Bachelor thesis
Computer Science

Radboud University

Ascon: An attempt in NEON on the Cortex-A8

Author:
Noël Bangma
s4433939

First supervisor/assessor:
Dr. Peter Schwabe
`peter@cryptojedi.org`

Second assessor:
Prof. Dr. Lejla Batina
`lejla@cs.ru.nl`

April 18, 2018

Abstract

We notice that the critical function of the authenticated-encryption algorithm and CAESAR[1] candidate Ascon[10] is the permutation function. While there exists a general optimization for 32-bit architectures utilizing the bit interleaving technique, we attempt to improve performance on the Cortex-A8 using the NEON instruction set extension. We conclude that our implementation is substantially slower than the nonspecific 32-bit optimization, and argue this is because of costly rotates and because Ascon is not well-suited to NEON's strong points for crypto.

Contents

Chapter 1

Introduction

CAESAR [1] *(Competition for Authenticated Encryption: Security, Applicability and Robustness)* is a competition for authenticated-encryption algorithms, with its first round started in 2014. Ascon [10] is one of the competitors. Ascon is designed by Dobraunig, Eichlseder, Mendel and Schläffler of Graz University of Technology. The algorithm is designed to be lightweight and provide 128 bits of security. On the 5th of March of 2018, it was announced that Ascon was one of the seven finalists of the competition.

Currently, the fastest implementation of Ascon on the Cortex-A8 is an implementation optimized for 32-bit systems. This implementation utilizes bit interleaving for inexpensive rotates on 64-bit words whilst using 32-bit instructions [6]. In the past years, multiple studies have shown that the NEON instruction set can be used to great effect. [5] [15] [7]. We try to improve performance by using the NEON extension on the Cortex-A8. This allows us to compute on 128-bit registers.

We find that the critical function within the Ascon algorithm is the permutation and therefore optimize that function. While doing so, there are a few important points to consider:

- The balancing of the different processing units of the Cortex-A8.
- Dual issue cycles where possible to essentially execute instructions for free.
- Utilizing the 128-bits instructions of the NEON instruction set where possible.
- Prevent latency between instructions as much as possible by maximizing the amount of instructions between two dependent instructions, so that the processing unit does not stall.

We find that our implementation is substantially slower than the general 32-bit optimization, but still twice as fast as the reference implementation.

This is mainly because Ascon is not well suited for the NEON instruction set. For example, NEON lacks a rotate instruction and Ascon has 10 rotations in the non-linear layer. Furthermore there is little use for dual issue cycles, and due to data cohesiveness a large number of instructions in the substitution layer still operates on 64-bit vectors.

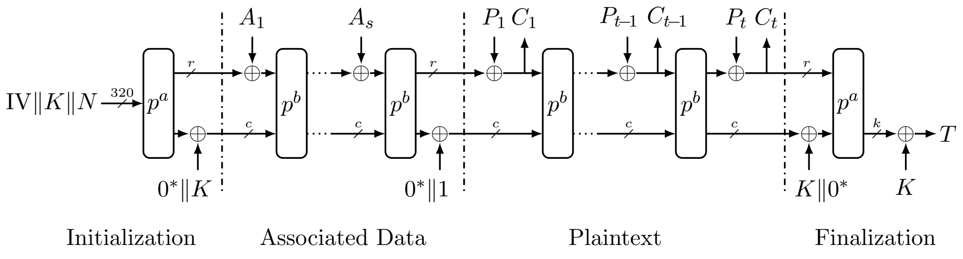
Chapter 2

Preliminaries

2.1 Ascon

Ascon [10] is a family of lightweight authenticated-encryption algorithms designed for the CAESAR [1] competition. In this thesis, we will focus on Ascon$_{12,6}$-128-64, which will be referred to as "Ascon". While parameters such as the key size are not set in stone, a key size of 128 bits is recommended as well as a public nonce of 128 bits. For more information about the default configuration, we refer to the Ascon specification document. All images used below are also taken from the submission paper [10].

Ascon utilizes a sponge-based mode of operation. For encryption, a 320-bit state is initialized with an IV, the key and public nonce. After 12 rounds of the permutation function, the algorithm starts with processing associated data, denoted as A_x. This process only manipulates the S-box. Ciphertext is extracted at the same time as the plaintext is processed, block by block. At last, a message tag of 128 bits is added.



As seen above, it is clear that Ascon has very little overhead. Only two base permutations are needed: one for initialization, and one for finalization.

2.1.1 Initialization

The 320-bit state is initialized by appending the IV with the secret key K and the used public nonce N. As per default configuration, the secret key and public nonce have a combined length of 256 bits. The IV makes up the remaining bits by describing the configuration of Ascon used via the key size k, rate r, initialization and finalization round number a, and intermediate round number b. The 32 remaining bits of the IV are set to 0. As per default configuration, we have a key size of 128 bits, a rate of 64 bits, initialization and finalization round number 12, and intermediate round number 6.

$$\begin{aligned} \mathrm{IV} &= k \,||\, r \,||\, a \,||\, b \,||\, 0^{288-2k} \\ \mathrm{IV} &= 16 \,||\, 8 \,||\, 12 \,||\, 6 \,||\, 0^{288-2k} \\ S &= \mathrm{IV} \,||\, K \,||\, N \end{aligned}$$

Then, a rounds of the permutation function are applied to the state. Finally, the last 16 bytes of the resulting state are xored with the key to complete the initialization.

2.1.2 Finalization

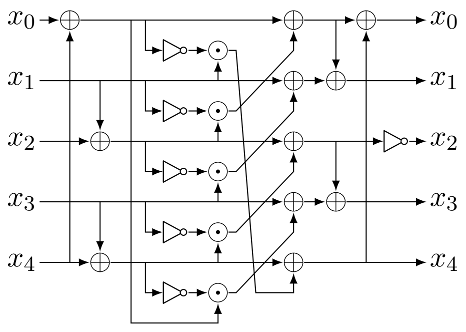
After encrypting all associated data and plaintext, the derived state is xored with the secret key K. The resulting state is manipulated by a rounds of the permutation function, after which the message tag is produced by xoring the final k bits of the state with K.

$$\begin{aligned} &S = p_a(S \oplus (0^r \,||\, K \,||\, 0^{c-k})) \\ &T = \lceil S \rceil^k \oplus K \end{aligned}$$

As a result, the ciphertext has length $|P| + |T|$.

2.1.3 Permutation function

The permutation function of Ascon consists of an addition of a round constant, a non-linear layer and a linear layer. For all layers, the 320-bit state is divided into five words x_0 through x_4 of 64 bits each. The addition of the round constant operates only on x_2. The figure below shows how each word is manipulated by the substitution layer.



The linear layer shown below diffuses 64-bit words by mixing the bits within each word.

$$\begin{aligned}
\Sigma_0(x_0) &= x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)\\
\Sigma_1(x_1) &= x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)\\
\Sigma_2(x_2) &= x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)\\
\Sigma_3(x_3) &= x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)\\
\Sigma_4(x_4) &= x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)
\end{aligned}$$

2.1.4 Security claims

Ascon claims to offer the following security:

Table 2.1: ASCON security claims

Requirement	Security in bits	
	ASCON-128	ASCON-128a
Confidentiality of plaintext	128	128
Integrity of plaintext	128	128
Integrity of associated data	128	128
Integrity of public message number	128	128

As most authenticated-encryption algorithms, Ascon leaks the length of the plaintext. The length of the ciphertext is simply the length of the plaintext with the message tag appended. If interested in the security analysis of Ascon, there are numerous papers available. For example, there is an analysis of Ascon's security claims made in their submission paper [16], a general cryptoanalysis of Ascon [9] and an analysis of the security of Ascon against state-recovery attacks [11]. The designers of Ascon list most relevant literature, both on security and implementation, on their website [1].

2.2 NEON architecture

Because the algorithm will be optimized for the Cortex-A8, we will specifically delve into the NEON implementation of this microprocessor. NEON is an SIMD (Single Instruction Multiple Data) extension for the ARM architecture. SIMD means that with a single instruction, the same operation can be performed on all elements of a vector. These elements are of course

[1] http://ascon.iaik.tugraz.at/analysis.html

also written to a vector, resulting for very fast data manipulation if the underlying data layout allows it. To illustrate this, the VADD instruction with 128 bit operands is able to perform 16 additions of 8-bit integers in a single cycle.

2.2.1 Register layout

The NEON architecture on the Cortex-A8 houses a total of 2048 bits in registers. It consists of 16 registers of 128 bits q_0 through q_{15}. However, these registers are made up of 32 registers of 64 bits d_0 through d_{31} in the way that d_0 and d_1 together represent q_0. Finally, these 32 registers are separated into 64 registers s_0 through s_{63}.

Table 2.2: How NEON-registers share memory space.

q_0				*128-bit register*
d_0		d_1		*64-bit registers*
s_0	s_1	s_2	s_3	*32-bit registers*

It is important to note that for most instructions, a singleword operation will take the same amount of NEON-cycles as a quadword operation. To fully utilize the NEON unit, it is crucial to use as much 128-bits instructions as possible.

2.2.2 ALU and Load/Store Unit

The NEON implementation consists of both an arithmetic unit and a load/-store unit. Both operate on vectors with a maximum size of 128 bits. These units execute independently of each other. Because of this design, we can create so-called "dual issue cycles" by using both units in a single NEON cycle. Besides dual issuing the ALU and the Load/Store unit, we can execute a regular ARM instruction at the same time as a NEON instruction. We do not find much use for the dual issue capability: NEON operates on a small state and we have no need for spills. However, we can still use it to copy the content of registers with a byte permute instruction whilst also executing an arithmetic instruction. This enables us to use instructions operating on 128 bits more than before, because we can move data around for free using byte permute instructions such as the vector-extract instruction.

2.3 Ascon128v12 optimized for 32-bit architectures

Ascon comes with multiple implementations. There are reference implementation available in C, Java and Python. For C, there also exists implementations optimized for 32-bit systems and 64-bit systems. When compared to the reference implementation, the 32-bit optimized version is about three times as fast on the Cortex-A8. A large part of the increase in performance is because of a technique called bit interleaving, which can be used to effectively compute rotations on 64-bit words with 32-bit registers. Consider 64-bit word W stored in 32-bit registers U and V, where U contains $W[i]$, $i \mod 2 = 0$ and V contains $W[i]$, $i \mod 2 = 1$. Then, rotating W by even number $2x$ bits means rotating both 32-bit words with x. Rotating W by odd number $2x + 1$ is slightly more difficult. Then, $U = ROT(V, x + 1)$ and $V = ROT(U, x)$. This principle is further outlined in a general sense in the Keccak implementation overview [6]. Since rotates of 64 bits with 32-bit registers are otherwise expensive and Ascon features 10 rotates in the permutation function, the gain in performance heavily outweighs the overhead caused by setting up this layout.

Chapter 3

Implementation

3.1 Bottleneck

As explained in Section 2.1, Ascon has small overhead. This means that to optimize the algorithm, one needs to only enhance the performance of the permutation function. This function accounts for over 90% of CPU when a kilobyte of associated data and message is processed.

Listing 3.1: Permutation function

```
void permutation(u8* S, int start, int rounds) {
        for (i = start; i < rounds + start; i++) {
                // addition of round constant
                x2 ^= ((0xfull - i) << 4) | i;
                // substitution layer
                x0 ^= x4;    x4 ^= x3;    x2 ^= x1;
                t0  = x0;    t1  = x1;    t2  = x2;    t3  = x3;    t4  = x4;
                t0 =~ t0;    t1 =~ t1;    t2 =~ t2;    t3 =~ t3;    t4 =~ t4;
                t0 &= x1;    t1 &= x2;    t2 &= x3;    t3 &= x4;    t4 &= x0;
                x0 ^= t1;    x1 ^= t2;    x2 ^= t3;    x3 ^= t4;    x4 ^= t0;
                x1 ^= x0;    x0 ^= x4;    x3 ^= x2;    x2 =~ x2;
                // linear diffusion layer
                x0 ^= ROTR(x0, 19) ^ ROTR(x0, 28);
                x1 ^= ROTR(x1, 61) ^ ROTR(x1, 39);
                x2 ^= ROTR(x2,  1) ^ ROTR(x2,  6);
                x3 ^= ROTR(x3, 10) ^ ROTR(x3, 17);
                x4 ^= ROTR(x4,  7) ^ ROTR(x4, 41);
        }
}
```

3.2 Lower bound

The permutation loop consists of three layers: the addition of round constant, the substitution layer and the diffusion layer. Besides the actual loop, it is also necessary to load and store the state to NEON registers.

The calculated lower bound depends on a number of factors. First of all, we assume that there is no latency. This means that for all instructions, the required source registers are available when the instruction is executed. Thus, no instruction will stall. Secondly, both the ALU and the load/store unit will be running in parallel at all times, potentially cutting the minimal amount of cycles in half. Lastly, we assume that all data is in the optimal location for 128-bit instructions when possible.

3.2.1 Initialization and finalization

The function is given a 320-bit state as argument. Before executing the loop, we need to load the state to NEON registers. This takes 4 cycles of the load/store unit. We also need to prepare the addition of the round constant: we store a constant `0xFU` to an ARM core-register and make sure the `d0` register consists of only zeroes. The regular ARN instructions and single NEON arithmetic instruction used for this can be executed in parallel, and thus do not influence the lower bound. Finally, we need to push the callee NEON registers. For finalization, we just need to store the state to r0, reverse the registers again and pop the callee registers. This also takes 7 cycles of the load/store unit.

3.2.2 Addition of round constant

The addition of the round constant is made up of four parts: subtracting the round number from a constant 0xFU, shifting the results by 4 bits to the left, performing a logical OR with the round number and finally an XOR with x_2 - bit 129 to 192. Bar the final XOR, we can use regular ARM-core instructions for this. However, moving the ARM-core register to a NEON register to perform the XOR on takes 2 full cycles of the load/store unit. The final XOR takes 1 cycle of the ALU.

3.2.3 Substitution layer

The substitution layer is made up of 22 arithmetic operations on 64-bit operands: 11 XORs, 6 INVs and 5 ANDs. All of these instructions take 1 cycle to execute. However, with NEON we can potentially halve the amount of instructions by using 128-bit operands instead. This comes down to a minimum of 6 XORs, 3 INVs and 3 ANDs performed by the ALU. By combining the remaining 64-bit XOR with the required XOR in the round constant, we can theoretically use a 128-bit instruction for those as well.

3.2.4 Diffusion layer

The diffusion layer consist of 10 rotates by an immediate, and 10 XORs. Unfortunately, the NEON instruction set does not include a rotate instruction. This means that for every rotate, we use two shifts and an XOR. This comes down to 20 shifts and 20 XORs in total. NEON only allows shifting on 128-bit registers if both 64-bit words are shifted by the same value, so those 20 shifts will take 20 cycles. However, the 20 XORs can potentially be reduced to 10 XORs on 128-bit operands. In total, this layer takes a minimum of 30 cycles of the ALU. We also note that none of the rotations used are by multiples of 8, which means that we are unable to use a byte permute instruction to imitate a shift.

3.2.5 Combining results

We have seen that ALU instructions dominate both regular ARM instructions and load/store instructions. Because of that, we will disregard all ARM and almost all load/store instructions: while we assume there is no latency, storing the state will always be the final instruction and requires its final source register to store in its 3rd cycle of execution. Thus, storing the state still takes at least 3 cycles because these registers are only available when the ALU is done with the permutation. Similarly, loading the first register with the first 64 bits of the state also takes 3 cycles. During those 3 cycles, no arithmetic instruction can be executed because they all depend on the state.

With that in mind, for each iteration of the loop we conclude a lower bound of 45 cycles: 2 for moving an ARM core register to a NEON register, 12 for all operations needed for the addition of the round constant and substitution layer, and 30 cycles for the diffusion layer. The overhead, that is the initialization and the finalization, takes a minimum of 6 cycles. This comes down to a total lower bound of $6 + 45n$, with n being the amount of rounds performed. This means 274 cycles per 6 rounds of permutation, and thus processing a byte every 34.25 cycles, bar non-loop overheads.

3.3 Deviation from lower bound

It should not be a surprise that the implementation deviates from the lower bound. Some latency is unpreventable because instructions depend on each other. An example of this is `VEOR q4, q4, q3 VMVN q5, q4`. Register `q4` is needed for the second instruction, but the result of the first instruction is not yet available when required. This causes the ALU to stall for several cycles.

3.3.1 Diffusion layer

Another example is the diffusion layer. In theory, the diffusion layer can be executed with 10 XORs and 20 shifts with every XOR being operated on two 128-bit registers. In practice, this is not possible due to the latency of the shift operations. Consider the following snippet:

Listing 3.2: Single rotate

```
VSHR.U64 d28, d1, #19
VSHL.I64 d29, d1, #45
VSHR.U64 d30, d1, #28
VSHL.I64 d31, d1, #36
 [...]
VEOR q14, q14, q15
VEOR d26, d28, d29
VEOR d1, d1, d26
```

It is obvious that the results of all shifts can be combined with a single XOR instruction on 128-bit operands, however the other two XORs are dependent on each other, so they cannot be executed at the same time with a single instruction. In practice, this means that we need 7 instructions to calculate a 64-bit word in the diffusion layer, versus 6 instructions we calculated in the above section: A net loss of 5 cycles per round. However, rather than calculating a single 64-bit word at a time, we can of course use a 128-bit register for the final XOR and perform two assignments in one instruction. This makes it possible to do 2 assignments in 13 cycles, losing only 1 cycle per two assignments. Another slight improvement we are able to do when we execute two assignments at the same time, is combining the intermediate XOR. However, since the required operands are in adjacent registers, we need to duplicate 3 registers to be able to combine the instructions. Due to the fact that we still need to wait on the intermediate computations, we frankly only have enough ALU cycles to burn to do this once without introducing additional latency. Ultimately, this means we can execute the diffusion layer in 32 cycles. See Appendix A.4 for the final isolated implementation of the diffusion layer.

3.3.2 Substitution layer

The substitution layer is a whole different beast altogether. Contrary to the diffusion layer, data is much more entangled. To put into perspective when which data is needed and what data an instruction depends on, we rewrote the atomic 64-bit operations, as outlined in the reference C implementation, to single static assignment form, which you can see in Appendix A.1.

This expression of the arithmetic operations was then used to try to create an order of instructions with as little latency as possible, whilst combining as many operations as possible to 128-bit instructions with the help of byte permutation instructions to move data around. To improve performance, the addition of the round constant was mixed in as well. The individual result can be seen in full at Appendix A.3. Of course, the actual code cannot be easily compared to the single static assignment form. The assembly was written with a couple of things in mind:

- Create a register alignment which allows as much 128-bit instructions as possible.
- Try to allow two instructions between computing the value of register and using it elsewhere again to prevent the stalling of the ALU.
- Move data around as soon as it is available using the VEXT instruction. VEXT takes elements from two source registers and outputs it to a destination register, so using VEXT with identical registers makes it a duplicate instruction.
- Allow a fluent transition from and to the diffusion layer, eliminating latency between the two phases and using the transition to prevent latency at the end of the substitution layer.
- Dual issue cycles where feasible.

3.3.3 Transposing into different layers

What is not shown in A.3 is how the substitution layer transposes into the diffusion layer. Because the last computations are highly dependent on each other, we combat the latency this would cause by starting to shift the final values of the substitution layer as soon as they are available for use.

Listing 3.3: An abstract image of transition between substitution and diffusion.

```
.looptop:
VEOR d14, d1, d5
VEXT.64 d6, d2, d2, #0
VMOV d0[0], r12
VEOR d8, d3, d0
VEXT.64 d7, d4, d4, #0
[...]
VMVN d3, d15
VEOR d5, d9, d2
VEOR q3, q3, q7
VEXT.64 d4, d7, d7, #0
VSHR.U64 d13,d3,#6
VSHL.I64 d15,d3,#58
VEOR d1, d14, d5
VEXT.64 d2, d6, d6, #0
**END OF SUBSTITUTION**
VSHR.U64 d12, d3, #1
VSHL.I64 d14, d3, #63
[...]
VEOR d1,d1,d26
VEOR q1,q1,q3
VEOR q2, q2, q15
[update loop var]
.bne looptop
VSTM.64 r0, {d1-d5}
```

Furthermore we ensure that no shifts of the diffusion layer stall, and that the first instructions of the substitution layer are not dependent on the last instructions of the diffusion layer. Coincidentally, we also compute them in the favorable order for storing when we exit the loop with as little latency as possible.

3.3.4 Round-constant preparations

All of the NEON-registers used during the permutation function are loaded with a result of an arithmetic operation or a byte permutation, bar the register used for the round constant. Because the round constant only uses 8 bits of the 64-bit register and the usual NEON load instructions would fill the whole register, we specify only to load a portion of the register. As a result, we need to wipe the remainder of this register before we use it. At the very beginning of the permutation, the load/store unit is saturated with loading the state. We use this time to dual issue clearing the `d0` register with an arithmetic instruction.

Chapter 4

Results

All of the algorithms participating in the CAESAR competition have been benchmarked using the SUPERCOP toolkit [2]. This is a framework used to benchmark all sorts of crypto algorithms, on all sorts of devices, provided by eBACS [3]. Because previous versions of Ascon have also been benchmarked on the Cortex-A8, as well as other finalists, it makes sense to use this framework because we can directly compare the outcome to previous results. We used the latest release of SUPERCOP at time of writing, which is the version released 18-12-2017. The full source code of our implementation is available on GitHub [1].

SUPERCOP benchmarks all implementations of an algorithm against each other, but also against the same implementation with different compilers and compiler flags. The compiler and compiler flags chosen by SUPERCOP for our implementation are `gcc -mcpu=cortex-a8 -mfloat-abi=hard -mfpu=neon -O3 -fomit-frame-pointer`, for the reference implementation they are `clang -O3 -fwrapv -march=armv7-a -mfloat-abi=hard -mfpu=vfpv3-d16 -fomit-frame-pointer -Qunused-arguments`, and for the implementation optimized for 32-bit systems (henceforth opt32) they are `gcc -fno-schedule-insns -O2 -fomit-frame-pointer`. The gcc version used was 9.4.2, for all implementations.

We publish the same categories in Table 4.1 as the categories listed on eBAEAD [4]. We use the same definition for 'long' as eBAEAD: 1/1920 of the difference in cycle counts between handling a 2048-byte message and handling a 128-byte message. Furthermore "$x + y$ *encrypt*" refers to encrypting a message with x bytes of associated data, and y bytes of message. The results are available in Table 4.1.

We note that our implementation is substantially slower than opt32, yet still considerably faster than the reference implementation. This is mainly because we are unable to utilize the NEON core's potential as much as we would like to:

[1]https://github.com/NoodleSkadoodle/Ascon128v12-NEON

- All of the rotates operate on 64-bit vectors, and furthermore there is no rotate instruction in NEON. Thus, every rotate takes at least 3 cycles using shifts and an XOR. As comparison, due to the bit-interleaving technique the same rotation can be performed using 32-bit instructions for free, as part of an arithmetic instruction.
- We find little use of dual issue cycles. None of the rotates are with multiples of 8, forcing us to use arithmetic instructions for all of them. We only use dual issue cycles to duplicate a register's content to another register, to be able to use instructions on 128-bit operands.
- Even with duplicating registers, there are still plenty of instructions that operate on 64-bit vectors. This is mainly because of the data cohesiveness. When we are unable to use 128-bit instructions without introducing additional latency, we are better off not using the 128-bit instruction at all.

Furthermore, it is also important to consider the other point of view: Perhaps our implementation is not typically slow, opt32 is just tremendously fast. We have calculated a lower bound of 34.25 cycles/byte, and opt32 outperforms even this lower bound. It would be interesting to try to combine both implementations, using regular ARM instructions for the rotates using bit interleaving, whilst using NEON for the substitution layer. However, because loading and storing from ARM-core registers to NEON registers is very expensive, and thus we do not think that this would improve performance over either a pure NEON or a pure regular ARM implementation.

Table 4.1: Benchmarking results

		Ascon128-Ref	Ascon128-Neon	Ascon128-opt32
long+0	encrypt	91.29	54.73	29.51
	decrypt	91.41	55.21	30.76
	forgery	91.41	55.22	30.75
long+long	encrypt	91.49	54.87	32.13
	decrypt	96.29	59.05	32.69
	forgery	91.26	55.36	32.70
0+long	encrypt	91.67	54.93	34.78
	decrypt	101.00	62.48	34.62
	forgery	91.01	55.50	34.63
1536+0	encrypt	94.29	56.79	30.89
	decrypt	94.49	57.26	30.19
	forgery	94.44	57.23	32.16
1536+1536	encrypt	93.00	55.87	32.83
	decrypt	97.76	59.97	33.42
	forgery	92.68	56.36	33.42
0+1536	encrypt	94.18	56.69	35.98
	decrypt	103.60	64.29	35.91
	forgery	93.45	57.60	35.90
64+0	encrypt	163.28	103.78	62.66
	decrypt	165.67	103.78	64.69
	forgery	163.90	103.28	64.53
64+64	encrypt	128.82	79.31	49.43
	decrypt	133.28	83.52	50.02
	forgery	126.48	79.05	49.86
0+64	encrypt	152.03	97.56	64.25
	decrypt	163.44	105.75	65.13
	forgery	149.69	96.72	64.88

Chapter 5

Related work

On the 5th of March 2018, the finalists of the CAESAR competition were announced. These are ACORN, AEGIS, Ascon, COLM, Deoxys-II, MORUS and OCB. For all candidates, benchmarks are made available online [3]. The comparison between those algorithms, Ascon-opt32 and our implementation is shown in Table 5.1. When multiple parameter configurations where available, we chose the primary recommendation listed in the submission documents. For example, the MORUS family has 3 configurations which differ in key size and state size. We compare with MORUS-1280-128, because it is the primary recommendation. Furthermore, since the online results were not completely up to date and sometimes incomplete, we ran the benchmarks using the SUPERCOP suite locally. Both OCB and COLM were not compatible with the framework and did not produce results. For OCB, benchmarks were available online but with large variance. Because of this unreliability, we chose not to include those results. As for COLM, no results were available online and this algorithm was thus also omitted. Our data is at the time of writing not yet included online, but the data dump is made available on GitHub [1].

Of the 6 other finalists, only MORUS [13] has an implementation optimized for NEON. This increased their performance, compared to the reference implementation, by 50% [8]. Despite the lack of existing implementations among the finalists, there are various papers that suggest crypto could benefit from using the NEON extension [15] [5] [7].

Our implementation of Ascon is quite slow compared to other algorithms. With short messages, like the 64+0 category, it performs very well and is only slower than MORUS. However, when the size of data increases, the other algorithms outperform Ascon. Only Deoxys-II [14] is slower. Deoxys-II performs notably slow when processing associated data: Where the algorithms are similar in speed in the long+0 category, Deoxys-II is almost twice as slow in 0+long. The same can be seen in the 1536+0 and 0+1536

[1]https://github.com/NoodleSkadoodle/Ascon128v12-NEON

categories. Deoxys-II is almost twice as slow in 0+1536, but performs similarly to Ascon128-NEON in 1536+0. While Ascon128-opt32 is substiantially faster than Ascon128-NEON, it is still significantly slower than ACORN and AEGIS in the long categories

5.1 Other Ascon implementations

The Ascon development team provides several implementations of the algorithm. There are reference implementations available for Python and Java, both reference and optimized implementations in C, and a hardware implementation. All of these are available on GitHub [2]. There are two optimized C implementations, besides the 32-bit implementation also one for 64-bit systems.

There are also other optimizations of Ascon not created by the development team. For example, Ascon has a hardware implementation suitable for RFID tags, Wireless Sensor Nodes and Embedded Systems that is able to process up to 5.5Gbit/sec of data (0.75 cycles/byte) [12]. For further reading on Ascon implementations, most related papers are linked on the Ascon website [3].

[2]https://github.com/ascon/ascon_collection
[3]http://ascon.iaik.tugraz.at/analysis.html

Table 5.1: Ascon vs other CAESAR finalists

		Ascon128-Neon	Ascon128-opt32	ACORN	AEGIS-128L	Deoxys-II-128	MORUS-1280-128
long+0	encrypt	54.73	29.51	18.25	19.01	55.75	5.42
	decrypt	55.21	30.76	19.25	18.98	55.71	5.39
	forgery	55.22	30.75	19.25	18.97	55.70	5.41
long+long	encrypt	54.87	32.13	18.50	19.50	84.04	4.41
	decrypt	59.05	32.69	19.50	19.33	83.98	4.39
	forgery	55.36	32.70	19.50	19.33	84.19	4.40
0+long	encrypt	54.93	34.78	18.76	19.92	112.34	6.04
	decrypt	62.48	34.62	19.78	19.68	112.19	6.02
	forgery	55.50	34.63	19.75	19.69	112.51	6.03
1536+0	encrypt	56.79	30.89	24.91	25.72	58.88	6.12
	decrypt	57.26	30.19	25.92	25.61	58.81	6.22
	forgery	57.23	32.16	25.93	25.61	58.86	6.26
1536+1536	encrypt	55.87	32.83	21.86	22.81	86.31	4.83
	decrypt	59.97	33.42	22.86	22.66	86.22	4.84
	forgery	56.36	33.42	22.87	22.65	86.40	4.85
0+1536	encrypt	56.69	35.98	25.42	26.55	116.13	6.94
	decrypt	64.29	35.91	26.43	26.34	116.03	6.91
	forgery	57.60	35.90	26.44	26.33	116.38	6.91
64+0	encrypt	103.78	62.66	177.97	178.88	130.47	108.55
	decrypt	103.78	64.69	179.12	178.52	130.50	111.09
	forgery	103.28	64.53	179.55	178.33	131.23	111.66
64+64	encrypt	79.31	49.43	99.00	99.55	138.06	44.60
	decrypt	83.52	50.02	100.13	99.25	138.11	45.69
	forgery	79.05	49.86	100.32	99.09	139.11	46.53
0+64	encrypt	97.56	64.25	178.43	179.28	203.59	84.84
	decrypt	105.75	65.13	179.69	179.31	203.73	87.53
	forgery	96.72	64.88	180.01	178.89	205.44	88.08

Bibliography

[1] Daniel Bernstein. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. `https://competitions.cr.yp.to/caesar.html`, 2018 (accessed April 18, 2018).

[2] Daniel Bernstein. System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives. `http://bench.cr.yp.to/supercop.html`, 2018 (accessed April 18, 2018).

[3] Daniel Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to/`, 2018 (accessed April 18, 2018).

[4] Daniel Bernstein and Tanja Lange. Measurements of CAESAR candidates, indexed by machine. `http://bench.cr.yp.to/results-caesar.html`, 2018 (accessed April 18, 2018).

[5] D.J. Bernstein and P. Schwabe. NEON Crypto. *In International Workshop on Cryptographic Hardware and Embedded Systems*, 7428:320–339, 2012.

[6] Daemen J. Peeters M. Van Assche G. & Van Keer R. Bertoni, G. Keccak implementation overview (v3.2). `https://keccak.team/files/Keccak-implementation-3.2.pdf`, 2012.

[7] J. López D. Câmara, C.P.L. Gouvêa and R. Dahab. Fast software polynomial multiplication on ARM processors using the NEON engine. *International Conference on Availability, Reliability, and Security*, 8128:137–154, 2013.

[8] Oussama Danba. Optimizing the authenticated cipher MORUS using NEON. Bachelor's thesis, Radboud University, Nijmegen, 2017.

[9] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Cryptanalysis of Ascon. In Kaisa Nyberg, editor, *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*, volume 9048 of *Lecture Notes in Computer Science*, pages 371–387. Springer, 2015.

[10] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round3/asconv12.pdf`, 2016.

[11] Ashutosh Dhar Dwivedi, Milos Kloucek, Pawel Morawiecki, Ivica Nikolic, Josef Pieprzyk, and Sebastian Wójtowicz. Sat-based cryptanalysis of authenticated ciphers from the CAESAR competition. *IACR Cryptology ePrint Archive*, 2016:1053, 2016.

[12] Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. Ascon hardware implementations and side-channel evaluation. *Microprocessors and Microsystems - Embedded Hardware Design*, 52:470–479, 2017.

[13] T. Huang H. Wu. The authenticated cipher MORUS (v2). Submission to the CAESAR competition: `http://competitions.cr.yp.to/round3/morusv2.pdf`, 2016.

[14] T. Peyrin J. Jean, I. Nikolic and Y. Seurin. Deoxys v1.v41. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round3/deoxysv141.pdf`, 2016.

[15] Hwajeong Seo, Zhe Liu, Johann Großschädl, and Howon Kim. Efficient arithmetic on ARM-NEON and its application for high-speed RSA implementation. *Security and Communication Networks*, 9(18):5401–5411, 2017.

[16] Serge Vaudenay and Damian Vizár. Under pressure: Security of caesar candidates beyond their guarantees. Cryptology ePrint Archive, Report 2017/1147, 2017. `https://eprint.iacr.org/2017/1147`.

Appendix A

Appendix

A.1 Single-static assignment translation of substitution layer

```
t0 = x0 ⊕ x4
t2 = x2 ⊕ x1
t4 = x4 ⊕ x3
n0 = ¬ t0
n1 = ¬ x1
n2 = ¬ t2
n3 = ¬ x3
n4 = ¬ t4
a0 = n0 ∧ x1
a1 = n1 ∧ t2
a2 = n2 ∧ x3
a3 = n3 ∧ t4
a4 = n4 ∧ t0
i0 = t0 ⊕ a1
i1 = x1 ⊕ a2
i2 = t2 ⊕ a3
i3 = x3 ⊕ a4
e4 = t4 ⊕ a0
e0 = i0 ⊕ e4
e1 = i1 ⊕ i0
e2 = ¬ i2
e3 = i3 ⊕ i2
```

A.2 Permutation function

```
.fpu neon
.text
.align 4
.global permutation
.global permutation
.type permutation STT_FUNC
.type permutation STT_FUNC
permutation:
permutation:
SETEND LE
VLDM.64 r0, {d1-d5}
MOV r3, #0xf
VMOV.I8 d0, #0

add r2, r1, r2

VPUSH {q4,q5,q6,q7}

._looptop:
sub r12, r3, r1
LSL r12, r12, #4
orr r12, r12, r1
#########################
# START OF SUBSTITUTION #
#########################
VEOR d14, d1, d5
VEXT.64 d6, d2, d2, #0
VMOV d0[0], r12
VEOR d8, d3, d0
VEXT.64 d7, d4, d4, #0
VMVN d2, d14
VEXT.64 d9, d5, d5, #0
VEOR q4, q4, q3
VMVN q5, q4
VEXT.64 d15, d8, d8, #0
VMVN q6, q3
VEXT.64 d3, d10, d10, #0
VAND q6, q6, q4
VAND d11, d11, d14
VAND q1, q1, q3
VEXT.64 d10, d3, d3, #0
```

```
VEOR q7, q7, q6
VEOR q3, q3, q5
VMVN d3, d15
VEOR d5, d9, d2
VEOR q3, q3, q7
VEXT.64 d4, d7, d7, #0
VSHR.U64 d13,d3,#6
VSHL.I64 d15,d3,#58
VEOR d1, d14, d5
VEXT.64 d2, d6, d6, #0

######################
# START OF DIFFUSION #
######################
VSHR.U64 d12,d3,#1
VSHL.I64 d14,d3,#63

VSHR.U64 d24,d2,#61
VSHL.I64 d25,d2,#3
VEOR q6, q6, q7
VSHR.U64 d26,d2,#39
VSHL.I64 d27,d2,#25

VSHR.U64 d28,d1,#19
VSHL.I64 d29,d1,#45
VEOR q12, q12, q13
VSHR.U64 d30,d1,#28
VSHL.I64 d31,d1,#36
VEXT.64 d7, d12, d12, #0

VSHR.U64 d16,d4,#10
VSHL.I64 d18,d4,#54
VEXT.64 d9, d13, d13, #0
VSHR.U64 d17,d4,#17
VSHL.I64 d19,d4,#47
VEXT.64 d6, d24, d24, #0

VSHR.U64 d20,d5,#7
VSHL.I64 d22,d5,#57
VSHR.U64 d21,d5,#41
VEXT.64 d8, d25, d25, #0
VSHL.I64 d23,d5,#23

VEOR q14, q14, q15
VEOR q12, q12, q13
VEOR q10, q10, q11
VEOR q8, q8, q9
VEOR q3, q4, q3
VEOR d26, d28, d29
VEOR d30,d16,d17
VEOR d31,d20,d21
VEOR d1,d1,d26
VEOR q1,q1,q3
VEOR q2, q2, q15

add r1, r1, #1
CMP r1, r2
bne ._looptop
VPOP {q4,q5,q6,q7}
VSTM.64 r0, {d1-d5}
# return with the link register
bx lr
```

A.3 Substitution layer and addition of round constant

```
sub r12, r3, r1
LSL r12, r12, #4
orr r12, r12, r1

VEOR d14, d1, d5
VEXT.64 d6, d2, d2, #0
VMOV d0[0], r12
VEOR d8, d3, d0
VEXT.64 d7, d4, d4, #0
VMVN d2, d14
VEXT.64 d9, d5, d5, #0
VEOR q4, q4, q3
VMVN q5, q4
VEXT.64 d15, d8, d8, #0
VMVN q6, q3
VEXT.64 d3, d10, d10, #0
VAND q6, q6, q4
VAND d11, d11, d14
VAND q1, q1, q3
VEXT.64 d10, d3, d3, #0
VEOR q7, q7, q6
VEOR q3, q3, q5
VMVN d3, d15
VEOR d5, d9, d2
VEOR q3, q3, q7
VEXT.64 d4, d7, d7, #0
VSHR.U64 d13,d3,#6
VSHL.I64 d15,d3,#58
VEOR d1, d14, d5
VEXT.64 d2, d6, d6, #0
```

A.4 Diffusion layer

```
VSHR.U64 d13,d3,#6
VSHL.I64 d15,d3,#58
VSHR.U64 d12,d3,#1
VSHL.I64 d14,d3,#63

VSHR.U64 d24,d2,#61
VSHL.I64 d25,d2,#3
VEOR q6, q6, q7
VSHR.U64 d26,d2,#39
VSHL.I64 d27,d2,#25

VSHR.U64 d28,d1,#19
VSHL.I64 d29,d1,#45
VEOR q12, q12, q13
VSHR.U64 d30,d1,#28
VSHL.I64 d31,d1,#36
VEXT.64 d7, d12, d12, #0

VSHR.U64 d16,d4,#10
VSHL.I64 d18,d4,#54
VEXT.64 d9, d13, d13, #0
VSHR.U64 d17,d4,#17
VSHL.I64 d19,d4,#47
VEXT.64 d6, d24, d24, #0

VSHR.U64 d20,d5,#7
VSHL.I64 d22,d5,#57
VSHR.U64 d21,d5,#41
VEXT.64 d8, d25, d25, #0
VSHL.I64 d23,d5,#23

VEOR q14, q14, q15
VEOR q12, q12, q13
VEOR q10, q10, q11
VEOR q8, q8, q9
VEOR q3, q4, q3
VEOR d26, d28, d29
VEOR d30,d16,d17
VEOR d31,d20,d21
VEOR d1,d1,d26
VEOR q1,q1,q3
VEOR q2, q2, q15
```