the G calculations is quite long and the relative jump can jump ±2k lines.

Implementation of G

Now, we need to add the code that performs the G calculations. $\mathsf{G}(a, b, c, d)$ consists of reading the four words, calculating their new values and writing them back. The assembly code is around 150 lines and can be found in Appendix B. This code takes exactly 176 cycles. MOVW is used to make a copy of - for example a in the first step - two bytes simultaneously to R0:R1. In step 2 and 4, rotations right (3 bits and 1 bit) are performed. The rest of the rotating does not take any cycles, because we "rename" the registers. After step 1 for example, instead of looking for d_1 (which is the first byte of d) in R26 we take the value of d_2 stored in R27 and treat it like it is d_1. This way no cycles are used to shuffle bytes.

The address displacements are left variable and contain the capitalized letters A, B, C and D. The code is first preprocessed by a simple python script that replaces these letters by the addresses of the arguments:

```
#!/usr/bin/python

def translate(s, a, b):
  for x, y in zip(a, b):
    s = s.replace(str(x), str(y))
  return s

g_args = [[0, 4, 8, 12],
          [1, 5, 9, 13],
          [2, 6, 10, 14],
          [3, 7, 11, 15],
          [0, 5, 10, 15],
          [1, 6, 11, 12],
          [2, 7, 8, 13],
          [3, 4, 9, 14]]

displacements = [[n * 4 for n in s_n] for s_n in g_args]

with open('g', 'r') as g_file:
  g_filestring = g_file.read()

f_filestring = "".join([translate(g_filestring, 'ABCD', g) for g in
    displacements])

with open('f', 'w') as f_file:
  f_file.write(f_filestring)
```

The resulting subroutine does not use the registers we had left for the first 11 bytes of the state. To change this, I removed the parts in the

G-calculations that have either s_0, s_1 or s_2 as input that load and store and had them use the registers R2:R12 instead. This saves 4 loads and 4 stores (16 cycles) in $\mathsf{G}(s_0, s_4, s_8, s_{12})$, $\mathsf{G}(s_1, s_5, s_9, s_{13})$, $\mathsf{G}(s_0, s_5, s_{10}, s_{15})$ and $\mathsf{G}(s_1, s_6, s_{11}, s_{12})$ and 3 loads and 3 stores (12 cycles) in $\mathsf{G}(s_2, s_6, s_{10}, s_{14})$ and $\mathsf{G}(s_2, s_7, s_8, s_{13})$.

I also removed the part that stores and loads s_{15} in the last column step and the first diagonal step. This required me to rename the registers containing s_{15} to undo the 3-byte rotation. This saves another 16 cycles between $\mathsf{G}(s_3, s_7, s_{11}, s_{15})$ and $\mathsf{G}(s_0, s_5, s_{10}, s_{15})$. In total, 104 cycles are saved per round.

Cycle Analysis of F^{R}

The result is an AVR assembly subroutine of about 1200 lines that performs one NORX permutation (F^{R}) on NORX state S containing 32-bit sized words. We can calculate exactly how many cycles this subroutine takes, see Table 3.3. This analysis does not take into account the cost of calling this function.

Action	# Performed	Cycles	Total
Push call-saved register	18	2	36
Load first 11 state bytes	11	2	22
Move parameters	1	3	3
Round			
G	$8R$	176	$1304R$ [1]
Decrement R2	R	1	R
Branch-if-equal (false)	$R-1$	1	$R-1$
Relative jump	$R-1$	2	$2R-2$
Branch-if-equal (true)	1	2	2
Store first 11 state bytes	11	2	22
Clear R1	1	1	1
Pop call-saved register	18	2	36
Return to caller	1	4	4
Total			$1308R + 123$

Table 3.3: Cycle analysis of the new F^{R} subroutine.

[1] Since 104 cycles are saved per round.

3.4 Results

3.4.1 Execution speed

Whereas one F-call took 3454 cycles using the reference implementation, the new F takes 1444 cycles. The new F^R is optimized for a variable number of rounds. The average cycles per round converge to 1312 as the number of rounds increases. This value is close to what we expect if we take a look at our cycle analysis.

Measuring is performed by using timers. The measured code is put between a function (cpucycles) that reads the value of the 24-bit timer mentioned in subsection 2.2.3 on page 16. Reading this value costs a few cycles. We first measure this overhead by measuring how many cycles "empty code" takes, i.e., we read the value twice without any code in between and calculate the cycle count difference.

To measure the effect of this change on `NORX32-4-1`, I ran multiple encryptions on different sized messages. Since header and trailer processing is performed completely similar, I only ran encryption using different sized headers and payloads and the trailer is left empty. Use of the trailer is optional, because anything that can be put in a trailer can also be put in the header. In this experiment header and payload are equally sized (both half the message length). In theory, payload bytes take a little more cycles since they produce cipher text and this has to be written in the memory. That is why I also ran the encryptions in which the payload size was equal to the message size (so no header or trailer) and the difference was negligible. Keys and nonces are pseudo-randomly generated using code I got from my supervisor, but we might as well use all-zero keys and nonces. The results are listed in Table 3.4. The cycles-per-byte values converge to a point as the message size grows, this is due to overhead caused by the initialization and finalization. This point (asymptotic speed) is called "long" in the table. If X is the number of cycles required to encrypt a 2048-byte message and Y is the number of cycles required to encrypt a 1024-byte message, the asymptotic speed is computed as $(X - Y)/1024$ cycles per byte.

Ref/AVR	Message size (bytes)	Cycles	Cycles/byte
Ref	8	77462	9682
	16	77558	4847
	32	77750	2429
	64	78134	1220
	128	108808	850
	256	170364	665
	512	263466	514
	1024	449670	439
	2048	852088	416
	long		393
AVR	8	30268	3783
	16	30364	1897
	32	30556	954
	64	30940	483
	128	42258	330
	256	64902	253
	512	99636	194
	1024	169104	165
	2048	318594	155
	long		146

Table 3.4: `NORX32-4-1` benchmark results on the ATmega2560.

3.4.2 Code size

Since some AVRs have limited memory, code size is an aspect that we should not forget. The ATmega2560 has 256kB of flash, which is the maximum amount in the ATmega series. The file `norx.c` contains both high-level functions `norx_aead_encrypt` and `norx_aead_decrypt`. When compiled, it creates an object file. The program `avrsize` is used to measure the size of this file. The reference implementation takes up 65578 kB. The AVR-optimized implementation takes up 6206 kB in C and 922 kB in AVR assembly.

3.4.3 Correctness

Unfortunately, a formal proof of correctness is beyond the scope of a Bachelors thesis. I will stick to comparing test vectors and the output of the reference implementation to the output of the AVR optimized implementation. To test whether the outcome of encryption is correct the results are compared to full NORX computations listed on page 55 and 56 in the NORX specification [5]. The ciphertext and authentication tags match the ones listed there, for both `NORX32-4-1` and `NORX32-6-1`. On a lower level, F is performed up to 8 times on the state listed on page 53 of the specification and those values match too. During this project I noticed that a small bug in F or G would result in very dramatic changes to the resulting state.

3.5 Future work

3.5.1 NORX64

Implementing the NORX permutation F^R for a NORX state of 64-bit words is a bit more of a challenge. The four words of G now take 8 registers each, which means intermediate saving and loading from the memory is required to perform G.

Also, the state is 1024 bits in NORX64. LDD and STD instructions have a maximum displacement of 63 bytes. In order to access the state, two pointers to both state halves of 512 bits are required (X and Z, Y cannot be used with LDD and STD).

3.5.2 Other parts of NORX

I prioritized optimizing F^R because most cycles went there. This does not mean that all the other parts do not need optimizing. The table below gives an indication (they might be off by a few cycles) of where the cycles currently go when encrypting an 8-byte message (4-byte header, 4-byte payload, no trailer).

Part	Cycles
Initialization	5680
Header processing	6110
Payload processing	6411
Trailer processing	65
Generating tag	12210

What strikes me here is how many cycles are used to generate the authentication tag. This is a lot of overhead for encrypting a small message. If the default authentication tag size of $4W$ is used, first the state is permutated one last time by F^{R} followed by setting A to the first $4W$ bits of the rate words $r_0 \parallel \ldots \parallel r_9$. This is basically:

$$S \longleftarrow \mathsf{F}^{R}(S)$$
$$A \longleftarrow s_0 \parallel s_1 \parallel s_2 \parallel s_3$$

The next step would be to figure out why so many cycles are used for something so simple and how we can decrease this number. This is just one example of a part that might need optimizing.

Chapter 4

Related Work

Comparing execution speeds of algorithms on different platforms is very hard if not impossible. For this reason I will only consider cryptographic primitive implementations for 8-bit AVRs.

As far as I know there are no other implementations of NORX for 8-bit AVRs. There are several cryptolibraries for the AVR, e.g., NaCl [14], AVR-Crypto-Lib [15] and TinyECC [16].

NaCl (pronounced as "salt") uses the stream cipher Salsa20 [10] to encrypt and Poly1305 [17] to authenticate messages. The implementation for AVRs requires 277 cycles/byte for Salsa20 and 211 cycles/byte for Poly1305. Both Salsa20 and Poly1305 use 256-bit keys. Salsa20 uses 8 quarterround function calls that take 176 cycles each in this implementation.

Unfortunately, there are not a lot of authenticated encryption ciphers implementations for the AVR. I hope that throughout the course of the CAESAR competition we are going to see some more implementations for the AVR.

Bibliography

[1] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, version 2, 1999. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf. 2

[2] Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, 2007. http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf. 2

[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission, January 2011. http://keccak.noekeon.org/. 2, 7

[4] CAESAR - Competition for Authenticated Encryption: Security, Applicability, and Robustness, 2014. http://competitions.cr.yp.to/caesar.html. 2

[5] J.P. Aumasson, P. Jovanovic, and S. Neves. NORX, A Parallel and Scalable Authenticated Encryption Algorithm, 2014. https://norx.io/. 3, 5, 27, 32

[6] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976. 5

[7] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Permutation-based encryption, authentication and authenticated encryption. Directions in Authenticated Ciphers, July 2012. http://keccak.noekeon.org/KeccakDIAC2012.pdf. 6

[8] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography (SAC)*, 2011. http://sponge.noekeon.org/SpongeDuplex.pdf. 8

[9] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, 2008. http://cr.yp.to/chacha.html. 11

[10] Daniel J Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008. http://cr.yp.to/snuffle.html. 11, 29

[11] The ECRYPT Stream Cipher Project, 2004-2008. http://www.ecrypt.eu.org/stream. 11

[12] ATmega2560 Datasheet. http://www.atmel.com/images/doc2549.pdf. 13, 14

[13] Atmel 8-bit AVR Instruction Set Manual. http://www.atmel.com/images/doc0856.pdf. 14

[14] Michael Hutter and Peter Schwabe. NaCl on 8-Bit AVR Microcontrollers. In *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 156–172. Springer Berlin Heidelberg, 2013. https://cryptojedi.org/papers/avrnacl-20130220.pdf. 29

[15] AVR-Crypto-Lib. https://www.das-labor.org/wiki/AVR-Crypto-Lib/en. 29

[16] An Liu and Peng Ning. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*, pages 245–256. IEEE, 2008. http://discovery.csc.ncsu.edu/software/TinyECC/TR-2007-36.pdf. 29

[17] Daniel J Bernstein. The Poly1305-AES message-authentication code. In *Fast Software Encryption*, volume 3557 of *LNCS*, pages 32–49. Springer, 2005. http://cr.yp.to/mac.html. 29

Appendix A. How to run

On Debian/Ubuntu:

Source The source code is available on Github:

```
$ git clone https://github.com/leonbotros/norxavr.git
```

Dependencies The packages avr-gcc, avr-libc, avrdude and binutils-avr are required:

```
$ sudo apt-get install gcc-avr avr-libc avrdude binutils-avr
```

Permissions Make sure the development board is plugged in through USB and seen by the OS as `/dev/ttyACM0`. Otherwise either unplug the other device(s) or change the `DEVICE` variable in the makefile. You can list devices by running the command:

```
$ ls -l /dev/ttyACM*
```

Access to this file is restricted to users in the group `dialout`. You can add a user to this group by running:

```
$ sudo useradd -G dialout <user>
```

Compile & Run Go to the directory `norxavr3241/` or `norxref3241/`, depending on which you want to run. To recreate the encryption results listed on page 55 (`NORX32-4-1`) and 56 (`NORX32-6-1`[1]) of the NORX specification [5], run:

```
$ make norxtest
```

To recreate the results in section 3.4 of this paper, run:

```
$ make speedtest
```

[1] The number of rounds can be changed by editing the variable `NORX_R` in the file `norx/norx_config.h`.

Appendix B. AVR Assembly: G

```
ldd     r14, Z + A + 0
ldd     r15, Z + A + 1
ldd     r16, Z + A + 2
ldd     r17, Z + A + 3

ldd     r18, Z + B + 0
ldd     r19, Z + B + 1
ldd     r20, Z + B + 2
ldd     r21, Z + B + 3

ldd     r22, Z + C + 0
ldd     r23, Z + C + 1
ldd     r24, Z + C + 2
ldd     r25, Z + C + 3

ldd     r26, Z + D + 0
ldd     r27, Z + D + 1
ldd     r28, Z + D + 2
ldd     r29, Z + D + 3

;; STEP 1

movw    r0, r14
eor     r14, r18
and     r0, r18
lsl     r0
eor     r14, r0
eor     r26, r14

eor     r15, r19
and     r1, r19
rol     r1
eor     r15, r1
eor     r27, r15

movw    r0, r16
eor     r16, r20
and     r0, r20
rol     r0
eor     r16, r0
eor     r28, r16

eor     r17, r21
and     r1, r21
rol     r1
eor     r17, r1
eor     r29, r17

;; STEP 2

movw    r0, r22
eor     r22, r27
and     r0, r27
lsl     r0
eor     r22, r0
eor     r18, r22

eor     r23, r28
and     r1, r28
rol     r1
eor     r23, r1
eor     r19, r23

movw    r0, r24
eor     r24, r29
and     r0, r29
rol     r0
eor     r24, r0
eor     r20, r24

eor     r25, r26
and     r1, r26
rol     r1
eor     r25, r1
eor     r21, r25

mov     r0, r18
lsr     r0

ror     r21
ror     r20
ror     r19
```

```
ror     r18

mov     r0, r18
lsr     r0

ror     r21
ror     r20
ror     r19
ror     r18

mov     r0, r18
lsr     r0

ror     r21
ror     r20
ror     r19
ror     r18

;; STEP 3

movw    r0, r14
eor     r14, r19
and     r0, r19
lsl     r0
eor     r14, r0
eor     r27, r14

eor     r15, r20
and     r1, r20
rol     r1
eor     r15, r1
eor     r28, r15

movw    r0, r16
eor     r16, r21
and     r0, r21
rol     r0
eor     r16, r0
eor     r29, r16

eor     r17, r18
and     r1, r18
rol     r1
eor     r17, r1
eor     r26, r17

;; STEP 4

movw    r0, r22
eor     r22, r29
and     r0, r29

lsl     r0
eor     r22, r0
eor     r19, r22

eor     r23, r26
and     r1, r26
rol     r1
eor     r23, r1
eor     r20, r23

movw    r0, r24
eor     r24, r27
and     r0, r27
rol     r0
eor     r24, r0
eor     r21, r24

eor     r25, r28
and     r1, r28
rol     r1
eor     r25, r1
eor     r18, r25

mov     r0, r18
rol     r0

rol     r19
rol     r20
rol     r21
rol     r18

std     Z + A + 0, r14
std     Z + A + 1, r15
std     Z + A + 2, r16
std     Z + A + 3, r17

std     Z + B + 0, r19
std     Z + B + 1, r20
std     Z + B + 2, r21
std     Z + B + 3, r18

std     Z + C + 0, r22
std     Z + C + 1, r23
std     Z + C + 2, r24
std     Z + C + 3, r25

std     Z + D + 0, r29
std     Z + D + 1, r26
std     Z + D + 2, r27
std     Z + D + 3, r28
```