M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Efficient implementation of recent authenticated encryption schemes

Master Thesis

Kristian Kjældgaard Riisgaard
(s103626)

Kongens Lyngby 2017

DTU

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

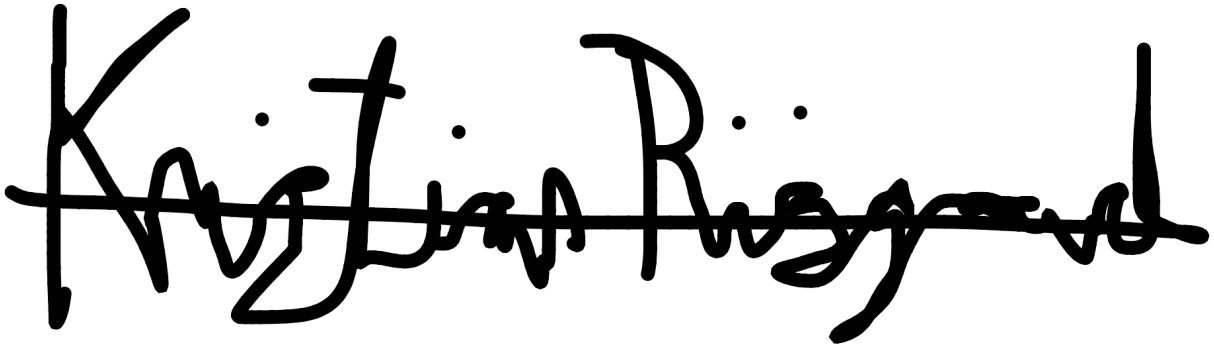
In this thesis, the implementation of CAESAR competition candidates COLM and Deoxys will be described. The implementations have been performed, both on an ARM mobile processor and three different x86 processors. The thesis will provide a description of the implementation, prediction of the scheme's performance, test of subprocesses within the schemes and tests of the schemes themselves. Both versions of COLM, as suggested by the authors will be tested, as well as the nonce misuse resistant version of Deoxys will be tested on three platforms. Both of these schemes are tested with AES as the cipher. Furthermore, one version of COLM and Deoxys will be implemented and tested on an ARM mobile processor, also with AES as the cipher, and Deoxys with the Skinny cipher will be implemented and tested on two computer processors.

The results of these tests will describe how the schemes perform, both with shorter and longer messages in mind, and will be used to suggest potential improvements to the schemes from a performance standpoint without loss of security.

Preface

This master thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a master's degree in mathematical modelling and computing.

Kongens Lyngby, August 9, 2017



Kristian Kjældgaard Riisgaard (s103626)

Acknowledgements

I would like to thank my supervisors Elmar Tischhauser and Stefan Kölbl for their help and patience throughout the process of writing this thesis.

I would also like to thank my friends Anders and Emil for help with my code, and finally I would like to thank my mom for proofreading my code.

Contents

CHAPTER 1

Introduction

The need for authenticated encryption arises from the need to achieve confidentiality, integrity and authenticity of data being transmitted. Confidentiality is commonly achieved through encryption schemes and an appropriate mode of operation. Data authentication is achieved through a message authentication code (MAC), and verifies that the data has not been changed as well the source of the message being who they claim to be. Authenticated encryption seeks to combine these goals.

Previously, attempts at using encryption and MAC separately could produce poor results, and simply putting together an encryption scheme and a MAC did not necessarily work well. An example of this is the CRIME attack against TLS, which was used against an authentication mechanism to get the information required to set up a connection pretending to be the victim [41]. The idea behind authenticated encryption is to gather encryption and authentication under one interface, which should refuse to decrypt the ciphertext if the authentication tag can't be verified.

Additionally, not all data transmitted needs to be kept confidential, and some data may even require to be sent in the clear, referred to as associated data (AD), which is how authenticated encryption with associated data (AEAD) came into consideration. There are no confidentiality goals for data sent in the clear, but integrity and authenticity are still required.

AE is required in any communication that required both confidentiality and authenticity, and the AD part enters where the requirement is that some data remain public but must still be authentic. For example in TLS where the header must remain public in order to transmit the message, which by itself requires confidentiality.

In 2013, the CAESAR competition was announced, which is a currently ongoing competition to design AEAD schemes. The CAESAR competition specifies three use cases [9], which the schemes attempt to meet in a prioritised order.

The need for optimized implementation arises in part from the fact that these schemes are intended to be used both in heavyweight, server-side settings, where SIMD instructions are used to obtain maximum performance from the processor, and on lightweight applications, where memory or processing power can be scarce. The focus of this thesis is optimized implementation on heavy, server-side processors.

In this thesis, I will implement two CAESAR candidates, COLM and Deoxys, and different variations thereof. I will compare them against each other on three different common processors, as well as one COLM and one Deoxys variant on a mobile processor. The schemes will be described, and tested in terms of performance.

CHAPTER 2

Authenticated encryption

In this chapter, the idea of authenticated encryption with associated data(AEAD) will be introduced. In general, an AEAD scheme takes in plaintext, associated data, a nonce and a key, and returns a ciphertext and returns a ciphertext and a tag.

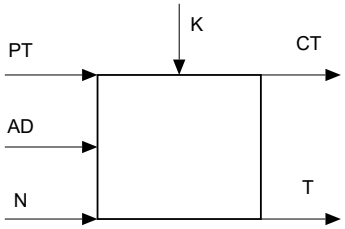


Figure 2.1: General depiction of an AEAD scheme.

The terms used in figure 2 will be described in the following section

2.1 Definition of terms

Several terms are used in the following section, which are defined in the following section.

2.1.1 Authenticated encryption Terms

1. Plaintext: A plaintext is raw information that can be directly read and understood by either a human or a computer. On figure 2, PT is the plaintext.

2. Ciphertext: A ciphertext is the result of a plaintext that has been modified, following an encryption algorithm, to appear unreadable to a human or a computer. On figure 2, CT is the ciphertext.

3. Key: A key is what allows someone to transform a plaintext into a specific ciphertext, and allows for the reverse transformation. The key must be kept secret from all illegitimate parties to ensure that messages can be sent confidentially between them. On figure 2, K is the key.

4. Associated data: Associated data is data treated by an AEAD algorithm that needs to remain authentic but does not require confidentiality, or even requires

that it is not confidential. An example of this is a TCP header, which described both the source and the destination among other information [44] . If this data is encrypted, any intermediate server will not know where to send the packet, and the packet will be lost. This data may still require authenticity. On figure 2, AD is the associated data.

5. Nonce: Nonce is short for ”number used once”, which is a rather descriptive name of how to correctly use nonces. Nonces are used in cryptography to add a random element during initialization of a cipher. The correct use of a nonce allows for a different keystream for each message, which in turn makes the cipher harder to break, ensuring confidentiality. On figure 2, N is the nonce.

6. Tag: A tag is used to ensure that the message is authentic, and is computed from either the plaintext or the ciphertext. Only those who have a legitimate key should be able to produce a valid tag. On figure 2, T is the tag.

7. Block: A block of data is 16 consecutive bytes of data.

2.1.2 Security goals

1. Integrity: Integrity covers that the data has not been modified by any unauthorized parties. In the scope of this thesis, it means that if the data is intercepted it can not be modified without the modification being detectable. Integrity for both plaintexts and ciphertexts exist.

    a) Integrity for plaintexts: It must be computationally infeasible for an attacker to produce a ciphertext that decrypts to a plaintext which has never been encrypted by the legitimate sender [7].

    b) Integrity for ciphertexts: It must be computationally infeasible for an attacker to produce a ciphertext that has not previously been computed by the legitimate sender, regardless of whether the plaintext has been encrypted before. Integrity for ciphertexts implies integrity for plaintexts [7].

2. Authenticity: Authenticity covers integrity, as well as being able to verify the source of the message.

3. Non-malleability: Non-malleability implies the inability to transform a ciphertext into a different ciphertext that will result in a similar plaintext. As an example, if a malleable cipher is used to send the message ”Transfer 1000€ to account 123”, it can be intercepted and changed to ”Transfer 9000€ to account 984”. This is done without knowledge of the key, only the structure of the message is known in this scenario. Malleable ciphers have their uses, in the sense that they are used in homomorphic encryption [20]. Indistinguishability are considered both under a chosen plaintext attack, and a chosen ciphertext attack.

4. Indistinguishability: Indistinguishability means that it is impossible to infer anything about a plaintext from a given ciphertext. This is one of the problems if a block cipher is used in ECB mode, the same plaintext will always encrypt to the same ciphertext, which can lead to inferences about the plaintext An example of this can be seen in figure 2.2. Indistinguishability are considered both under a known plaintext attack, and a chosen plaintext attack.

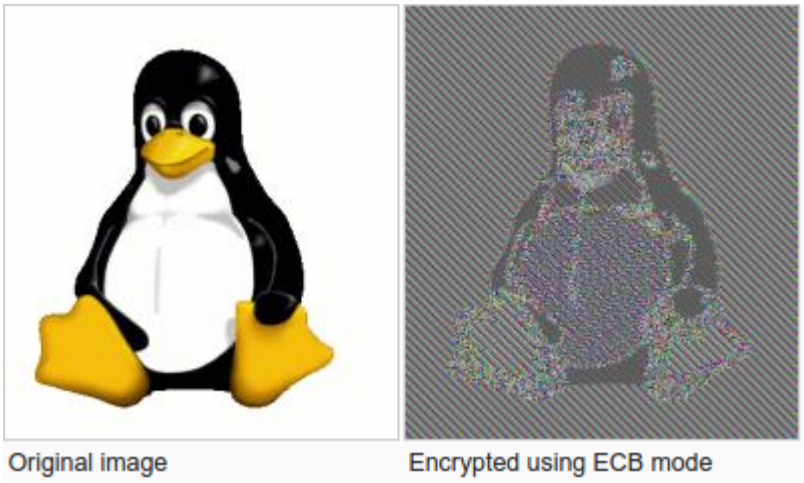


Figure 2.2: ECB penguin, the result of ECB encryption [38] .

5. Unforgeability: Unforgeability means that it is impossible to forge a valid signature on a given message. There are different kinds of forgery, and unforgeability for either one is the inability to commit such a forgery. The forgeries are listed in such an order that if such a forgery is impossible, the forgeries mentioned below are also impossible.

    a) Existential forgery: An adversary can create a valid message and signature pair, an arbitrary message. This message does not have to carry any specific meaning, for example the zero message. This is the weakest adversarial goal [50], and if a cipher is existentially unforgeable, the following forgeries are also impossible.

    b) Selective forgery: Selective forgery is when an adversary is capable of forging a signature for fixed message that the adversary has not chosen. This does not imply that the adversary can forge a signature for any message, just one message of a subset of valid messages [50].

    c) Universal forgery: An adversary can create a valid signature for any given message.

2.1.3 Attacks

There are two kinds of attacks which are of interest for the following definitions, chosen plaintext attack and chosen ciphertext attack.

1. Chosen plaintext attack: In a chosen plaintext attack (CPA), the attacker chooses a set of plaintexts and can receive the corresponding ciphertexts. An adaptive chosen plaintext attack is where the attacker uses the knowledge of a previously obtained ciphertext to choose the next plaintext [39].

2. Chosen ciphertext attack: In a chosen ciphertext attack (CCA), the attacker chooses a set of ciphertexts, and can receive the corresponding plaintexts. An adaptive chosen plaintext attack is where the attacker uses the knowledge of a previously obtained plaintext to choose the next ciphertext [39]. Any further reference to CPA or CCA assumes an adaptive attack.

2.1.4 Nonce misuse

Nonce-misuse, or nonce-reuse, can create a problem when using ciphers that are not designed to resist such a problem. Reusing a nonce leads to the same keystream being generated, unless the cipher is designed to account for such a problem, which in turn can make it easier for an adversary to derive information about the key, allowing said adversary to break the cipher. An example of this can be made from the nonce respecting (meaning that no security is guaranteed in case of nonce-reuse) mode of Deoxys, which has an encryption loop that looks as follows:

```
for j = 0 to l-1 do
  Checksum = xor(Checksum,M[j])
  C[j] = enc(0000||N||j,M[j])
end for
```

Listing 2.1: Nonce respecting mode of Deoxys.

In the given code example, listing 2.1, assume that two different messages, M_1 and M_2 are sent under the same nonce. Deoxys takes the nonce concatenated with a counter as a tweak input (will be further explained in the description of Deoxys). This tweak is used along with the key to generate the ”tweakey” used for encryption. If the key is the same, the expanded key will be the same as well. The counter doesn't change either, meaning that the cipher relies on the nonce to produce a different keystream. If the nonce is identical, every block is encrypted under the same keystream.

In order to resist nonce-misuse, certain steps can be taken. COLM is designed with nonce-misuse in mind, and Deoxys also has a nonce-misuse resistant mode. Using the nonce-misuse resistant mode of Deoxys as an example, what happens during encryption is

```
for j = 0 to l-1 do
  C[j] = xor(M[j], enc(  xor(1 || tag,j), (0^8 || N) )
```

```
3 end for
```

Listing 2.2: Nonce misuse resistant mode of Deoxys.

In this case, listing 2.2, the tweak input is the tag [35], which derives from the message and the associated data, while the nonce is the input being encrypted. The output of the encryption is in turn used to pad the plaintext. Now, the keystream not only relies on the nonce, but also the message and the associated data, meaning that if the nonce is reused, the keystream is still different. While keystreams can be accidentally identical, the only way to deliberately force this is to use the same associated data, the same message and the same nonce.

2.2 Construction of authenticated encryption

Authenticated encryption (AE) is when encryption and authentication is combined into one algorithm and one programming interface. The reason for this is that even with a cipher that has been proven safe and a strongly unforgeable MAC [6], the combination doesn't necessarily achieve confidentiality, integrity and authentication [8]. Furthermore, having authentication and decryption within the decryption algorithm allows for the rejection of a poorly constructed, in the sense that it has been malignantly modified, ciphertext.

Additionally, not all data needs to be sent in a confidential manner. In the scope of authenticated encryption, data that does not need confidentiality is referred to as associated data. Algorithms that work on both data that requires confidentiality and associated data are called AEAD algorithms, meaning authenticated encryption with associated data. The associated data does not require confidentiality, but still requires authenticity, which AEAD schemes must still provide.

There are three approaches to authenticated encryption, using the generic composition method where an encryption algorithm and a MAC are used together, which cover when the MAC is computed relative to the ciphertext, and whether it is encrypted. COLM and Deoxys are constructed as dedicated AEAD algorithms, and do not follow these constructions.

2.2.1 Encrypt-then-MAC

Encrypt-then-MAC (EtM) is when the plaintext is encrypted first, then the MAC is computed from the ciphertext. Encrypt-then-MAC is standardized by ISO [29], and has been added to TLS as an extension [21].

The main advantage of EtM is that integrity for ciphertexts and indistinguishability under a chosen plaintext attach are ensured. Those two properties are considered

desirable, because they imply indistinguishability and non-malleability under chosen ciphertext attack, which in turn imply the same properties under chosen plaintext attacks [6].

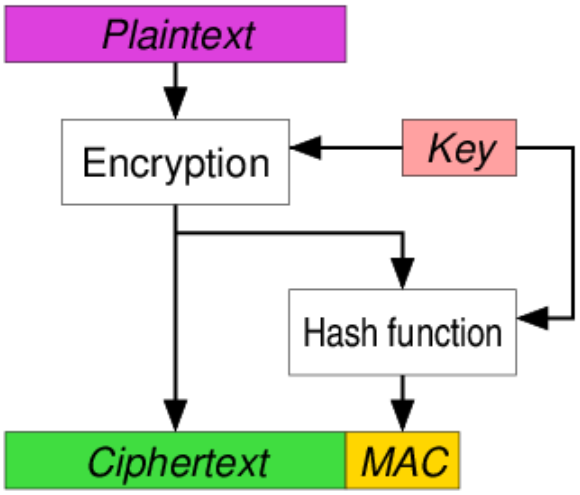


Figure 2.3: Picture of the EtM approach [46] .

2.2.2 Encrypt-and-MAC

Encrypt-and-MAC (EaM) is the approach where the MAC is generated from the plaintext, then appended to the encrypted message on its own. This is the approach used in SSH [7], despite not having been proven secure against indistinguishability under a chosen plaintext attack or integrity of ciphertexts. EaM does provide integrity of plaintexts.

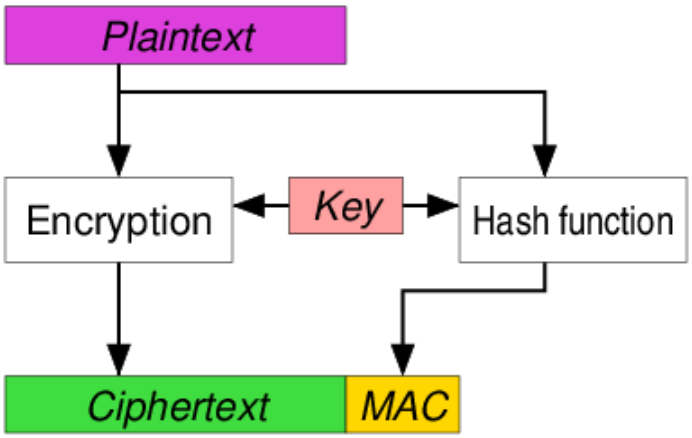


Figure 2.4: Picture of the EaM approach [45] .

2.2.3 MAC-then-Encrypt

MAC-then-Encrypt (MtE) is the approach where the MAC of the plaintext is computed, and both the MAC and the plaintext are encrypted afterwards. MtE provides integrity of plaintexts and indistinguishability under a chosen plaintext attack, but not non-malleability under a chosen plaintext attack, which implies that indistinguishability under a chosen ciphertext attack and integrity of ciphertexts are not provided either.

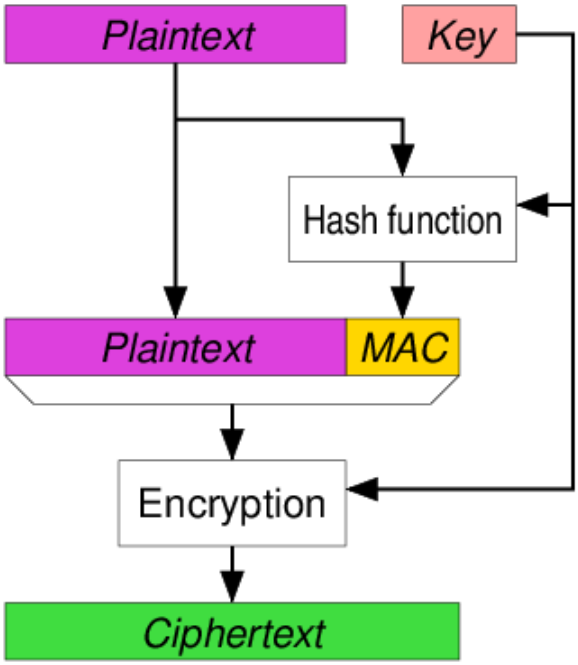


Figure 2.5: Picture of the MtE approach [47] .

The NIST recommended modes for authenticated encryption are CCM and GCM [18] [19] follow the generic composition [43] [23]. Neither of these are nonce misuse resistant [22] [37]. The performance of an optimized version of CCM is found to be 5.1 cycles per byte, and an optimized version of GCM is found to be 1.14 cycles per byte [12] for a message size of 2048 bytes. The ciphers investigated in this thesis are dedicated AEAD ciphers, both of which are nonce misuse resistant.

2.3 CAESAR

The CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [14] competition has the purpose of identifying a portfolio of authenticated schemes that offer advantages of AES-GCM and are suitable for widespread adoption. Fifteen ciphers have made it into the third round of the competition, and I have chosen to focus on COLM and Deoxys.

2.4 Requirements

The CAESAR competition has a set of requirement for the submitted ciphers. The requirements for input to the encryption function are

- Variable length plaintext. Requires confidentiality and integrity.
- Variable length AD. Requires integrity
- Fixed length nonce. A length of zero is permitted.
- Fixed length private nonce. A length of zero is permitted. Also requires confidentiality.
- Fixed length key. Requires confidentiality

Those five inputs are used to compute a variable length ciphertext which includes the authentication tag. The decryption algorithm must take in the key, the ciphertext and every input that did not require confidentiality. From those inputs, it must be able to compute the plaintext and whether it is authentic. In case of a non-authentic plaintext, the algorithm must reject it and state that the message has not been verified. The practical limitation on the ciphertext is 2^{64} bits, due to limitations in the API specified in the CAESAR competition [14].

Not all algorithms use a private or a public nonce. Both COLM and Deoxys use private nonces of length zero [2] [35].

A submission may contain multiple ciphers with slightly different properties, which is called a family of ciphers. The different properties can be key size, use of intermediate tags, resistance against nonce misuse.

Another requirement is that each recommendation must specify in a prioritized order, which of the three use cases the ciphers target [9]. The three use cases are lightweight applications, high-performance applications and defence in depth.

Lightweight applications The critical point in lightweight application is that the algorithm can fit either into a small area, or that the code is not that big, such that an 8-bit CPU can read and execute it without problems. Aside from that, other properties considered desirable are hardware performance, speed, and the ability to naturally protect itself from side channel attacks, meaning attacks based on information obtained from the physical implementation of the system. Furthermore, this use case is primarily geared towards short messages.

High-performance applications The critical point in high-performance applications is efficiency on 64-bit CPU's and/or dedicated hardware, and efficiency on 32-bit CPU is also considered a desirable point. Another desired property is that for any message and associated data of a given length, the running time should be constant. In this case, longer messages are primarily considered.

Defence in depth The critical point in defence in depth, is that the cipher should still provide authenticity if nonces are misused. Other desirable properties are limited

privacy damage from misused nonces, as well as limited confidentiality damage and authenticity in the case of released plaintexts, which means unverified plaintexts being made available to the attacker. An example of released plaintexts is the padding oracle attack [54], where the attacker can retrieve data about the padding of a ciphertext. This can be used to guess the plaintext within a fairly short time.

2.5 Implementation in practice

In practice, memory management is important for how efficiently any implementation performs. The memory structure has three levels of caches and the RAM, where all data is assumed to be stored before starting to encrypt, and is assumed to be stored until it's written to the disk.

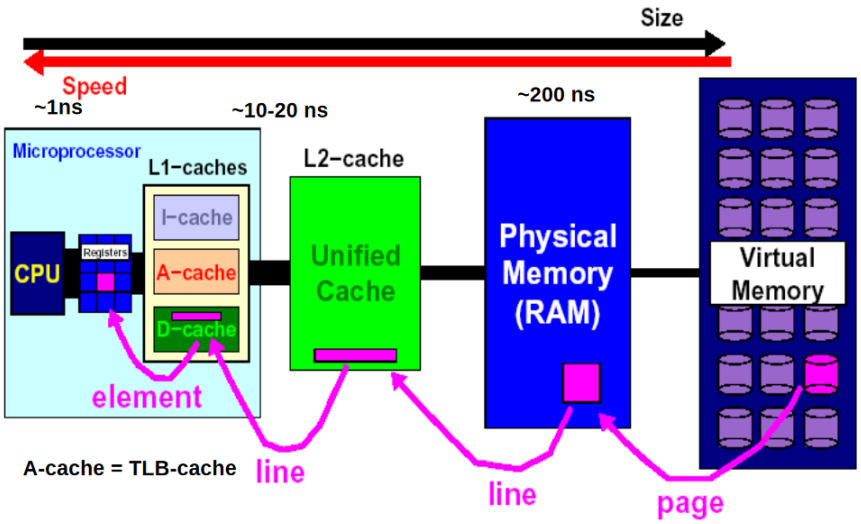


Figure 2.6: The Memory Hierarchy [17] .

This means that memory use must be taken into account. In the scope of high performance applications it must be done to ensure as high performance as possible by ensuring that the processor loads as little as possible from slow memory, and lightweight applications may not have that much memory to work with at all. It follows from the specifications that plaintext, ciphertext and associated data are of variable length.

While not a specific requirement, ensuring that the cipher uses constant memory internally reduces the risk of extra loads from slow memory, which can affect performance. Constant memory only means independence from variable length data, and there can be tradeoffs in terms of using a bit more memory to achieve a better performance. Consider the following code example:

```
1 __m128i deltaM,deltaC;
2 __m128i W,mes,Y;
```

Listing 2.3: Initialization of sequential code.

```
__m128i deltaM[4],deltaC[4];
__m128i W[4],mes[4],Y[4];
```

Listing 2.4: Parallel.

Both are examples of the initialization of the necessary variables in the $COLM_0$ scheme, where the encryption after the initialization shown in listing 2.3 runs sequentially, and the encryption after the initialization shown in listing 2.4 runs in parallel. Listing 2.4 uses four times as much memory, keeping the message and every variable in arrays, while listing 2.3 only uses single values. The extra memory used by listing 2.4 allows for substantially faster computation, as can be seen in table 2.5, as it allows for the processor to use the pipeline to perform the computations in parallel. Both tests have been run on the Skylake processor, and the performance numbers are the average of 1000 batches of 100000 trials.

	$COLM_0$ serial	$COLM_0$ parallel
Performance	3.37 cpb	2.23 cpb

Table 2.1: Performance numbers from serial versus parallel implementations of $COLM_0$. 2064 byte plaintext, 144 byte associated data, nonce and key as inputs. The test of $COLM_0$ running in parallel is described in section 8.3 and the test of $COLM_0$ running in serial is described briefly in appendix D.2.

On every normal weight processor that the schemes have been tested on, the cache sizes have been bigger than the amount of memory used by the scheme internally, which is why this point has not been given much thought in the implementations, meaning that every implementation has been designed to ensure maximum utility from the pipeline.

CHAPTER 3

Tweakable block ciphers

Tweakable block ciphers are a kind of block cipher that do not only take a plaintext and a key input, but also a tweak input, which is an additional input that may be public.

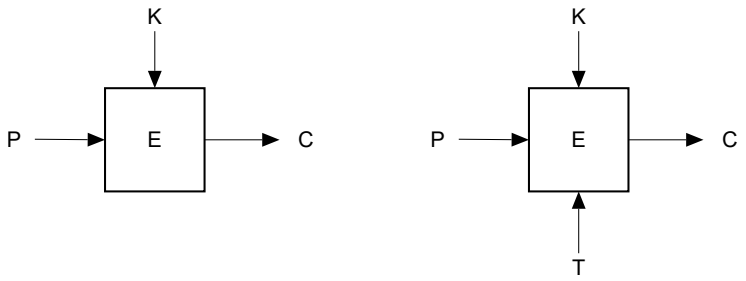


Figure 3.1: Regular block cipher (left) and tweakable block cipher (right).

Tweakable block ciphers have existed since the AES competition by NIST, where the Hasty Pudding Cipher had an additional input called ”spice”, which serves the function of a tweak in a current tweakable block cipher [34]. Of particular interest in the scope of this thesis, is the Tweakey framework, which the Skinny cipher [5] and the Deoxys AEAD scheme [35] follow.

3.1 The Tweakey framework

The Tweakey framework allows for a tweak input of almost any given length, which can be added to almost any key-alternating, meaning a cipher where each round has a different key input, block cipher. The term ”tweakey” itself refers to an input that can be both tweak or key input without distinction. It is a framework to build a tweakable block cipher with a t-bit tweak and a k-bit key, and consists of the n-bit internal state s and the $(t + k)$-bit tweakey state tk. A non-tweakable cipher, such as regular AES, can be considered a tweakable cipher under the tweakey framework, with a tweak length of 0 bits. The cipher is composed of r rounds of the following steps.

- A subtweakey extraction function, called g
- An internal state (s) update permutation, called f
- A tweakey state update function, called h

Using AES as an example, the extraction function g is simply xor, and is performed in the AddRoundKey step. The internal state update are the remaining AES steps, and h is the key expansion algorithm. For another example of the Tweakey framework, see the Deoxys chapter. Tweaks and keys are not distinct under the framework itself, but in certain ciphers, such as Deoxys, a clear distinction can be made.

Certain issues may occur under the Tweakey framework, such as identical tweak and key input cancelling each other out if identical h permutations are used, or certain tweak input cancelling each other out, for example if g is xor. The Superposition Tweakey (STK) construction takes that into account by adding an additional step before h, which is the multiplication by a constant α, which differs for each update function. In the tweakable ciphers Deoxys or Skinny, this is handled by applying an LFSR to each byte of a partial tweakey, a partial tweakey being tweakey material of the same size as the internal state. An example of the STK construction with three partial tweakeys can be seen in figure 3.1 .

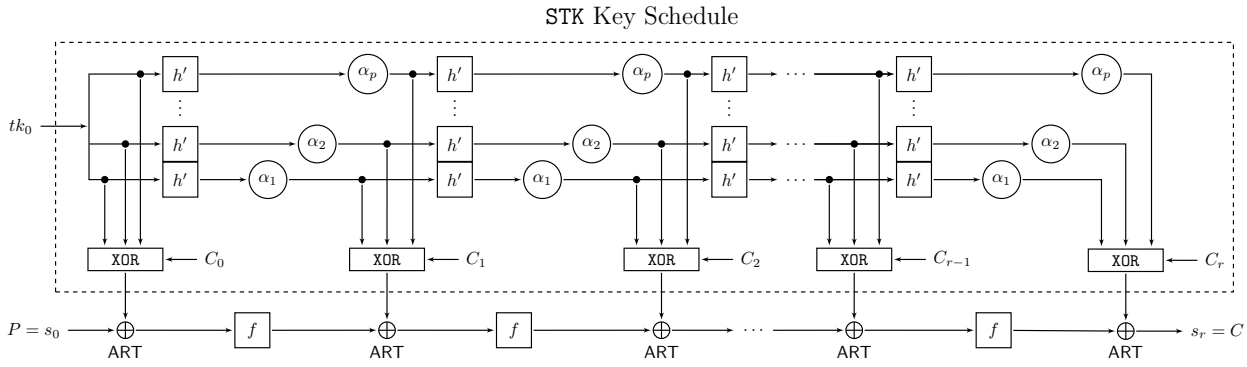


Figure 3.2: Example of STK construction with three partial tweakeys [30] .

CHAPTER 4

Skinny

Skinny is a tweakable lightweight block cipher, designed for lightweight application, with an attempt to find the spot where good performance meets strong security. [5]

4.1 Description

Two versions of Skinny exist, a 64-bit version and a 128-bit version, with variable tweak sizes of $|k|$, $2|k|$ and $3|k|$, where $|k|$ is the key size. The number of rounds used in encryption depends on the tweakey size. When writing the state of Skinny as a 4×4 matrix, it is interpreted row-wise, as opposed as the column-wise interpretation used in AES. The focus of this thesis is on Skinny with a 128-bit key and a 128-bit tweak.

$$\begin{bmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{bmatrix}$$

The rounds of Skinny consist of five operations: SubCells, AddConstants, AddRoundTweakey, ShiftRows and MixColumns.

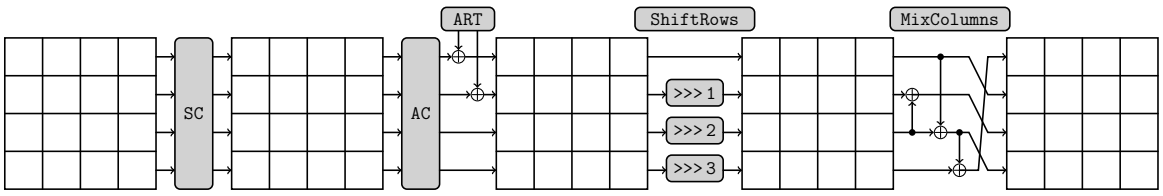


Figure 4.1: Skinny Round Function [32].

4.1.1 SubCells

The Sbox for Skinny with a 128-bit takes in a byte input, and switches the bits by the following schematic, where x_0 is the most significant bit, and x_7 is the least significant bit.

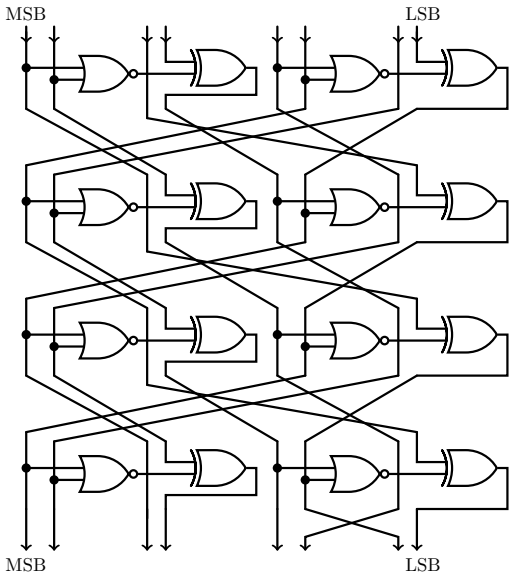


Figure 4.2: Skinny 8-bit Sbox [31].

From an implementation standpoint, the Sbox can be described by the following formula, where x'_n is the updated value of x_n, and replaces it immediately.

$$\begin{aligned}
t_0 &= x_7 \oplus \neg(x_4 \vee x_5) \\
t_1 &= x_1 \oplus \neg(x_5 \vee x_6) \\
x'_1 &= x_3 \oplus \neg(x_0 \vee x_1) \\
t_2 &= x_2 \oplus \neg(x'_1 \vee t_0) \\
t_3 &= x_6 \oplus \neg(t_0 \vee x_4) \\
x'_6 &= x_0 \oplus \neg(t_1 \vee t_2) \\
x'_0 &= t_2 \\
x'_2 &= t_0 \\
x'_3 &= x_4 \oplus \neg(t_2 \vee x'_1) \\
x'_4 &= t_3 \\
x'_7 &= x_5 \oplus \neg(t_3 \vee x'_6) \\
x'_5 &= t_1
\end{aligned} \tag{4.1}$$

This construction ensures that any temporary variables that need to be reused are maintained for multiple uses, the temporary variables are kept explicit such that memory use is easy to compute, and the number of temporary variables is kept low, the new values for x_1 and x_6 were originally designed as temporary variables, and later fixed immediately.

4.1.2 AddConstants

In this step, the following matrix of constants is added to the state

$$\begin{bmatrix} c_0 & 0 & 0 & 0 \\ c_1 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The constant c_2 is defined as $0x2$ for the 4-bit version, and $0x02$ for the 8-bit version of Skinny. c_0 and c_1 are defined from a 6-bit LFSR, which is defined as follows:

$$(rc_5||rc_4||rc_3||rc_2||rc_1||rc_0) \rightarrow (rc_4||rc_3||rc_2||rc_1||rc_0||rc_5 \oplus rc_4 \oplus 1) \tag{4.2}$$

The initial stage of the LFSR is $(0||0||0||0||0||0)$, and is updated before any given round. c_0 and c_1 are set after the update, using the following formula for the 128-bit version.

$$(c_0, c_1) = ((0||0||0||0||rc_3||rc_2||rc_1||rc_0), (0||0||0||0||0||0||rc_5||rc_4)) \tag{4.3}$$

The AC matrix is xor'ed to the state.

4.1.3 AddRoundTweakey

AddRoundTweakey follows the tweakey framework, and is updated in the following way for each round tweakey, where the LFSR is not applied to the tweakey material called $TK1$. In the case used in the thesis, there are two tweakey blocks per round, $TK1$ and $TK2$.

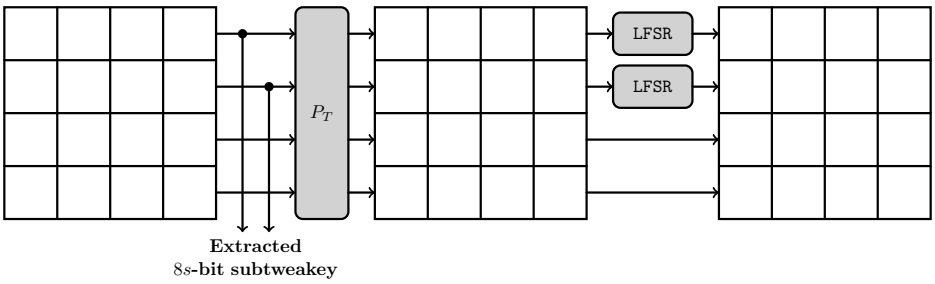


Figure 4.3: Skinny Tweakey Schedule [33].

The LFSR looks as follows, and is also used in the Deoxys scheme

$$(x_7||x_6||x_5||x_4||x_3||x_2||x_1||x_0) \rightarrow (x_6||x_5||x_4||x_3||x_2||x_1||x_0||x_5 \oplus x_7) \tag{4.4}$$

And P_T is a byte permutation, where the bytes are moved in the following way, where the bytes in the original array are moved to to the following positions in the new array.

$$P_T = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7] \tag{4.5}$$

Only the most significant half of the bits are xor'ed to the state. The P_T permutation ensures that all key material is used by moving the unused key material into the slots where it will be used, and moving the used key material into the slots where it won't be used. The most significant bits, which are to be used in the next rounds are subject to the LFSR.

4.1.4 ShiftRows

The ShiftRows operation is similar to the operation by the same name in AES, except that the rows are shifted right by 0, 1, 2 and 3 places respectively, as shown in 4.1. Furthermore, since the interpretation of a state in Skinny is as a row-wise matrix, the ShiftRows operation can be described with the following permutation.

$$SR = [0, 1, 2, 3, 7, 4, 5, 6, 10, 11, 8, 9, 13, 14, 15, 12] \tag{4.6}$$

4.1.5 MixColumns

The MixColumns operation is another operation, which is similar to the operation by the same name in AES. Each column in the internal state is multiplied by the MC matrix.

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

4.2 Bitslicing

The Skinny implementation that has been provided to me uses a method called bitslicing to achieve good parallel performance. Bitslicing was used in cryptography in 1997 for an efficient implementation of DES [11], and the AES finalist [13] Serpent was designed to allow fast decryption by this method [1].

Bitslicing is the process of rearranging the data in such a way that operations are performed on bytes or registers rather than bits, and can save computations depending on the implementation. The idea is that the bits are moved from from their original positions, and reorganized into new positions such that it is easier to run calculations on them. In figure 4.2, eight bytes are rearranged such that the first bit from each byte at the top becomes the first byte at the bottom, and so on. This is a simple depiction of the process, and can be a bit more complex in practice.

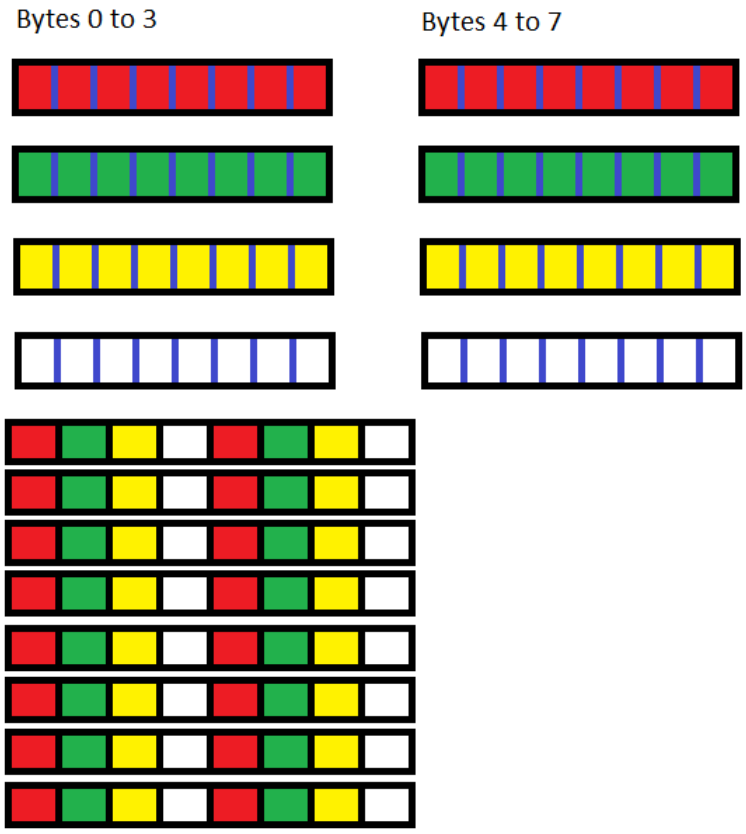


Figure 4.4: Conceptual depiction of bitslicing.

The advantage of bitslicing is that it allows for a lot of computations in parallel. For instance, the Skinny implementation that has been provided to me processes 64 blocks in parallel. The disadvantage of bitslicing is that the original array needs to be rearranged on a bit level. For example, the sbox takes in eight 256-bit registers and runs the sbox operation on these, rather than on the bits themselves. In this case, every register represents a bit position in the original setting, and contains bits from 32 different blocks.

In the Skinny cipher, the following bitslicing method is used to transform an array of 64 blocks of 16 bytes each into an array of 32 times 32 bytes. The following transformation describes how each individual bit will move around, where the first index describes row number, second index is column number and b_n means that the bit is moved to position n in the byte in that position.

$$in[c][p]b_a \rightarrow out \left[a + 8\left\lfloor\frac{p}{4}\right\rfloor\right]\left[(p \mod 4) + 16(c \mod 2) + 4\left\lfloor\frac{c}{16}\right\rfloor\right] b_{\lfloor\frac{c}{2}\rfloor} \tag{4.7}$$

Deriving this formula has been done through setting every bit to 0, setting single bits to 1, tracing how the bits moved throughout the system. Afterwards, it has been tested by generating random numbers and verifying that the formula predicts bit movement correctly.

CHAPTER 5

Vector instructions

Doing an optimized implementation of AEAD schemes relies on using processor specific vector instructions; specifically single instruction, multiple data (SIMD) instructions. SIMD instructions allow for data packed in to specific data structures to be processed in parallel. These specific data structures will be referred to as vectors. These vectors can be used to perform operations on data in parallel by applying the same instruction to a vector, which contains different data points.

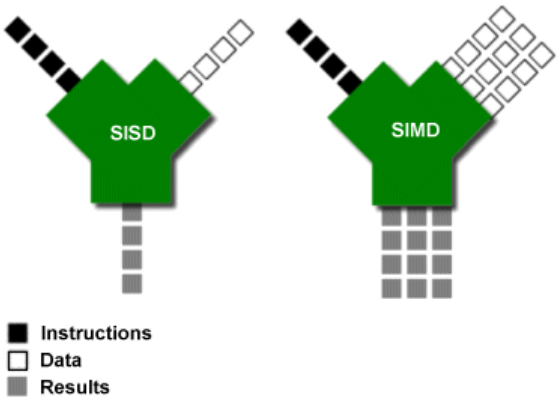


Figure 5.1: Depiction of SIMD processors [52] .

For a simple example, consider the addition of two lists of 16 eight-bit integers. The naive way of adding such two sets would be to simple iterate through the lists, adding the numbers and storing the results sequentially. SIMD instructions allow the programmer to do these instructions in parallel, by storing the two lists in their respective vectors and adding them, an operation that is visualized in figure 5.

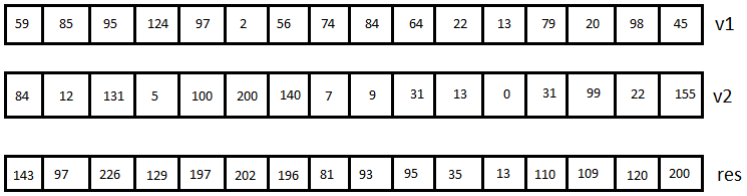


Figure 5.2: Depiction of SIMD addition operation.

SIMD instructions aren't limited to arithmetic or logical operations, but can also be used to do more advanced operations, such as encryption. Certain Intel and ARM processors allow the users to access AES encryption functions [27], which go through the process one AES round, or in the case of ARM, one AES round except the MixColumn step [3]. This has the advantage of being the optimal implementation in terms of performance. It also means that the cipher is implemented as a black box, preventing a faulty homemade implementation.

In both the Intel and the ARM SIMD instruction sets, certain operations exist that can be parallelized using the instruction pipeline. The important factor is to understand the terms latency and throughput. Latency describes how long it takes from a command is issued to the processor, until it has finished. Throughput describes the time it takes from a command is issued to the processor, until it can begin processing a new command. For example, the AES round operation on an Intel Skylake processor has a latency of four, but a throughput of one, allowing for parallel processing in the manner depicted in figure 5.3.

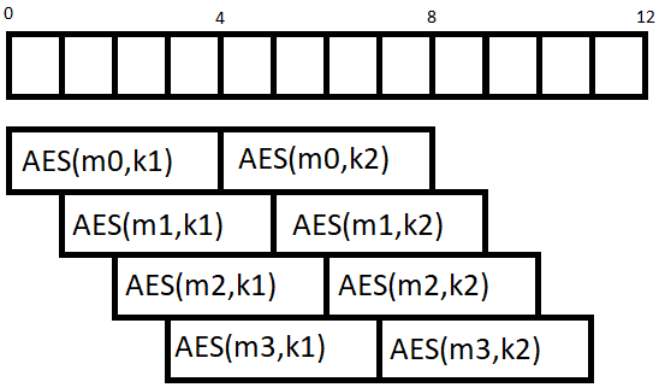


Figure 5.3: Depiction of pipelined operations.

The Intel pipeline is different from the ARM pipeline, in the sense that cryptographic instructions have their own pipeline, while all instructions share the same pipelines on ARM. While it's fairly easy to determine how many encryption operations that need to be grouped together in an Intel implementation, the ARM pipeline being structured the way it is, means that it can vary how many encryption operation that need to be grouped together to achieve optimal performance.

In both the Intel and the ARM instruction set, the encryption and decryption operations can run on a pipeline, meaning that to achieve optimal performance, the order of when the instructions are called is important. Consider the following example:

```
1 for(i=0;i<length;++i{ // LOAD and STORE call the load and store intrinsics,
    while ENC performs one round of encryption
2   LOAD(pt+i,m);
3   for(j=0;j<11;++j){ // Compiler will unroll this loop
4     ENC(m,k[j]);
```

```
  }
  STORE(ct+i,m);
}
```

Listing 5.1: Sequential code.

```
for(i=0;i<length;i+=4{ // LOAD and STORE call the load and store intrinsics,
     while ENC performs one round of encryption
  LOAD(pt+i,m[0]);
  LOAD(pt+i+1,m[1]);
  LOAD(pt+i+2,m[2]);
  LOAD(pt+i+3,m[3]);
  for(j=0;j<11;++j){ // Compiler will unroll this loop
    ENC(m[0],k[j]);
    ENC(m[1],k[j]);
    ENC(m[2],k[j]);
    ENC(m[3],k[j]);
  }
  STORE(ct+i,m[0]);
  STORE(ct+i+1,m[1]);
  STORE(ct+i+2,m[2]);
  STORE(ct+i+3,m[3]);
}
```

Listing 5.2: Encryption in parallel.

While the two sets of codes do the exact same thing, the listing 5.2 code is much more efficient. This is because the encryption function runs on the pipeline, while the first block is processing, the processor can begin processing the second block. How many blocks can be pipelined depends on the processor. The above example with four blocks allows for full pipeline utilization on an Intel Skylake [27] processor, as it can begin processing the fourth block while the first block is being processed.

One important thing to note when working with SIMD vectors is that some operations assume that the data within the vector consists of independent elements. Performing certain operations, such as addition and multiplication, can cause overflows within a data spot need to be accounted for.

The solution is to change the interpretation of the data stored in the vector, and to account for the data that would otherwise be lost. For example, a 128-bit vector can be changed from being interpreted as sixteen 8-bit elements or two 64-bit elements. Some operations, such as a vector-wide bitshift, which part of the field doubling in the COLM algorithm, requires use of permutations to account for overflows within the registers.

```
__m128i mul2(__m128i x)
{
  const __m128i red  = _mm_set_epi64x(0x8700000000000000,0x0000000000000000)
     ;
  const __m128i ZERO = _mm_setzero_si128(); // Set all elements to zero
  __m128i mask = _mm_cmpgt_epi32(ZERO,x); // Treats vector as a vector with
     four 32-bit elements
  mask = _mm_shuffle_epi32(mask,0xff);  // Same as above
```

```
7   __m128i x2 = _mm_or_si128(_mm_slli_epi64(x,1),_mm_srli_epi64(
        _mm_slli_si128(x,8),63));  // Varuíous interpretation of vectors used
        here
8   return _mm_xor_si128(x2,_mm_and_si128(red,mask));
9 }
```

Listing 5.3: Intel intrinsics constant time field doubling code [53].

This code takes in an 128-bit vector, and doubles it within a field. The flexibility of Intel's 128-bit vectors allow for operations to assume that the vectors have different size registers, as well as whether an element is signed or unsigned. The third and fourth lines treat the vector as if it has four 32-bit elements, while the `epi64` commends on the fifth line treats it as having two 64-bit elements.

What happens is that the input vector is treated as a signed integer on the third line. If it is less than zero, meaning that the most significant bit is 1, the field polynomial is added in the final operation. The remaining code shifts all bits left, while ensuring that the bits that should carry over to a different register by the shift do so. There are some distinct differences in the ARM implementation.

```
1  u128 mul2(u128 x){
2    u128 red = {0x00,0x00,0x00,0x00, 0x00,0x00,0x00,0x00, 0x00,0x00,0x00,0x00,
         0x00,0x00,0x00,0x87};
3    u128 sm =  {0x01,0x02,0x03,0x04, 0x05,0x06,0x07,0x08, 0x09,0x0a,0x0b,0x0c,
         0x0d,0x0e,0x0f,0x0f};
4    u128 ca = SR(x,7); /* Get most significant bit of each element */
5    u128 r_shift = {0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0
        xff, 0xff,0xff,0xff,0x00};
6    u8 cmp[16] = {0x00,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff
        ,0xff, 0xff,0xff,0xff};
7    u128 CMPT = {0x00,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0
        xff, 0xff,0xff,0xff};
8    u128 check = vcgtq_u8(ca, CMPT); /*Check if ca > CMPT, which is only
        possible on the first register*/
9    check = PERMUTE(check,r_shift); /* Set up to and with field polynomial*/
10   u128 o = SL(x,1); /* Doubling every element*/
11   ca = PERMUTE(ca,sm); /*Most all MSB's to LSB in the next position*/
12   ca = ca & r_shift;  /* Remove a potential 1 from the MSB vector*/
13   o = o | ca; /* Adding shifted MSB's */
14   o = o^(red & check); /* Addition of field polynomial if required */
15   return o;
16 }
```

Listing 5.4: ARM intrinsics constant time field doubling code.

This code performs the same basic operation, but with the difference that every vector is interpreted as sixteen unsigned 8-bit integers, due to ARM operations and vectors being less flexible in interpretation than Intel ones. The principle is the same. A comparison (line 7) to get whether the most significant bit is a one, ensuring that the bit that must carry over to the next register does so, and addition of the field polynomial if necessary.

5.1 Processors

5.1.1 Regular processors

Every scheme I have implemented have been tested on the following three processors. I have not only tested the performance of the algorithms, but also the variance in performance, in order to say something about the reliability of the results. This is particularly important for the AMD Piledriver and Intel Kaby Lake processors, which are both in my personal computers that are never fully idle. Table 5.1 lists the setups used in the tests.

Skylake Intel Core i7-6700 is a processor where the Skylake architecture was used, and is the processor in a computer located at the Computer Security department of DTU. It has a processor speed of 3.4 GHz. On this machine, turbo boost is turned off, and the processor is as close to idle as possible when testing. The tests were 1000 batches of 100000 runs of the encryption algorithms. The compiler used is GCC 5.2.1. The Skylake architecture supports SSE extensions up to SSE4.2, AVX instructions up to AVX2 and AES instructions [24] .

Piledriver The AMD A6-4455 processor, based on the AMD Piledriver architecture, is the processor in my old laptop, which I primarily use as a streaming box and for backup storage. It has a processor speed of 2.1 GHz. The tests were run with turbo boost turned off, and with as many unnecessary processes as possible turned off. It should be noted that every result on this processor needs to be taken with a grain of salt in terms of absolute performance numbers, as it is a four and a half year old laptop that has not been maintained well. This point is underlined by the variance of the test results, which can be seen in appendix A.2, where the AMD tests have substantially more variance than the Skylake or Kaby Lake tests. In terms of relative performance between schemes, it is expected to show somewhat reliable numbers. The tests on the Piledriver processor ran 200 bathes of 100000 runs of the encryption algorithms. The compiler used on this computer is GCC 4.7.1. The Piledriver architecture supports SSE extensions up to SSE4a, AVX instructions up to AVX 1.1 and AES instructions [42].

Kaby Lake Intel Core i5-7300HQ is a processor where the Kaby Lake architecture was used, and is the processor in my personal laptop. It has a processor speed of 2.5 GHz. The processor was not completely idle, but rather running with turbo boost turned off, and as many unnecessary processes as possible turned off, but being a Windows 10 computer that I use on a daily basis, it may have certain processes that slows it down. While performance numbers for the algorithm relative to each other are expected to hold, the absolute performance figures may be a bit slower than achievable figures. The tests were 1000 batches of 100000 runs of the encryption algorithms. Two sets of tests were made, one using GCC 4.9.2 and another using GCC 5.2.0. The Kaby Lake architecture supports SSE extensions up to SSE4.2, AVX instructions up to AVX2 and AES instructions [25] .

5.1.1.1 Comparison table

Architecture	Core	Clock	L1 data cache size	Compiler
Skylake	Intel Core i7-6700	3.4 GHz	32 KB [55]	GCC 5.2.1
Kaby Lake	Intel Core i5-7300HQ	2.5 GHz	32 KB [56]	GCC 5.2.0
Piledriver	AMD A6-4455	2.1 GHz	32 KB [15]	GCC 4.7.1
ARM	ARM Cortex A-57	2.1 GHz	32 KB [57]	GCC 5 cross compiler

Table 5.1: The four setups used for performance tests.

5.1.2 Mobile processors

I have tested my $COLM_0$ implementation and my Deoxys-AES implementation on the following processor.

ARM The ARM AArch 64 processor is found in a Samsung Galaxy S6 phone which is wired to the Skylake computer at the Computer Security department at DTU. The core is an ARM Cortex A-57 with a processor speed of 2.1 GHz. Code is cross-compiled using the GCC 5 cross compiler for ARM64 and transferred over, which may lead to certain suboptimal translations. The tests on the ARM processor ran 200 batches of 100000 runs of the encryption algorithms.

CHAPTER 6

COLM

COLM is a block cipher based AEAD scheme. There are two schemes described in the COLM paper, $COLM_0$ and $COLM_{127}$, both of which are designed with nonce-misuse resistance in mind [2].

6.1 Description

COLM uses a key size and a state size of 128 bits, and is designed in a generalized manner with a parameter τ that describes how often intermediate tags are computed, as well as a length of these intermediate tags, l_τ. In general, a COLM variant can be can be called $COLM_\tau$, where $COLM_0$ and $COLM_{127}$ are the two recommended variants, both with a tag length of 128 bits. $COLM_0$ is a special case where no intermediate tags are generated, while $COLM_\tau$ generates an intermediate tag for every τ blocks of plaintext encrypted, meaning that an additional tag is generated after 127 blocks in the case of $COLM_{127}$. Only the two recommended variants are considered in this thesis.

A block is defined as a bistring of at most 128 bits, where a shorter string is called an incomplete block. The bits in a block A can be written as $a_{127}a_{126}\dots a_1a_0$. Any string can be represented as a sequence of blocks, where the last block may be incomplete. All computations are done in the finite field $GF(2^n)$, where each block can be represented as the polynomial $a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$. Addition in the field is simply the xor operation between bistrings, while multiplication is defined as polynomial multiplication modulo the irreducible [2] polynomial $x^{128} + x^7 + x^2 + x + 1$. Any addition and multiplication described in this chapter will follow these rules, where $+$ and $\oplus$ are used interchangeably. Furthermore, numbers are used as short notation for polynomials in formulas, where $\mathsf{2}$ means x, $\mathsf{3}$ means $x + 1$ and $\mathsf{7}$ means $x^2 + x + 1$, and raised to any power means field multiplication of said polynomial.

During encryption, COLM uses a linear mixing function ρ, which takes two inputs, x and w and transforms them in the following way:

$$\begin{aligned} y &= x \oplus \mathsf{3}w \\ w' &= x \oplus \mathsf{2}w \end{aligned} \tag{6.1}$$

During decryption, the process is inverted by applying the following transformation, which takes the inputs y and w'

$$\begin{aligned} x &= y \oplus 3w' \\ w &= y \oplus w' \end{aligned} \tag{6.2}$$

COLM in general takes the following inputs, where no numbers are shorthands for polynomials

- 128-bit encryption key $K \in \{0,1\}^{128}$
- Public message number $nonce \in \{0,1\}^{64}$
- Parameter set which represents the tag interval τ and the tag length l_τ. This can be fixed within an implementation.
- Associated data of arbitrary length, $A \in \{0,1\}^*$, with the technical limitation that it must be less than 2^{64} bits long [14].
- Plaintext of arbitrary length, $M \in \{0,1\}^*$, with the technical limitation that it must be less than 2^{64} bits long.

COLM takes no secret message number, or takes a secret message number of length 0. $COLM_\tau$ returns a tagged ciphertext $C \in \{0,1\}^{|M|+128(h+1)}$, where $h = 0$ if $\tau = 0$, and $h = \left\lfloor \frac{l-1}{\tau} \right\rfloor$ is the number of intermediate tags otherwise, and $l = \left\lceil \frac{|M|}{128} \right\rceil$ is the number of blocks to encrypt.

The first step in the encryption process is to generate the value $L = E_K(0^{128})$, where 0^{128} means a string of 128 zeros. This is used to compute $L_1 = 3 \cdot L$ and $L_2 = 3^2 \cdot L$.

The second step is the processing of AD, which is done by splitting the AD into blocks, and if the final block is incomplete, it gets completed by 10^* padding, which means adding a one followed by the number of zeros it will take to make it a complete block. The algorithm for AD processing looks as follows, where a is the number of AD blocks and A is the AD.

```
W'[0] = ENC( XOR( (nonce || param) , delta_A), K)
for i=1..a:
  AA[i] = XOR(A[i], delta_A[i])
  Z[i] = ENC(AA[i],K)
  W'[i] = XOR(Z[i], W'[i-1])
end for
IV = W'[a] // To be used later
```

Listing 6.1: COLM AD processing.

Where $\Delta_A[i]$ is defined by

$$\Delta_A[i] = \begin{cases} 7 \cdot 2^{i-1} \cdot L_1 & i = a \text{ and } |A^*[a]| < 128 \\ 2^i \cdot L_1 & \text{otherwise} \end{cases}$$

The last value of $W'[a]$ is used as the initial value for the encryption loop. This means that the encryption does not only rely on the nonce, which was used as one of the initial steps of the associated data processing, but also on the associated data itself. After this point, the algorithms $COLM_0$ and $COLM_{127}$ differ.

6.1.1 $COLM_0$ encryption

The first step of the encryption is done by splitting the message into blocks. The final block is denoted $M^*[l]$, and the final block is 10^* padded if it is not complete. The algorithm is as follows, where l is the number of blocks in the plaintext:

```
W'[0] = IV
M[1] = M[1] \oplus \cdots \oplus M[l-1] \oplus (M*[l] || 10*)
M[l + 1] = M[l]
for i=1..(l+1):
  MM[i] = XOR(M[i], delta_M[i])
  X[i] = ENC(MM[i],K)
  (Y[i],W'[i]) = rho(X[i], W'[i-1])
  CC[i] = ENC(Y[i],K)
  C[i] = XOR(CC[i],delta_C[i])
end for
```

Listing 6.2: $COLM_0$ encryption.

Where the masking values, $\Delta_M[i]$ and $\Delta_C[i]$, are defined by

$$\Delta_M[i] = \begin{cases} 7 \cdot 2^{i-1} \cdot L & i \in l, l+1 \text{ and } |A^*[a]| = 128 \\ 7^2 \cdot 2^{i-1} \cdot L & i \in l, l+1 \text{ and } |A^*[a]| < 128 \\ 2^i \cdot L & \text{otherwise} \end{cases}$$

$$\Delta_M[i] = \begin{cases} 7 \cdot 2^{i-1} \cdot L_2 & i \in l, l+1 \text{ and } |A^*[a]| = 128 \\ 7^2 \cdot 2^{i-1} \cdot L_2 & i \in l, l+1 \text{ and } |A^*[a]| < 128 \\ 2^i \cdot L_2 & \text{otherwise} \end{cases}$$

Aside from the encryption step, the ρ step is a minor bottleneck, because it must be performed sequentially. The other steps shown directly in the algorithm can be performed in parallel, by utilizing the pipeline.

The algorithm is described from an implementation standpoint, as the final steps require different masking values depending on the situation, and are dependent on the computation of $M[l]$, which is done most effectively during encryption, rather than in its own loop. The algorithm returns the tagged ciphertext C.

$$C = (C[1], ..., C[l], \lfloor C[l+1] \rfloor_{|M^*[l]|}) \tag{6.3}$$

COLM is an online scheme, meaning that the plaintext is only processed once, and the i'th ciphertext block is available immediately after encryption.

6.1.2 $COLM_{127}$ encryption

The first step of the encryption is done by splitting the message into blocks. The final block is denoted $M^*[l]$, and the final block is 10^* padded if it is not complete. The algorithm is as follows, where h is the number of tags:

```
W'[0] = IV
j = 1
for i=1..(l+h+1):
  MM[i] = XOR(M[i], delta_M[i])
  X[i] = ENC(MM[i],K)
  (Y[i],W'[i]) = rho(X[i], W'[i-1])
  if(i$ mod $127 = 0):
    TT[j] = ENC(W,K)
    T[j] = XOR(delta_C[i],TT[j])
    j = j + 1
    i = i + 1
  end if
  CC[i] = ENC(Y[i],K)
  C[i] = XOR(CC[i],delta_C[i])
end for
```

Listing 6.3: $COLM_{127}$ encryption.

The masking values change to
Where the masking values, $\Delta_M[i]$ and $\Delta_C[i]$, are defined by

$$\Delta_M[i] = \begin{cases} 7 \cdot 2^{i-1} \cdot L & i \in l+h, l+h+1 \text{ and } |A^*[a]| = 128 \\ 7^2 \cdot 2^{i-1} \cdot L & i \in l+h, l+h+1 \text{ and } |A^*[a]| < 128 \\ 2^i \cdot L & \text{otherwise} \end{cases}$$

$$\Delta_M[i] = \begin{cases} 7 \cdot 2^{i-1} \cdot L_2 & i \in l, l+1 \text{ and } |A^*[a]| = 128 \\ 7^2 \cdot 2^{i-1} \cdot L_2 & i \in l, l+1 \text{ and } |A^*[a]| < 128 \\ 2^i \cdot L_2 & \text{otherwise} \end{cases}$$

The algorithm has been written from an implementation focused standpoint, due to the finalization step being different, and to link the intermediate tags with the general encryption scheme. Again, the ρ function is a minor bottleneck. The generation of intermediate tags is another bottleneck, as they require single-block encryptions. The remaining steps can be pipelined.

6.1.3 $COLM_0$ decryption and verification

$COLM_0$ decryption follows the steps of $COLM_0$ encryption in reverse. First the initial value is generated from the AD, then used to decrypt, using the following algorithm:

```
W'[0] = IV
for i=1..(l+1):
```

```
  CC[i] = XOR(C[i],delta_C[i])
  Y[i] = DEC(CC[i],K)
  (X[i],W[i]) = rho_inv(Y[i], W[i-1])
  MM[i] = DEC(X[i],K)
  M[i] = XOR(MM[i], delta_M[i])
end for
M[l] = M[1] XOR ... XOR M[l-1] XOR (M*[l] || 10*)
M[l + 1] = M[l]
```

Listing 6.4: $COLM_0$ decryption.

The decryption algorithm has the same bottleneck as the encryption algorithm, with the ρ function having to run sequentially. After the ciphertext is decrypted, the verification follows the following algorithm

```
MM[l+1] = XOR(M[l+],delta_M[l+1)
X[l+1] = ENC(MM[l+1],K)
(Y[l+1],W[l+1]) = rho(X[l+1], W[i])
CC[l+1] = ENC(Y[l+1],K)
C[l+1] = XOR(CC[l+1],delta_C[l+1])
```

Listing 6.5: $COLM_0$ verification.

The verification succeeds if

- In the case that the final block is complete, we have $C[l+1] = C'[l+1]$.
- In the case that the final block isn't complete, we have $C[l+1] = \lfloor C'[l+1] \rfloor_{|C[l+1]|}$, and the last $128 - |C[l+1]|$ bits of $M^*[l]$ are 10^*.

If the verification is successful, the decryption will return the plaintext, otherwise it must return an error to verify.

It can be seen in the algorithm that the final ciphertext block is the authentication tag. This leads to $COLM_0$ (and $COLM_{127}$) having an EtM-like approach. The final ciphertext block is the result of xor on all ciphertext blocks, but isn't directly hashed to produce the tag.

6.1.4 $COLM_{127}$ decryption and verification

$COLM_{127}$ follows the same decryption process as $COLM_0$, with verification of intermediate tags as an additional part of the process.

Verification of the final tag follows the exact same process as $COLM_0$ verification.

To verify an intermediate tag, the following must be computed

$$\begin{aligned} TT[j] &= T[j] \oplus \Delta_C[128 \cdot j] \\ W'[j] &= E_K^{-1}(TT[j]) \end{aligned} \tag{6.4}$$

Where the tag is verified if:

$$W[127 \cdot j] = W'[j] \tag{6.5}$$

If an intermediate tag is not verified, the ciphertext is rejected, and the algorithm returns failure to verify.

6.2 Theoretical performance

To compute the number of operations, I have decided to do so under two different assumptions: A theoretical one, and a practically feasible one. The theoretical assumption is that everything not explicitly stated in the encryption loop can be precomputed, and there is enough memory to do so, regardless of the length of the message. This is called the unlimited memory assumption (UMA). If UMA is in effect, it is possible to precompute every value of Δ_A, Δ_M and Δ_C for any key, since these values rely solely on the key. A tradeoff where the initial values of Δ_A, Δ_M and Δ_C are precomputed is left as an unexplored option in practice.

The practically feasible assumption is that the scheme should use constant memory internally, and the only elements that are allowed to vary is the length of the message, length of the AD, and the length of the ciphertext. This is called the constant memory assumption (CMA). The number of cycles each intrinsic operation takes can be found in appendix B.2. The theoretical performance of Kaby Lake is assumed to be equal to that of Skylake, being an optimization of Skylake [16] .

6.2.0.1 First steps and AD processing

The first step, before processing the AD, is to generate the key schedule. The key schedule is generated under both UMA and CMA, and has a performance of 200 cycles on Skylake, and 180 cycles on other Intel processors. The calculation can be seen in appendix C.1. The reason for this choice is that in the CAESAR API, the key is listed as input.

The first step of AD processing is to generate the value $W'[0]$, which takes a single block encryption (number of cycles depends on processor, varies from 41 to 81 cycles, calculation in C.4), a concatenation (assumed free), an xor, and a load of $\Delta_A[0]$. In the constant memory case, there is no load of $\Delta_A[0]$, but the value L must also be computed to generate the value $\Delta_A[0]$. Computation of L is a single block encryption, preceded by setting a block to zero (one cycle), and to get $\Delta_A[0]$, an additional xor and field doubling, which can be seen in appendix C.2, are required. It should be noted that while the theoretical performance of field doubling is nine cycles, the practical performance is actually two cycles, as determined by experiment described in section 8.2.1.

In the processing loop itself, which runs over the length of the AD, performance depends on whether CMA or UMA is in effect. What happens are the following operations.

- Two xors
- One load
- Single block encryption
- Field doubling under CMA, load under UMA.

Field doubling can be parallelized to a certain degree, but since every use of field doubling in a loop has an input that is directly dependent on the previous input, and the actual performance of field doubling is much better, the point is rendered moot.

This is where parallel processing allows for increasing the number of bytes processed per cycle, and where performance numbers for different processors become an issue. For this particular calculation, it is assumed that Ivy Bridge, Broadwell and Haswell processors encrypt eight blocks in parallel, while theoretical numbers are computed for Skylake encrypting both four and eight blocks in parallel.

Processor \Step	load	doubling	xor	encryption	xor*	total
Skylake 4	1	36 (1)	2	43	3	85 (50)
Skylake 8	2	72 (2)	3	83	4	164 (94)
Broadwell / Haswell	3	72 (3)	3	87	4	169 (100)
Ivy Bridge	4	72 (4)	3	88	4	171 (103)

Table 6.1: Performance per iteration in cycles for AD processing under CMA. UMA in parenthesis.

The justification for the encryption numbers can be seen in appendix C.5. The load operation and the first xor operation are fairly straightforward, as can be seen in table B.2. The operation marked as xor* takes the recently computed blocks and xor's them together, which means that the operation won't be completely parallel, but to achieve as much parallelism as possible, the operations will be performed in the most parallel manner possible.

The performance numbers are close to identical, for Broadwell, Haswell and Ivy Bridge, while Skylake only take around half the number of cycles. It should be noted that Skylake also only processes half the information, meaning that processing AD will be faster on a Broadwell or a Haswell processor, if only by 0.002 cycles per byte. The advantage of only processing four blocks per iteration is that it won't be necessary to revert to single encryptions, which are more expensive, before there are less than four block remaining.

Single block processing for AD takes two xor's, an encryption, and a load under UMA or a field doubling under CMA.

Regardless of the processor, another 20 cycles are required for $\Delta_A[a]$ if the final block is incomplete, because of the multiplication by 7, unless under the unlimited memory assumption.

The total performance numbers come out to, where n_A is the number of AD blocks, and I_A is a binary variable, which is 1 if the last plaintext block is incomplete, and 0 otherwise.

Processor \Step	Precomputation	Each iteration	Final blocks	Incomplete block
Skylake 4	294 (242)	85 (50)	52 (44)	63 (44)
Skylake 8	294 (242)	164 (94)	52 (44)	63 (44)
Broadwell / Haswell	334 (252)	169 (100)	82 (74)	93 (74)
Ivy Bridge	354 (262)	171 (103)	92 (84)	103 (84)

Table 6.2: Performance of the elements on COLM AD processing in cycles.

Processor \Step	total
Skylake 4 (CMA)	$294 + 85\lfloor \frac{n_A}{4} \rfloor + 52(n_A \mod 4) + 63I_A$
Skylake 4 (UMA)	$242 + 50\lfloor \frac{n_A}{4} \rfloor + 44(n_A \mod 4) + 44I_A$
Skylake 8 (CMA)	$294 + 164\lfloor \frac{n_A}{8} \rfloor + 52(n_A \mod 8) + 63I_A$
Skylake 8 (UMA)	$242 + 94\lfloor \frac{n_A}{8} \rfloor + 44(n_A \mod 8) + 44I_A$
Broadwell / Haswell (CMA)	$334 + 169\lfloor \frac{n_A}{8} \rfloor + 82(n_A \mod 8) + 93I_A$
Broadwell / Haswell (UMA)	$252 + 100\lfloor \frac{n_A}{8} \rfloor + 74(n_A \mod 8) + 74I_A$
Ivy Bridge (CMA)	$354 + 171\lfloor \frac{n_A}{8} \rfloor + 62(n_A \mod 8) + 103I_A$
Ivy Bridge (UMA)	$262 + 103\lfloor \frac{n_A}{8} \rfloor + 84(n_A \mod 8) + 84I_A$

Table 6.3: Performance formulas for COLM AD processing under CMA, where n_A is the number of AD blocks, and I_A is an indicator variable, which is 1 if there is an incomplete block and 0 otherwise. Result will be in cycles.

This is where the Skylake processor shines. While the standard iterations take approximately the same number of cycles per byte, precomputation and the final blocks cost significantly less in terms of cycles per byte, as does the incomplete block.

6.2.0.2 $COLM_0$ plaintext processing

The plaintext processing covers both the encryption and tag generation steps. It is assumed that at this point, the initial value has already been generated, but it is still necessary to compute $\Delta_C[0]$, which takes a field doubling and an xor. Operations are counted for both CMA and UMA. The ρ function takes 11 cycles, and the calculation can be seen in appendix C.3. That calculation relies on the previously calculated value for field doubling, which is much more efficient in practice. The performance numbers per iteration are.

Processor \Step	load	mul2	xor*	enc	ρ	enc	mul2	xor	store	total
Skylake 4	1	36 (1)	3	43	44	43	36 (1)	2	1	209 (139)
Skylake 8	2	72 (2)	4	83	88	83	72 (2)	3	2	409 (269)
Broadwell / Haswell	3	72 (3)	4	87	88	87	72 (3)	3	3	419 (281)
Ivy Bridge	4	72 (4)	4	88	88	88	72 (4)	3	4	423 (285)

Table 6.4: Performance per iteration of encryption and tag generation under CMA. UMA in parenthesis. All numbers are in cycles.

Again, the performance numbers are close to identical in terms of cycles per byte for the processors, and the difference is in single block processing, where less blocks must be processed by themselves, and the process is substantially cheaper on Skylake.

Processing a single block requires one load, two xors, two encryptions and one ρ step, as well as two field doublings or loads, depending on the assumption. This leads to a cost of 32 cycles plus two single block encryptions under the constant memory assumption, and 16 blocks plus two single block encryptions under the unlimited memory assumption.

Furthermore, it is assumed that the last two blocks are treated as a special case in the implementation, as both Δ_M and Δ_C contain multiplications by 7. The last block is just a regular single block without one load (31/15 cycles), but the penultimate block costs 53 cycles plus two encryptions, 22 cycles more than a regular single block, which is twice the difference between doubling and multiplication by 7. If the final plaintext block is incomplete, an additional 40 cycles are added. The total performance figures are, where n_B is the number of plaintext blocks and I_B is a binary variable, which is 1 if the last block is incomplete and 0 otherwise:

Processor \Step	Precomputation	Each iteration	Single blocks	Final blocks	Incomplete block
Skylake 4	10 (0)	209 (139)	116 (100)	248 (190)	40 (0)
Skylake 8	10 (0)	409 (269)	116 (100)	248 (190)	40 (0)
Broadwell / Haswell	10 (0)	419 (281)	174 (158)	368 (312)	40 (0)
Ivy Bridge	10 (0)	423 (285)	194 (178)	408 (352)	40 (0)

Table 6.5: Total the individual elements of $COLM_0$ encryption under CMA. UMA in parenthesis. All numbers are in cycles.

Skylake 4 (CMA)	$258 + 209\lfloor \frac{n_B}{4} \rfloor + 116(n_B \bmod 4) + 40I_B$
Skylake 4 (UMA)	$190 + 139\lfloor \frac{n_B}{4} \rfloor + 116(n_B \bmod 4)$
Skylake 8 (CMA)	$258 + 409\lfloor \frac{n_B}{8} \rfloor + 116(n_B \bmod 8) + 40I_B$
Skylake 8 (UMA)	$190 + 269\lfloor \frac{n_B}{8} \rfloor + 116(n_B \bmod 8)$
Broadwell / Haswell (CMA)	$378 + 419\lfloor \frac{n_B}{8} \rfloor + 174(n_B \bmod 8) + 40I_B$
Broadwell / Haswell (UMA)	$312 + 281\lfloor \frac{n_B}{8} \rfloor + 174(n_B \bmod 8)$
Ivy Bridge (CMA)	$418 + 423\lfloor \frac{n_B}{8} \rfloor + 194(n_B \bmod 8) + 40I_B$
Ivy Bridge (UMA)	$352 + 285\lfloor \frac{n_B}{8} \rfloor + 178(n_B \bmod 8)$

Table 6.6: Performance formulas for encryption and tag generation, where n_B is the number of plaintext blocks and I_B is an indicator variable which is 1 if there is an incomplete block and 0 otherwise.

The difference between the processors can be seen under the last steps. Preprocessing is identical, because the major steps in preprocessing had already taken place under AD generation. The difference in terms of cycles per byte is under the final regular blocks and final two blocks step, where Skylake uses less processing cycles for encryption, and does not need to process as many regular blocks at the end. The total performance formulas are

Skylake 4 (CMA)	$552 + 209\lfloor \frac{n_B}{4} \rfloor + 116(n_B \bmod 4) + 40I_B$ $+85\lfloor \frac{n_A}{4} \rfloor + 52(n_A \bmod 4) + 63I_A$
Skylake 4 (UMA)	$432 + 139\lfloor \frac{n_B}{4} \rfloor + 116(n_B \bmod 4)$ $+50\lfloor \frac{n_A}{4} \rfloor + 44(n_A \bmod 4) + 44I_A$
Skylake 8 (CMA)	$552 + 409\lfloor \frac{n_B}{8} \rfloor + 116(n_B \bmod 8) + 40I_B$ $+164\lfloor \frac{n_A}{8} \rfloor + 52(n_A \bmod 8) + 63I_A$
Skylake 8 (UMA)	$432 + 269\lfloor \frac{n_B}{8} \rfloor + 116(n_B \bmod 8)$ $+94\lfloor \frac{n_A}{8} \rfloor + 44(n_A \bmod 8) + 44I_A$
Broadwell / Haswell (CMA)	$712 + 419\lfloor \frac{n_B}{8} \rfloor + 174(n_B \bmod 8) + 40I_B$ $+169\lfloor \frac{n_A}{8} \rfloor + 82(n_A \bmod 8) + 93I_A$
Broadwell / Haswell (UMA)	$564 + 281\lfloor \frac{n_B}{8} \rfloor + 174(n_B \bmod 8)$ $+100\lfloor \frac{n_A}{8} \rfloor + 74(n_A \bmod 8) + 74I_A$
Ivy Bridge (CMA)	$772 + 423\lfloor \frac{n_B}{8} \rfloor + 194(n_B \bmod 8) + 40I_B$ $+171\lfloor \frac{n_A}{8} \rfloor + 62(n_A \bmod 8) + 103I_A$
Ivy Bridge (UMA)	$614 + 285\lfloor \frac{n_B}{8} \rfloor + 178(n_B \bmod 8)$ $+103\lfloor \frac{n_A}{8} \rfloor + 84(n_A \bmod 8) + 84I_A$

Table 6.7: Performance formulas for $COLM_0$ encryption and AD processing, where n_B is the number of plaintext blocks and I_B is an indicator variable which is 1 if there is an incomplete block and 0 otherwise.

6.2.0.3 $COLM_{127}$ plaintext processing and tag generation

Most of the operations in $COLM_{127}$ are similar to the operations in $COLM_0$, except generation of intermediate tags. The intermediate tag generation is performed alongside the ρ operations, allowing for minimal disruption of the parallelism. Generation of an intermediate tag takes a single block encryption, an xor and a field doubling (CMA) or a load (UMA). No new operations are described, and it is assumed that the only necessary precomputation in the contant memory case is the initial value of Δ_C, which takes 10 cycles. The starting point, as in the previous section, is after the AD processing. The interesting case for the parallel iterations is how often an intermediate tag is generated, and how expensive the intermediate tags are to generate, as the other performance numbers are otherwise identical to the performance numbers seen in table 6.4.

Processor \Step	enc	mul2	xor	store	total
Skylake	41	9 (1)	2	1	53 (45)
Broadwell / Haswell	71	9 (1)	3	3	86 (78)
Ivy Bridge	81	9 (1)	3	4	97 (89)

Table 6.8: Performance numbers in cycles for intermediate tag generation in $COLM_{127}$.

This is another case where the latency of the encryption operation determines the number of cycles required to do the computation. The two differences are the minor difference in the store, and the major difference in the encryption performances.

Another difference is the final two blocks, the first of which can be succeeded by an intermediate tag. This does not matter much to the total performance figures, as it just moves the expensive multiplication by 7 to happen during computation of the intermediate tag, as opposed to afterwards. This means that the performance figures for the last two blocks can be assumed identical. The total performance figures are, where n_T is the number of intermediate tags:

Processor \Step	Precomputation	Iteration	Tags	Single blocks	Final blocks	Incomplete block
Skylake 4	10 (0)	209 (139)	53 (45)	116 (100)	248 (190)	40 (0)
Skylake 8	10 (0)	409 (269)	53 (45)	116 (100)	248 (190)	40 (0)
Broadwell / Haswell	10 (0)	419 (281)	86 (78)	174 (158)	368 (312)	40 (0)
Ivy Bridge	10 (0)	423 (285)	97 (89)	194 (178)	408 (352)	40 (0)

Table 6.9: Performance numbers for the individual parts of $COLM_{127}$ encryption.

Processor \Step	Total
Skylake 4 (CMA)	$258 + 209\lfloor \frac{n_B}{4} \rfloor + 116(n_B \bmod 4) + 40I_B + 53n_T$
Skylake 4 (UMA)	$190 + 139\lfloor \frac{n_B}{4} \rfloor + 100(n_B \bmod 4) + 45n_T$
Skylake 8 (CMA)	$258 + 409\lfloor \frac{n_B}{8} \rfloor + 116(n_B \bmod 8) + 40I_B + 53n_T$
Skylake 8 (UMA)	$190 + 269\lfloor \frac{n_B}{8} \rfloor + 100(n_B \bmod 8) + 45n_T$
Broadwell / Haswell (CMA)	$378 + 419\lfloor \frac{n_B}{8} \rfloor + 174(n_B \bmod 8) + 40I_B + 86n_T$
Broadwell / Haswell (UMA)	$312 + 281\lfloor \frac{n_B}{8} \rfloor + 158(n_B \bmod 8) + 78n_T$
Ivy Bridge (CMA)	$418 + 423\lfloor \frac{n_B}{8} \rfloor + 194(n_B \bmod 8) + 40I_B + 97n_T$
Ivy Bridge (UMA)	$352 + 285\lfloor \frac{n_B}{8} \rfloor + 178(n_B \bmod 8) + 89n_T$

Table 6.10: Performance formulas for $COLM_{127}$ encryption and AD processing, where n_B is the number of plaintext blocks, I_B is an indicator variable which is 1 if there is an incomplete block and 0 otherwise, and n_T is the number of intermediate tags.

The overall conclusion from the theoretical numbers is that the processor doesn't make much of a difference in the ideal case, but when the use case deviates from the ideal, it shows that Skylake with a lower latency for the intrinsic encryption function achieves far better performance figures, due to not requiring nearly as many cycles to compute the ciphertext under a single block encryption.

Processor \Step	Total
Skylake 4 (CMA)	$258 + 209\lfloor \frac{n_B}{4} \rfloor + 116(n_B \bmod 4) + 40I_B + 53n_T$ $+85\lfloor \frac{n_A}{4} \rfloor + 52(n_A \bmod 4) + 63I_A$
Skylake 4 (UMA)	$190 + 139\lfloor \frac{n_B}{4} \rfloor + 100(n_B \bmod 4) + 45n_T$ $+50\lfloor \frac{n_A}{4} \rfloor + 44(n_A \bmod 4) + 44I_A$
Skylake 8 (CMA)	$258 + 409\lfloor \frac{n_B}{8} \rfloor + 116(n_B \bmod 8) + 40I_B + 53n_T$ $+164\lfloor \frac{n_A}{8} \rfloor + 52(n_A \bmod 8) + 63I_A$
Skylake 8 (UMA)	$190 + 269\lfloor \frac{n_B}{8} \rfloor + 100(n_B \bmod 8) + 45n_T$ $+94\lfloor \frac{n_A}{8} \rfloor + 44(n_A \bmod 8) + 44I_A$
Broadwell / Haswell (CMA)	$378 + 419\lfloor \frac{n_B}{8} \rfloor + 174(n_B \bmod 8) + 40I_B + 86n_T$ $+169\lfloor \frac{n_A}{8} \rfloor + 82(n_A \bmod 8) + 93I_A$
Broadwell / Haswell (UMA)	$312 + 281\lfloor \frac{n_B}{8} \rfloor + 158(n_B \bmod 8) + 78n_T$ $+100\lfloor \frac{n_A}{8} \rfloor + 74(n_A \bmod 8) + 74I_A$
Ivy Bridge (CMA)	$418 + 423\lfloor \frac{n_B}{8} \rfloor + 194(n_B \bmod 8) + 40I_B + 97n_T$ $+171\lfloor \frac{n_A}{8} \rfloor + 62(n_A \bmod 8) + 103I_A$
Ivy Bridge (UMA)	$352 + 285\lfloor \frac{n_B}{8} \rfloor + 178(n_B \bmod 8) + 89n_T$ $+103\lfloor \frac{n_A}{8} \rfloor + 84(n_A \bmod 8) + 84I_A$

Table 6.11: Performance formulas for $COLM_{127}$ encryption, where n_B is the number of plaintext blocks, I_B is an indicator variable which is 1 if there is an incomplete block and 0 otherwise, and n_T is the number of intermediate tags.

6.3 Implementation

The first step of implementation is the key scheduling algorithm, which follows a standard implementation based on Intel's own implementation example [26]. Using Intel Intrinsics, the key expansion is fairly straightforward

```
unsigned char keys[11]; // should be keys[20] if decrypting

__m128i key_exp_assist(__m128i t1, __m128i t2)
{
  __m128i t3 = _mm_slli_si128(t1,0x04);
  t1 = _mm_xor_si128(t1,t3);
  t3 = _mm_slli_si128(t1,0x04);
  t1 = _mm_xor_si128(t1,t3);
  t3 = _mm_slli_si128(t1,0x04);
  t1 = _mm_xor_si128(t1,t3);
  t2 = _mm_shuffle_epi32(t2,0xFF);
  return _mm_xor_si128(t1,t2);
}
void generate_enc_key(__m128i key)
{
  const unsigned char rcon[10] = {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0
     x1b,0x36};
  __m128i kt;  int i;
  keys[0] = key;
  for(i=0;i<10;++i){ // Loop fully unrolled in actual implementation
    kt = _mm_aeskeygenassist_si128(keys[i], rcon[i]);
    keys[i+1] = key_exp_assist(keys[i],kt);
  }
  /*for(i=11;i<20;++i){ // Decryption keys only
    keys[i] = _mm_aesimc_si128(keys[20-i]);
  } */
}
```

Listing 6.6: Key expansion code.

It shows why key scheduling is such a heavy operation. Every subkey is dependent on the last, and the `key_exp_assist` subfunction is entirely serial as well. Under decryption, another nine keys are needed, generation of which are shown in the commented for-loop. The operations here run independently of each other, and can utilize the pipeline, meaning that generating these additional keys will take 24 cycles on Skylake and 30 cycles on Broadwell, Haswell and Ivy Bridge. Latency and throughput can be seen in appendix C.1.

The constant time field doubling, which can be seen in listing 5.3, can be parallelized to process 3 blocks at the time since all processors can process three logical operations simultaneously. The processing will now take 19 cycles, as opposed to 9 for a single block. The reason this isn't further explored is that there is no instance where field doubling can be performed in parallel, as every time a field doubling is performed, it is dependent on the directly preceding value.

Another point to note here is treatment of the final two blocks, which are considered a special case due to the field multiplication by 7 for the second last block, and can conceptually be seen in listing 6.7, with the encryption loop and block processing stripped away. An alternate way of implementing this could be through if-statements and encrypting the plaintext blocks in parallel instead. I chose this way to improve readability and to clarify that it is a special case, as well as accounting for flexibility for cases where the plaintext isn't of a length that allows for perfect parallel processing without having additional single block encryption steps, which would cover most cases. The expensive multiplication by 7 would take place regardless of what is done in the implementation. This implementation is rather subjective, and every test I have performed on COLM schemes have been with a plaintext of a size that takes advantage of the implementation.

```
int mf = numblocks_mes-1; // numblocks_mes = number of complete blocks
for(i=0;i<mf;i+=PARA){ // PARA is the number of blocks being processed in
    parallel, and is defined by a macro
  /*Encryption loop*/
}
delta = a_Delta[PARA-1]; // Gather last delta value
_2delta = mul2(delta);
delta = _mm_xor_si128(_mm_xor_si128(delta ,_2delta),mul2(_2delta));
deltaC = a_DeltaC[PARA-1];
__m128i _2deltaC = mul2(deltaC);
deltaC = _mm_xor_si128(_mm_xor_si128(deltaC ,_2deltaC),mul2(_2deltaC));
if(fin_mes%16){ // If last block is incomplete
  _2delta = mul2(delta);
  delta = _mm_xor_si128(_mm_xor_si128(delta ,_2delta),mul2(_2delta));
  _2deltaC = mul2(deltaC);
  deltaC = _mm_xor_si128(_mm_xor_si128(deltaC ,_2deltaC),mul2(_2deltaC));
}
/*Regular block processing*/
```

Listing 6.7: Getting final values for Δ_M and Δ_C.

CHAPTER 7

Deoxys

Deoxys is a tweakable AEAD scheme. There are two versions of Deoxys. One version which is geared towards defence in depth, and is resistant towards nonce misuse, and another which is geared towards high performance applications, and is not resistant towards nonce misuse. In this thesis, the focus is on the misuse resistant version with a 128 bit key, called Deoxys-II, which is based on stream ciphers.

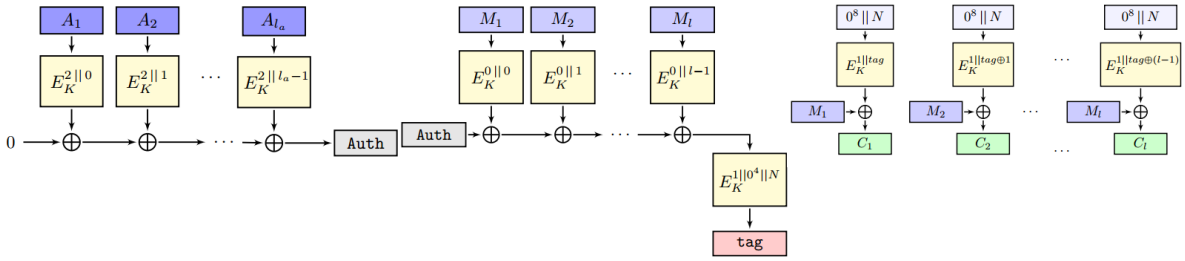


Figure 7.1: An overview of the encryption process for nonce misuse resistant Deoxys [35].

7.1 Description

Deoxys is designed with a state size of 128-bits in mind, but with key sizes of both 128 bits and 256 bits. The tweak size is also fixed at 128 bits. The nonce size for Deoxys-II is set to 120 bits, and the authentication tag will have a size of 128 bits. A block is defined as a bitstring of at most 128 bits, and a block of less than 128 bits is called incomplete.

The design of Deoxys-II is based on a tweakable block cipher mode called SCT (Synthetic Counter in Tweak), using the exact same encryption, but with a different computation of the verification tag.

In this thesis, two versions of Deoxys-II have been implemented. One with AES as the cipher, and one with Skinny as the cipher. For the AES version, unlike regular AES, the number of rounds is 14 for the version with a key size of 128 bits, and 16 for a key size of 256 bits.

Deoxys with AES has a few differences from a regular AES implementation. The first difference is the number of rounds, which is 14 rather than 10. The second difference is a different set of round constants, or to be precise, that the round constants have been

shifted. The first round constant is `0x2f`, which is the 16th round constant of AES. The first round constant of AES is `0x01`. The key scheduling algorithm is also different from AES, which is not just a consequence of the tweak input being a part of it. The key schedule looks as follows, where K is the key and T is the tweak:

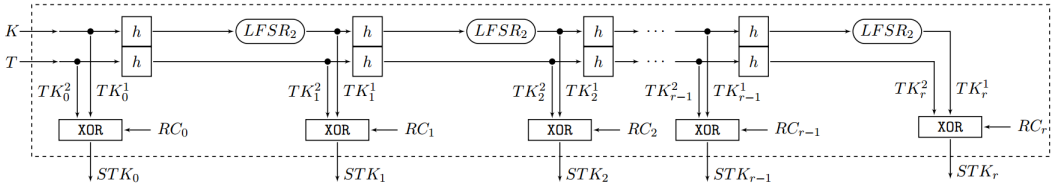


Figure 7.2: The Deoxys Tweakey Schedule [35]. Image modified from original source, to highlight implementation specific details. .

Two boxes can be seen in the key schedule. A box that is simply marked h, and the Skinny LFSR 4.4. The LFSR is only applied to the key input. The box h means the following permutation, where the block is treated as 16 bytes, and the bytes are reordered by position.

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 1 & 6 & 11 & 12 & 5 & 10 & 15 & 0 & 9 & 14 & 3 & 4 & 13 & 2 & 7 & 8 \end{bmatrix}$$

After eight permutations, every element will be back in their original place.
The key schedule round constants are defined as

i	0	1	2	3	4	5	6	7	
RCON[i]	`0x2f`	`0x5e`	`0xbc`	`0x63`	`0xc6`	`0x97`	`0x35`	`0x6a`	
i	8	9	10	11	12	13	14	15	16
RCON[i]	`0xd4`	`0xb3`	`0x7d`	`0xfa`	`0xef`	`0xc5`	`0x91`	`0x39`	`0x72`

Table 7.1: Deoxys round constants.

And the round tweakeys are defined as

$$TK_i = K_i \oplus T_i \oplus RC_i \tag{7.1}$$

Deoxys relies on the cipher used as the engine, and the description so far has been with AES as the engine. If one were to substitute AES with Skinny, Deoxys encryption will have 48 rounds rather than 14, under the assumption that 128-bit keys and tweaks are used. Furthermore, the encryption will follow the steps of Skinny encryption. In short, other tweakable block ciphers can be used as black boxes.

7.1.1 Encryption

The AEAD algorithm Deoxys-II itself goes as follows, where A is the AD, M is the plaintext, N is the nonce, a is the number of complete AD blocks and l is the number of complete plaintext blocks:

```
auth = 0
for i=0 to a-1:
  auth = XOR(auth,ENC(0010||i,A[i],K)) // Inputs to ENC: tweak, plaintext,
      key
end for
if an incomplete block exists:
  auth = xor(auth,ENC(0110||i,A[a]||10*,K))
end if
tag = auth
for i=0 to l-1:
  tag = xor(tag,ENC(0000||i,M[i],K))
end for
if an incomplete block exists:
  tag = xor(tag,ENC(0100||i,M[1]||10*,K))
end if
tag = ENC(00010000||N,tag,K)
tag1 = OR(tag,10*)
for i=0 to l-1:
  C[i] = XOR(M[i],ENC(XOR(tag1,i),0^8||N,K))
end for
if an incomplete block exists:
  C[l] = XOR  ((M[l]||10*),ENC(XOR(tag1,l),0^8||N,K)
end if
```

Listing 7.1: Deoxys encryption algorithm.

Which returns

$$C = (C_0||C_1||\cdots||C_{l-1}||C_l||tag) \tag{7.2}$$

Every single step has a prefix in the tweak input, which is to distinguish between them.

The tag is constructed from encrypted plaintext, rather than the ciphertext, and sent along with the ciphertext. The final encryption ensures that any attempt to modify the AD and the tag falls flat. The approach falls somewhere between EaM and EtM. On one hand, the tag is generated independently of the ciphertext, on the other hand, the tag is generated from encrypted plaintext.

The plaintext, AD and nonce are all used in encryption to ensure that even if a nonce is reused, other factors will figure into the keystream. The plaintext and AD generate the tag, while the nonce is being encrypted in the encryption process. The only way the same keystream can occur is either by coincidence, or if the same AD, plaintext and nonce are all used in encryption, leading to the same input giving the same output, which is a useless result for an attacker. The tweak input also takes a counter to avoid the same key being used over and over again.

7.1.2 Decryption and verification

The decryption algorithms for Deoxys-II goes as follows:

```
tag1 = OR(in_tag,10*)
for i=0 to l-1:
  M[i] = XOR(C[i],ENC(XOR(tag1,i),0^8||N,K))
end for
if an incomplete block exists:
  M[l] = XOR  ((C[l]||10*),ENC(XOR(tag1,l),0^8||N,K)
end if
auth = 0
for i=0 to a-1:
  auth = XOR(auth,ENC(0010||i,A[i],K)) // Inputs to ENC: tweak, plaintext,
     key
end for
if an incomplete block exists:
  auth = xor(auth,ENC(0110||i,A[a]||10*,K))
end if
tagv = auth
for i=0 to l-1:
  tag = xor(tagv,ENC(0000||i,M[i],K))
end for
if an incomplete block exists:
  tag = xor(tagv,ENC(0100||i,M[1]||10*,K))
end if
tagv = ENC(00010000||N,tag,K)
if(tagv != in_tag)
  return failure
else
  return M
end if
```

Listing 7.2: Deoxys decryption algorithm.

The tag is verified if the tag computed from the decrypted ciphertext and the AD matches the tag that was made by the sender.

Deoxys is an inverse-free cipher, meaning that the block cipher decryption function is not used at any point in the decryption process. This is possible because the encryption function is applied to the tag and nonce rather than the plaintext, and the operation that encrypts the plaintext is an operation that is its own inverse, in this case xor. Inverse-free encryption has the advantage that only the encryption function is used, meaning that less hardware is necessary, which is important in lightweight applications. If the encryption function is faster than the decryption function, which is the case for AES, it allows for a faster implementation [10] .

7.2 Theoretical performance

To compute the theoretical performance of Deoxys, I have decided to work under two assumptions, a theoretical one and a practically feasible one. Like in the COLM chapter, the assumptions are unlimited memory (UMA), which is only theoretically feasible, and constant memory (CMA), which is practically feasible. This section assumes certain liberties that will be justified in the implementation section to reduce running time.

7.2.1 First steps and AD processing

The first step of Deoxys is the key schedule, which uses the Skinny LFSR and costs 6 cycles on the different Intel processors, as shown in appendix C.6 . The LFSR can be parallelized, but since there is no setting where a parallelized LFSR would work, I have not done so. Key scheduling itself costs 113 cycles regardless of the processor, the calculation can be seen in appendix C.6 . There is also setting up a zero variable, which is used at various points in the code. Finally, there are the arrays which are described in the implementation section, they cost 78 cycles to set up if eight blocks per iteration, and 46 blocks if four blocks per iteration. This can be seen in appendix C.9. These arrays could have been precomputed, as they only rely on the h permutation, and is a tradeoff that is not further explored under CMA.

Experimentation on the encryption loop, by benchmarking and getting performance numbers, shows that the following model for the pipeline is closest to being correct.

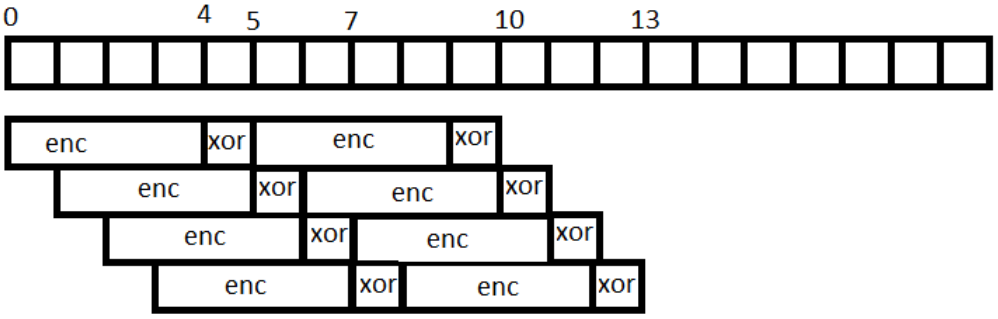


Figure 7.3: Drawing of how the parallelized encryption and xor were calculated.

7.3 Description

As with COLM, it is assumed that Ivy Bridge, Haswell and Broadwell will process eight blocks under a regular iteration, while Skylake will be calculated for both four and eight blocks per iteration.

Processor Step	load	two xor	Encryption	add	xor*	total
Skylake / Kaby Lake 4	1 (2)	3	72	2 (0)	3	81 (80)
Skylake / Kaby Lake 8	2 (4)	6	128	4 (0)	6	142 (140)
Broadwell / Haswell	3 (6)	6	120	4 (0)	4	136 (135)
Ivy Bridge	4 (8)	6	132	4 (0)	4	148 (148)

Table 7.2: Performance numbers in cycles for AD processing in Deoxys under CMA, with UMA in parenthesis.

It shows here that precomputation does very little for the AD processing loop, which is a tendency that will continue throughout the algorithm description. In fact, it makes no difference on the Ivy Bridge processor, where the extra load takes the same amount of cycles the addition would have taken. The xor* operation is the same operation as is defined in the COLM section.

A regular single block encryption is processor dependent, and will cost between 70 and 126 cycles, the calculation can be seen in appendix C.7 . For Deoxys, no distinction is required between an unfinished block and a block that isn't processed in parallel. The performance numbers for the initial steps and the AD processing comes out to, where n_A is the number of AD blocks.

Processor Step	Precomputation	Per iteration	Final blocks
Skylake 4	159 (114)	81 (80)	71 (64)
Skylake 8	191 (114)	142 (140)	71 (64)
Broadwell / Haswell	191 (114)	136 (135)	113 (106)
Ivy Bridge	191 (114)	148 (148)	127 (120)

Table 7.3: Performance figures in cycles for elements of Deoxys AD processing under CMA, with UMA in parenthesis.

Processor Step	total
Skylake 4	$159 + 71(n_A \mod 4) + 81\lfloor \frac{n_A}{4} \rfloor [114 + 80\lfloor \frac{n_A}{4} \rfloor + 64(n_A \mod 4)]$
Skylake 8	$191 + 71(n_A \mod 8) + 142\lfloor \frac{n_A}{8} \rfloor [114 + 142\lfloor \frac{n_A}{8} \rfloor + 64(n_A \mod 8)]$
Broadwell / Haswell	$191 + 113(n_A \mod 8) + 136\lfloor \frac{n_A}{8} \rfloor [114 + 135\lfloor \frac{n_A}{8} \rfloor + 106(n_A \mod 8)]$
Ivy Bridge	$191 + 127(n_A \mod 8) + 148\lfloor \frac{n_A}{8} \rfloor [114 + 148\lfloor \frac{n_A}{8} \rfloor + 120(n_A \mod 8)]$

Table 7.4: Performance formulas for Deoxys AD processing under CMA, with UMA in square brackets. n_A is the number of AD blocks.

7.3.1 Tag generation

As previously written, Deoxys falls somewhere between EtM and EaM, and this component is why it can be argued that Deoxys follows the EaM approach. Tag generation has

no precomputation, except for setting every value in an array to zero, which is assumed to be a free operation. Otherwise, the steps taken in the tag generation are the exact same as in AD processing, meaning that the values for the loop are the exact same as in table 7.2. The only difference is the addition of one final encryption step minus a load and an xor, leading to the following performance numbers, where n_B is the number of plaintext blocks.

Processor Step	Precomp.	Per iteration	Final blocks	Final round
Skylake 4	0	81 (80)	71 (64)	70
Skylake 8	0	142 (140)	71 (64)	70
Broadwell / Haswell	0	136 (135)	113 (106)	112
Ivy Bridge	0	148 (148)	127 (120)	126

Table 7.5: Total performance figures in cycles for Deoxys tag generation under CMA, with UMA in parenthesis. Storage of the tag in the ciphertext is added under final round.

Processor Step	total
Skylake 4	$70 + 71(n_B \mod 4) + 81\lfloor \frac{n_B}{4} \rfloor [70 + 80\lfloor \frac{n_B}{4} \rfloor + 64(n_B \mod 4)]$
Skylake 8	$70 + 71(n_B \mod 8) + 142\lfloor \frac{n_B}{8} \rfloor [70 + 140\lfloor \frac{n_B}{8} \rfloor + 64(n_B \mod 8)]$
Broadwell / Haswell	$112 + 113(n_B \mod 8) + 136\lfloor \frac{n_B}{8} \rfloor [113 + 135\lfloor \frac{n_B}{8} \rfloor + 106(n_B \mod 8)]$
Ivy Bridge	$126 + 127(n_B \mod 8) + 148\lfloor \frac{n_B}{8} \rfloor [127 + 148\lfloor \frac{n_B}{8} \rfloor + 120(n_B \mod 8)]$

Table 7.6: Performance formulas for Deoxys tag generation under CMA, with UMA in square brackets. Result will be in cycles. Storage of the tag in the ciphertext is added under final round. n_B is the number of plaintext blocks.

7.3.2 Encryption

The final step in the Deoxys algorithm is encryption, which has some precomputation. The tag is truncated, and is added directly to the keystream, which takes 24 cycles.

The steps in the encryption loop are similar to the other two steps, with the exception that it is the nonce that is encrypted, and the result of the encrypted nonce is added to the plaintext by xor.

Processor Step	xor	xor in loop	enc	load	xor	store	add	total
Skylake 4	2	72	N/A	1 (2)	2	1	2 (0)	80 (79)
Skylake 8	3	128	N/A	1 (2)	2	1	4 (0)	139 (137)
Broadwell / Haswell	3	120	N/A	3 (6)	3	3	4 (0)	136 (135)
Ivy Bridge	3	132	N/A	4 (8)	3	4	4 (0)	150 (150)

Table 7.7: Performance numbers in cycles for the encryption loop in Deoxys under CMA, with UMA in parenthesis.

Like the other two steps, any block that can't be processed in parallel must be accounted for, leading to the following total performance numbers:

Processor Step	Precomputation	Per iteration	Final blocks
Skylake 4	24	80 (79)	71 (64)
Skylake 8	24	142 (140)	71 (64)
Broadwell / Haswell	24	136 (135)	113 (106)
Ivy Bridge	24	150 (150)	127 (120)

Table 7.8: Total performance in cycles figures for the elements of Deoxys encryption under CMA, with UMA in parenthesis.

Processor Step	Total
Skylake 4	$71(n_B \mod 4) + 81\lfloor \frac{n_B}{4} \rfloor + 24[24 + 80\lfloor \frac{n_B}{4} \rfloor + 64(n_B \mod 4)]$
Skylake 8	$71(n_B \mod 8) + 142\lfloor \frac{n_B}{8} \rfloor + 24[24 + 140\lfloor \frac{n_B}{8} \rfloor + 64(n_B \mod 8)]$
Broadwell / Haswell	$113(n_B \mod 8) + 136\lfloor \frac{n_B}{8} \rfloor + 24[24 + 136\lfloor \frac{n_B}{8} \rfloor + 106(n_B \mod 8)]$
Ivy Bridge	$127(n_B \mod 8) + 150\lfloor \frac{n_B}{8} \rfloor + 24[24 + 150\lfloor \frac{n_B}{8} \rfloor + 120(n_B \mod 8)]$

Table 7.9: Performance formulas for Deoxys encryption under CMA, with UMA in square brackets. Result will be in cycles. n_B is the number of plaintext blocks.

The total performance numbers for Deoxys are, adding the total figures from tables 7.3, 7.5 and 7.9:

Processor Step	Total performance
Skylake 4 (CMA)	$254 + 161\lfloor \frac{n_B}{4} \rfloor + 81\lfloor \frac{n_A}{4} \rfloor + 142(n_B \mod 4) + 71(n_A \mod 4)$
Skylake 4 (UMA)	$209 + 80\lfloor \frac{n_A}{4} \rfloor + 159\lfloor \frac{n_B}{4} \rfloor + 64(n_A \mod 4) + 128(n_B \mod 4)$
Skylake 8 (CMA)	$286 + 281\lfloor \frac{n_B}{8} \rfloor + 142\lfloor \frac{n_A}{8} \rfloor + 142(n_B \mod 8) + 71(n_A \mod 8)$
Skylake 8 (UMA)	$209 + 140\lfloor \frac{n_A}{8} \rfloor + 280\lfloor \frac{n_B}{8} \rfloor + 64(n_A \mod 8) + 128(nb \mod 8)$
Broadwell / Haswell (CMA)	$294 + 272\lfloor \frac{n_B}{8} \rfloor + 136\lfloor \frac{n_A}{8} \rfloor + 226(n_B \mod 8) + 113(n_A \mod 8)$
Broadwell / Haswell (UMA)	$251 + 135\lfloor \frac{n_A}{8} \rfloor + 270\lfloor \frac{n_B}{8} \rfloor + 106(n_A \mod 8) + 212(nb \mod 8)$
Ivy Bridge (CMA)	$328 + 298\lfloor \frac{n_B}{8} \rfloor + 148\lfloor \frac{n_A}{8} \rfloor + 254(n_B \mod 8) + 127(n_A \mod 8)$
Ivy Bridge (UMA)	$267 + 148\lfloor \frac{n_A}{8} \rfloor + 298\lfloor \frac{n_B}{8} \rfloor + 120(n_A \mod 8) + 240(n_B \mod 8)$

Table 7.10: Total performance formulas for Deoxys. Result will be in cycles. n_A is the number of AD blocks and n_B is the number of plaintext blocks.

Deoxys is a two-pass scheme, meaning that the plaintext is processed twice. In this case, it's processed to generate the authentication tag, which is also used in encryption, and then again in the encryption. The i'th ciphertext block is not available until the entire plaintext has been processed once, and up to i plaintext blocks have been processed the second time.

7.4 Implementation

Computing the key schedule every time any plaintext is encrypted would cause a major bottleneck, and since the tweaks are counters concatenated with constants, or counters xor'ed with constants, implementation of the keystream is somewhat different. Recall equation 7.1 that describes every round tweakey, the key part and the round constants do not change, only the tweak part does.

The goal is to do as few computations within the encryption and processing loops as possible, and have these computations be cheap. Exploiting the pipeline is another goal in achieving optimal performance. Using a constant called C_p to describe the number of encryptions done in parallel, precomputation for the different loops of the algorithm will be described in the following section.

7.4.1 Precomputation

- Precompute values of C_p for each possible state under the h permutation, which leads to an array of length 8.
- Precompute values of every natural number from 1 to $C_p - 1$ for each possible state under the h permutation, which leads to a matrix of size $8 \times (C_p - 1)$. This will be referred to as the index counter.
- Precompute value of $0010||0^{124}$ for each possible state under the h permutation, where 0^{124} denotes a bitstring of 124 zeros, which leads to an array of size 8. This is called the tweak array.

The third array are used to contain a multiple of C_p under each permutation, which is updated by addition by the elements of the first array at the end of each iteration. The added values will not conflict with the prefix, as the counter has a maximum length of 64 bits, which are in the least significant end. The values from the second matrix are added to each block, depending on the number of said block. This is xor'ed to the key, and one round of encryption runs.

```
1  tweaks[0] = ad_reg; // ad_reg = {0x20,0x00,0x00,0x00, 0x00,0x00,0x00,0x00, 0
       x00,0x00,0x00,0x00, 0x00,0x00,0x00,0x00}
2  eight[0] = {0x00,0x00,0x00,0x00, 0x00,0x00,0x00,0x00, 0x00,0x00,0x00,0x00, 0
       x00,0x00,0x00,0x08};
3  z_s[0][0] = one; //  one = {0x00,0x00,0x00,0x00, 0x00,0x00,0x00,0x00, 0x00,0
       x00,0x00,0x00, 0x00,0x00,0x00,0x01};
4  for(i=1;i<C_p;++i){ // Table setup
5    z_s[i][0] = _mm_add_epi8(one,zero);
6    for(j=1;j<(C_p-1);++j){
7      z_s[i][j] = _mm_add_epi8(one,z_s[i][j-1]); // Index counter
8      one = _mm_shuffle_epi8(one,h);
9    }
10   tweaks[i] = _mm_shuffle_epi8(tweaks[i-1],h); // Tweak table
11   eight[1] = _mm_shuffle_epi8(eight[i-1],h);  // Values of C_p
```

```
}
for(i=0;i<numblocks_ad;i+=C_p){
  for(j=0;j<C_p;++j){
    mes[j] = _mm_loadu_si128(ad+(i+j)*CRYPTO_KEYBYTES);
  }
  tmp = _mm_xor_si128(keys[0],tweaks[0]);
  mes[0] = _mm_xor_si128(mes[0],tmp);
  for(j=1;j<C_p;++j){
    mes[j] = _mm_xor_si128(mes[j],_mm_xor_si128(tmp,z_s[0][j-1]) );
  }
  for(j=1;j<15;++j){
    tmp = _mm_xor_si128(keys[j],tweaks[j%C_p]);
    for(z=0;z<C_p-1;++z){
      xx[z] = _mm_xor_si128(tmp,z_s[j][z]);
    }
    mes[0] = _mm_aesenc_si128(mes[0],tmp);
    for(z=1;z<C_p-1;++z){
      mes[z] = _mm_aesenc_si128(mes[z],xx[z-1] );
    }
  }
}
```

Listing 7.3: Implementation of Deoxys AD processing algorithm.

7.4.2 Associated data processing

Implementation of AD processing is fairly straightforward, and can be seen in the code below. The AD is loaded, the tweakey is made by xor'ing the tweak to the key, and the rounds are completed. In the last steps, the AD parts are xor'ed, and the tweak is updated by adding C_p, which is named `eight` in the implementation as a legacy. Liberties have been taken compared to the actual code in order to shorten it into something that would fit here.

```
for(i=0;i<numblocks_ad;i+=Cp) {
    mes[j] = _mm_loadu_si128(ad+(i+j)*CRYPTO_KEYBYTES); // for j from 0 to
        C_p - 1

    tmp = _mm_xor_si128(keys[0],tweaks[0]); // keys is the key schedule, and
         tweaks is the tweaks array
    // xx is an intermediate array to clarify what happens in the code, and
        is not strictly necessary
    xx[j] = _mm_xor_si128(tmp,z_s[0][j]); // z_s is the index counter, array
         is for j from 0 to C_p - 2 (recall that index counter only goes to
        C_p - 1

    mes[0] = _mm_xor_si128(mes[0],tmp);
    mes[j] = _mm_xor_si128(mes[j],xx[j-1] ); // for j from 1 to C_p - 1

    for(j=1;j<15;++j){ // Completely unrolled in the code
```

```
12      tmp = _mm_xor_si128(keys[j],tweaks[j%8]);
13      xx[z] = _mm_xor_si128(tmp,z_s[j%8][z]); // for z from 0 to C_p - 2
14
15      mes[0] = _mm_aesenc_si128(mes[0],tmp);
16      mes[z] = _mm_aesenc_si128(mes[z],xx[z-1] ); // for z from 0 to C_p - 1
17    }
18    tweaks[j] = _mm_add_epi64(tweaks[j],eight[j]); // for j from 0 to 7
19    auth ^= mes[j]; // for j from 0 to C_p
```

Listing 7.4: AD processing code.

7.4.3 Tag generation

Tag generation uses the exact same arrays and method as AD processing, with the difference being that the third array under AD processing is set to contain zero values instead, and not permuted since permuting a zero would result in the same output as the input.

7.4.4 Encryption

Encryption follows the same process to a certain point, but with the difference that the truncated tag is used as a tweak input. Recomputing the key schedule is not an option, but permuting the truncated tag and xor'ing it to the keystream allows for reuse of the previous arrays and methods without further change.

```
tmp = one_tag;
for(i=0;i<7;++i)
{
  keys[i] = _mm_xor_si128(keys[i],tmp);
  keys[i+8] = _mm_xor_si128(keys[i+8],tmp);
  tmp = _mm_shuffle_epi8(tmp,h);
}
keys[7] = _mm_xor_si128(keys[7],tmp);
```

Listing 7.5: Tag addition code.

Otherwise the process is the same as under AD processing, with the exception that the nonce, not the message, is encrypted, and the message is added at the end by xor.

7.5 With Skinny

I have also implemented Deoxys with Skinny rather than AES, using a fairly different method of implementation. Unlike Deoxys with AES, which I programmed from scratch, I was provided with a complete Skinny implementation [40], and decided to expand it from 128-bit Skinny with no tweak, to 128-bit Skinny with a 128-bit tweak. Skinny is

designed as a tweakable block cipher, meaning that the key schedule for Skinny overrules the key schedule for Deoxys, and does not need further work. The difference between the two are the number of rounds and the addition of the tweak part of the key, the latter of which I had to write.

In the implementation of Skinny itself, the plaintext data and the key are represented as arrays of characters, which are shuffled around in order to make bitslicing work. The tweak can be precomputed entirely, since it does not rely on either plaintext or AD, except for when the truncated tag is used as the tweak, but the truncated tag required packing in any case, and the packed truncated tag is added to the tweak by xor. This is taken advantage of in the implementation, though it can potentially cause memory problems if the message or AD is long, because it will make the tweak long.

CHAPTER 8

Performance

Both ciphers were on three different processors, under similar conditions, using both four and eight blocks per iteration. All tests use an ideal case, where the message length perfectly aligns with the number of blocks being processed in parallel and accounts for the special case of the final block in COLM. Additionally, $COLM_0$ and Deoxys have been tested on an ARM mobile processor, using four blocks per iteration.

8.1 Testing method

For testing, a small program was written to generate random plaintexts, which was only composed of lowercase letters. Limiting the plaintext to the 26 lowercase letters served the purpose of a quick verification, as it is unlikely that a ciphertext or a failed decryption would lead to only lowercase letters. The exception to this rule was the 2048 byte plaintext, which was a legible text copied from the web and padded [48].

The programs were run in 1000 batches of 100000 trials on Skylake and Kaby Lake, where the performance was represented as the average performance, where internal C functions were used to measure the amount of cycles used to process the algorithm, which were divided by the message length and the number of trials to produce the number of cycles per byte. The average of those 1000 batches are then used to represent the performance of a scheme. The timing script was given [53], where the only work required was to set the number of trials and batches. On AMD and ARM, the same process applied, except that it was only 200 batches. Testing Deoxys with Skinny was a bit different. The Skinny implementation was provided to me [40], along with a benchmarking script, and Deoxys was built around this implementation. The testing script is similar to the testing script provided for the other implementation, with the difference being that the key figure from the trials is the median, not the average. Due to initial testing providing some discouraging results, the tests were 100 batches of 25000 trials.

8.2 Subprocesses

Certain subprocesses of the different schemes have been tested to understand the discrepancies between theoretical and practical performance. Tests of subfunctions have

used the same timing script as tests of algorithms, with the twist that it has been 10 batches of 10^7 trials.

8.2.1 COLM field doubling

The theoretical running time of the COLM field doubling was said to be 9, which is calculated in appendix C.2. Knowing that doubling works on 16 bytes, the performance should be $\frac{9}{16} \approx 0.56$ cycles per byte (cpb). The performance turned out to be faster. The test was performed on 4096 bytes of data in order to reduce overhead and achieve the most realistic result. On average, this function takes 0.129 cpb, and the exact figure can be seen in listing D.1 . In conclusion, this function actually has the performance of $0.129 \cdot 16 \approx 2$ cpb.

8.2.2 $COLM_0$ encryption loop

The theoretical performance for the $COLM_0$ encryption loops, with four and eight blocks per iteration respectively were:

$$\begin{aligned} perf_4 &= 209 \\ perf_8 &= 409 \end{aligned} \tag{8.1}$$

These numbers don't quite hold up, as it's proven that field doubling takes much less time than in these calculations. It also assumes the idealized case, number of blocks being perfectly aligned to the implementation, however that one is indeed in effect in the tests. Accounting for field doubling being substantially more effective, meaning that it is only counted as 2 cpb and the ρ function is only counted as 4 cpb, the actual numbers become 125 and 241 respectively. In terms of cycles per byte, the expected performance numbers for the encryption loop become:

$$\begin{aligned} \frac{125}{64} &\approx 1.95 \\ \frac{241}{128} &\approx 1.88 \end{aligned} \tag{8.2}$$

For four blocks, the performance is tested to be 2.148 cpb, and more exact results can be found in listing D.2 . For eight blocks, the performance is tested to be 2.105 cpb and more exact results can be found in listing D.3 .

Two differences are visible here. The difference between 4 and 8 blocks per iteration isn't as large as expected, and both loops are a bit slower after correcting for the effect of field doubling.

8.2.3 Deoxys encryption loops

The performance of the Deoxys encryption loops are estimated to be

$$\begin{aligned} perf_4 &= 81 \\ perf_8 &= 142 \end{aligned} \tag{8.3}$$

Like with the $COLM_0$ encryption loops, this assumes the idealized case where the number of blocks align perfectly with the implementation, which is in effect in the benchmarking. Unlike with $COLM_0$ there is no twist that change the numbers. The numbers in terms of cycles per byte become

$$\begin{aligned} \frac{81}{64} &\approx 1.27 \\ \frac{142}{128} &\approx 1.11 \end{aligned} \tag{8.4}$$

It should be noted that two loops with this expected performance are run sequentially, for tag generation and encryption respectively, and the expected performance of Deoxys is at least double of what is stated here. The actual performance is 1.22 cpb, where more exact numbers can be seen in listing D.4 . For eight blocks, the performance is 1.12 cpb. This means that the Deoxys loops align almost exactly with the theoretical estimate, where more exact numbers can be seen in listing D.5.

8.2.4 Skinny encryption function

The Skinny encryption function was tested using the benchmarking script that was supplied, running 10 batches of 2^{21} trials. The Skinny implementation itself would take 4.05 cpb, where more exact figures can be found in listing D.6 .

Requiring three passes of the encryption function, the performance of Deoxys with Skinny is expected to start at 12.15 cpb plus overhead, and become lower as the AD size is fixed while the message size isn't.

8.3 $COLM_0$

Testing of $COLM_0$ followed the testing procedure described, except that another block of plaintext and AD have been added to account for the implementation specific detail that the last block processed is treated as a special case, due to being treated in a special manner in the algorithm. This means that every message has a length of $2^r \cdot 128 + 16, r \in 0, ..., 7$.

All message lengths are written as a length that fits the formula $2^r \cdot 128, r \in 0, ..., 7$ for consistency, but for COLM, both versions, it is implied that the message length is

actually $2^r \cdot 128 + 16$ throughout the report. The performance on the different processors with various message sizes for $COLM_0$ with four blocks per iteration can be seen in figure 8.1.

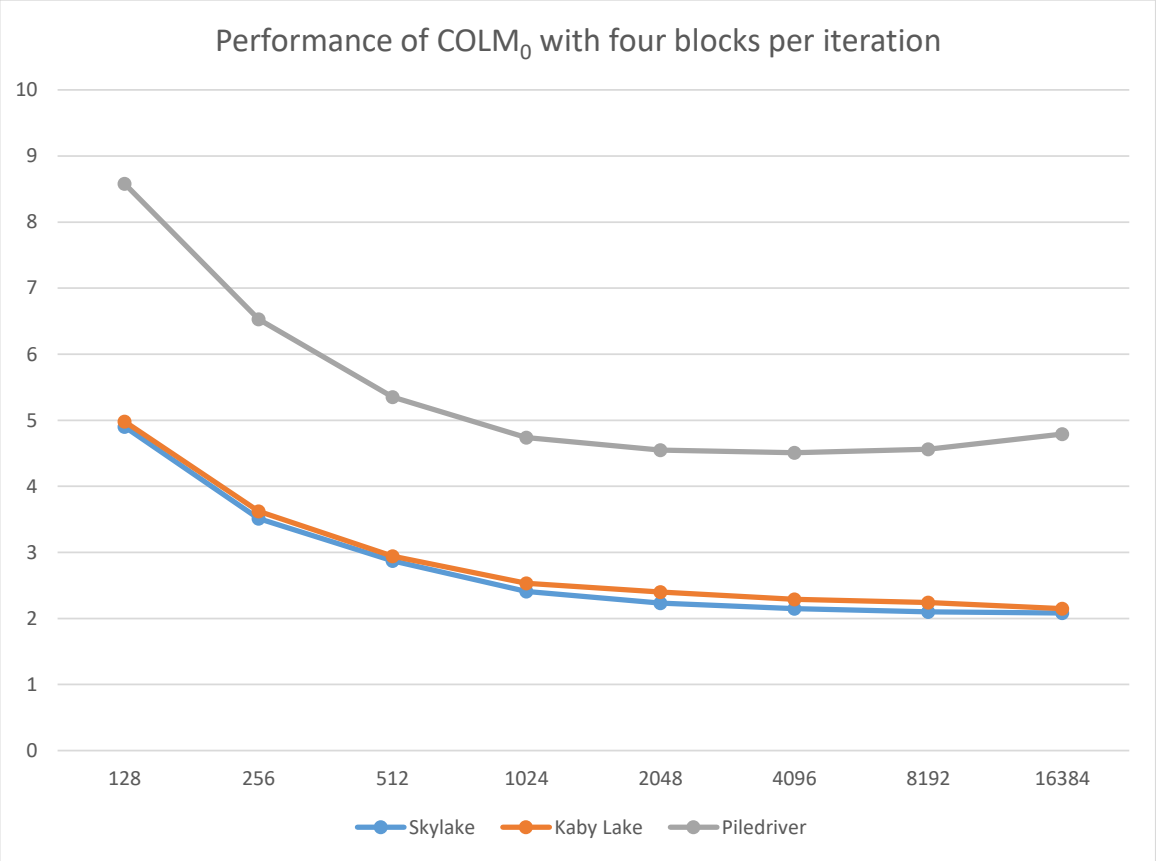


Figure 8.1: Performance of $COLM_0$ with four blocks processed per iteration. Exact numbers can be seen in table A.1 .

Variance in the batches that were tested can be seen in table 8.1

Processor \length	128	256	512	1024	2048	4096	8192	16384
Skylake	$6.1 \cdot 10^{-5}$	$2.9 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$	$6.9 \cdot 10^{-5}$	$8.7 \cdot 10^{-5}$	$1.1 \cdot 10^{-5}$	$2.8 \cdot 10^{-5}$	$1.7 \cdot 10^{-5}$
Kaby Lake	$7 \cdot 10^{-4}$	$2.4 \cdot 10^{-4}$	$8.2 \cdot 10^{-5}$	$3.4 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$7.6 \cdot 10^{-6}$	$9 \cdot 10^{-6}$	$1.4 \cdot 10^{-5}$
Piledriver	0.147	0.2	0.015	0.0061	0.036	0.017	0.022	0.037

Table 8.1: Variance table of the performance measurements of $COLM_0$ with four blocks per iteration.

It shows here that the performance becomes better for bigger message sizes, which is a result of precomputation and final block processing being less significant compared to the computation loop. It also shows that the Kaby Lake processor is a little bit slower

than the Skylake processor, which is almost even at the smallest and largest message size, while the difference is a bit larger for intermediate message sizes. It should be noted that constant time field doubling benchmarks as being significantly faster than first calculated, taking only two cycles instead of 9, which has the effect that the ρ function is also significantly faster, taking a huge chunk out of the encryption loop.

Generally, the variance in the results displays that it is easier to get a consistent result if the trial takes a longer time, which is especially apparent on Kaby Lake and Piledriver. While the variance is larger on Kaby Lake than on Skylake, it is still small enough that the results are reliable. The variance on Piledriver also underlines the idea of an old, ill-maintained computer that may have background processes that eat processing power will not always produce good results. While the results for larger messages can be considered somewhat consistent, the results for larger messages have large variance. Comparing the results to the theoretical performance, we get

$$\begin{aligned} 552 + 209\left\lfloor\frac{1024}{4}\right\rfloor + 116(1024 \mod 4) + 40\cdot 0 + 85\left\lfloor\frac{8}{4}\right\rfloor + 52(8 \mod 4) + 63\cdot 0 \\ = 552 + 209\cdot 256 + 85\cdot 2 = 54226 \\ \frac{54226}{16384} = 3.30969 \\ 552 + 209\left\lfloor\frac{8}{4}\right\rfloor + 116(8 \mod 4) + 40\cdot 0 + 85\left\lfloor\frac{8}{4}\right\rfloor + 52(8 \mod 4) + 63\cdot 0 \\ = 552 + 209\cdot 2 + 85\cdot 2 = 1140 \\ \frac{1140}{128} = 8.90625 \end{aligned} \tag{8.5}$$

The first and obvious observation is that the scheme outperforms the theoretical estimate. This is not particularly surprising as the theoretical estimate is proven to overestimate the field doubling step, which is in fact much cheaper than estimated. Adjusting the theoretical performance to reflect the observed performance of the field doubling allow, we obtain an estimate of 1.997 cpb for the test containing the largest message and 7.59 cpb for the smallest message. For the large message, the performance is a bit worse, while the performance is quite a bit better for the small message.

The performance on the different processors with various message sizes for $COLM_0$ with eight blocks per iteration can be seen in figure 8.2.

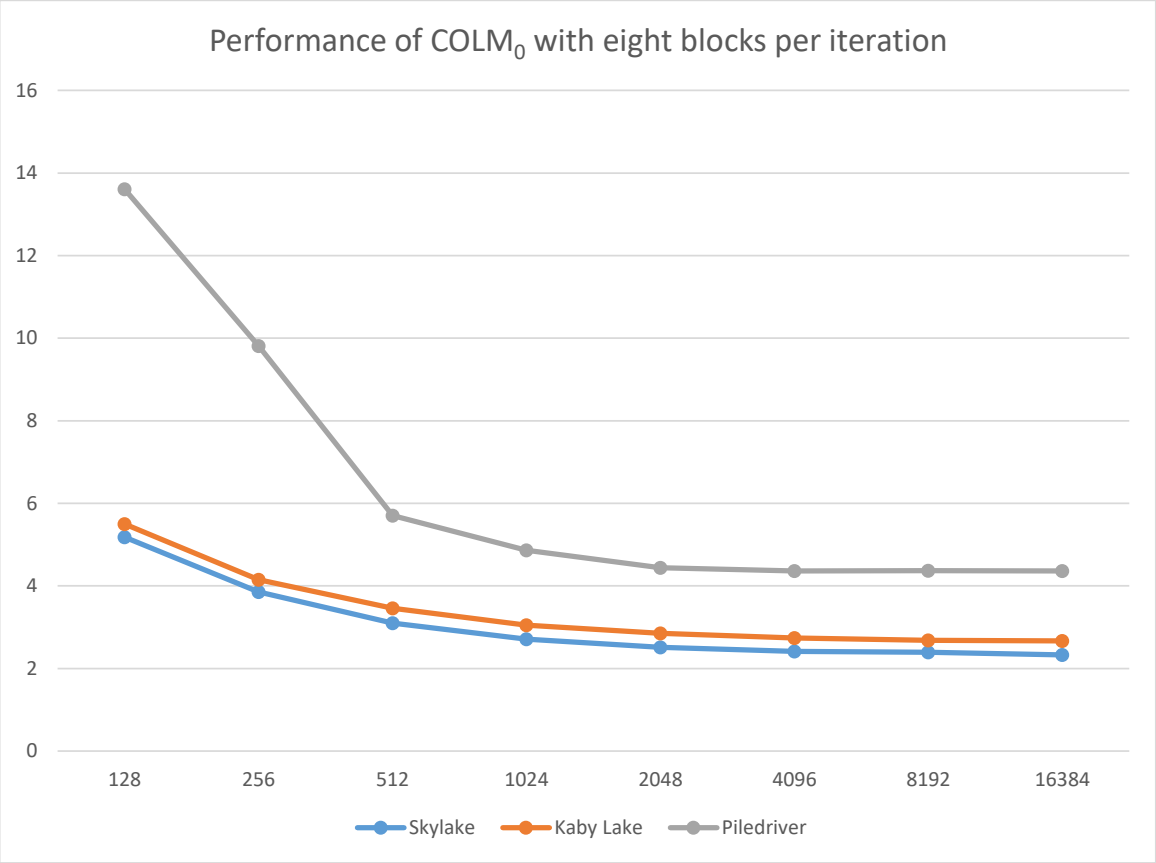


Figure 8.2: Performance of $COLM_0$ with eight blocks processed per iteration. Exact numbers can be seen in table A.2 .

Despite the theoretical calculations saying otherwise, 8 blocks per iterations is consistently slower on the Intel processors. Kaby Lake is also consistently slower than Skylake by around 0.3 cycles per block. The Piledriver processor has better performance numbers for eight blocks per iteration, which implies that full pipeline utilization is not achieved on the Piledriver processor with only four blocks per iteration.

8.4 $COLM_{127}$

The tests of $COLM_{127}$ were identical to the tests of $COLM_0$ in terms of input. The performance on the different processors with various message sizes for $COLM_{127}$ with four blocks per iteration can be seen in figure 8.3.

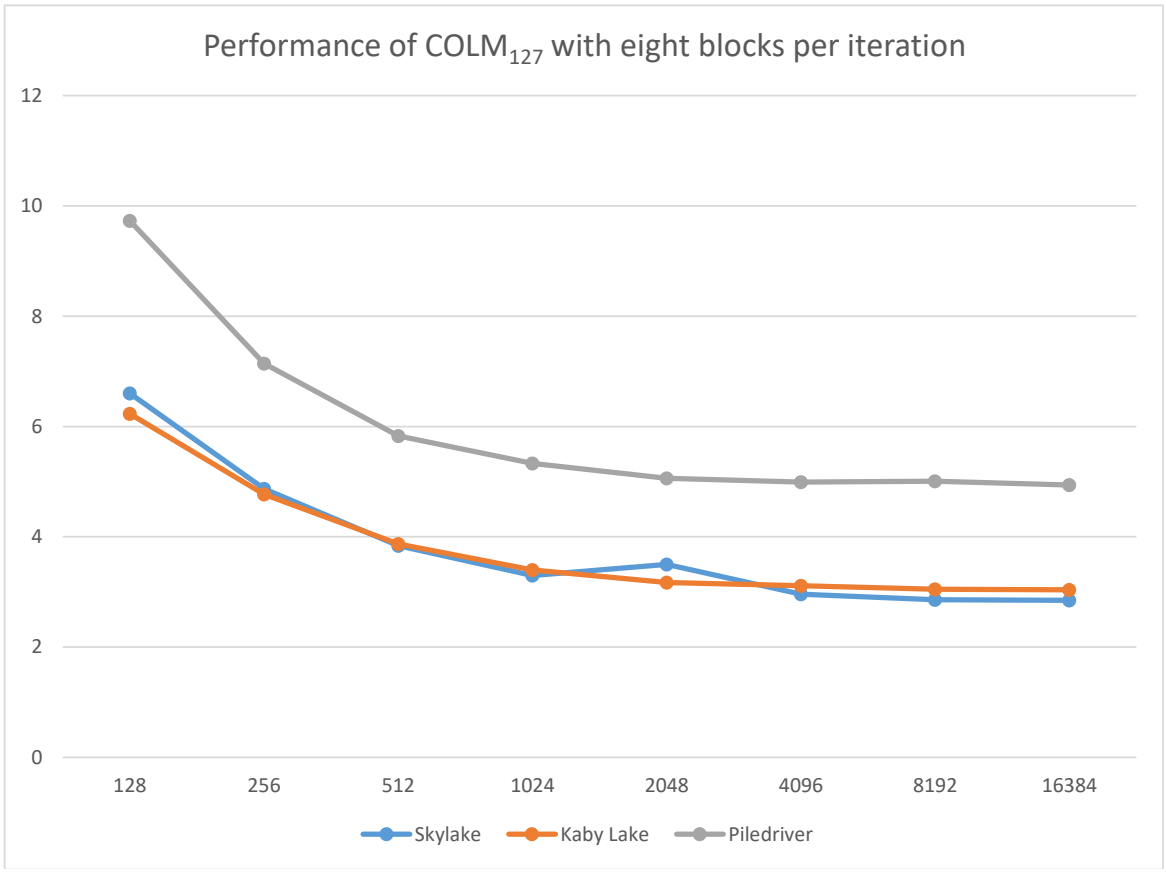


Figure 8.3: Performance of $COLM_{127}$ with four blocks processed per iteration. Exact numbers can be seen in table A.3 .

Unsurprisingly, the performance numbers for $COLM_{127}$ are consistently a bit worse than the performance numbers for $COLM_0$. The performance on Skylake for the 2048 byte message is worse than the performance for the 1024 byte message, which may relate to the extra encryption. It could also be caused by a failed branch prediction, as every iteration of the encryption loop except the final one has the check for an intermediate tag evaluate to false.

What is notable is the difference in results for plaintext of length 1024 byte and below, where the $COLM_{127}$ process is indistinguishable from the $COLM_0$ process. From a theoretical standpoint, the performance should be identical, but that is far from the case. This is likely due to the fact that modular operations are used to determine whether an intermediate tag is to be computed, which is not an intrinsic operation, meaning that it isn't counted, but since a division operation is performed to produce the value [36], and division is an expensive operation compared to other arithmetic operations [51] (to a point where Intel have patented their own division algorithm [49]).

Kaby Lake and Skylake appear to follow each other quite well for the longer messages, with Skylake being slightly faster.

The performance on the different processors with various message sizes for $COLM_{127}$

with eight blocks per iteration can be seen in figure 8.4.

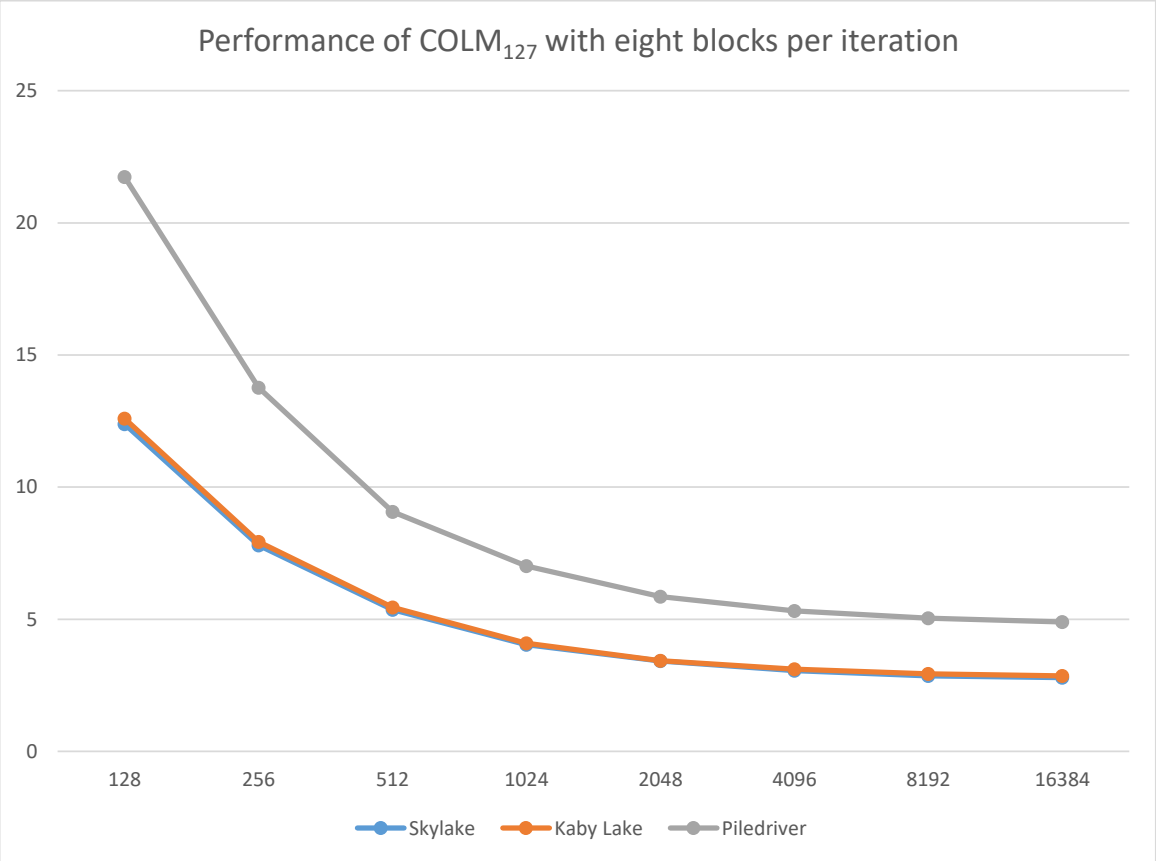


Figure 8.4: Performance of $COLM_{127}$ with eight blocks processed per iteration. Exact numbers can be seen in table A.4 .

Here, it shows that while faster for longer messages, the performance of $COLM_{127}$ with eight cycles per iteration breaks down for messages smaller than 2048 bytes, reaching over 10 cpb for the smallest message on both Kaby Lake and Skylake, and over 20 cpb on Piledriver.

8.5 Deoxys

Like with COLM, the tests of Deoxys were ideal scenarios, in this case fixed AD of 128 bytes and a message length that perfectly aligns with the number of blocks being processed in parallel, meaning that the message length is $2^r \cdot 128, r \in 0, ..., 7$. The results obtained for Deoxys with four blocks per iteration can be seen in figure 8.5.

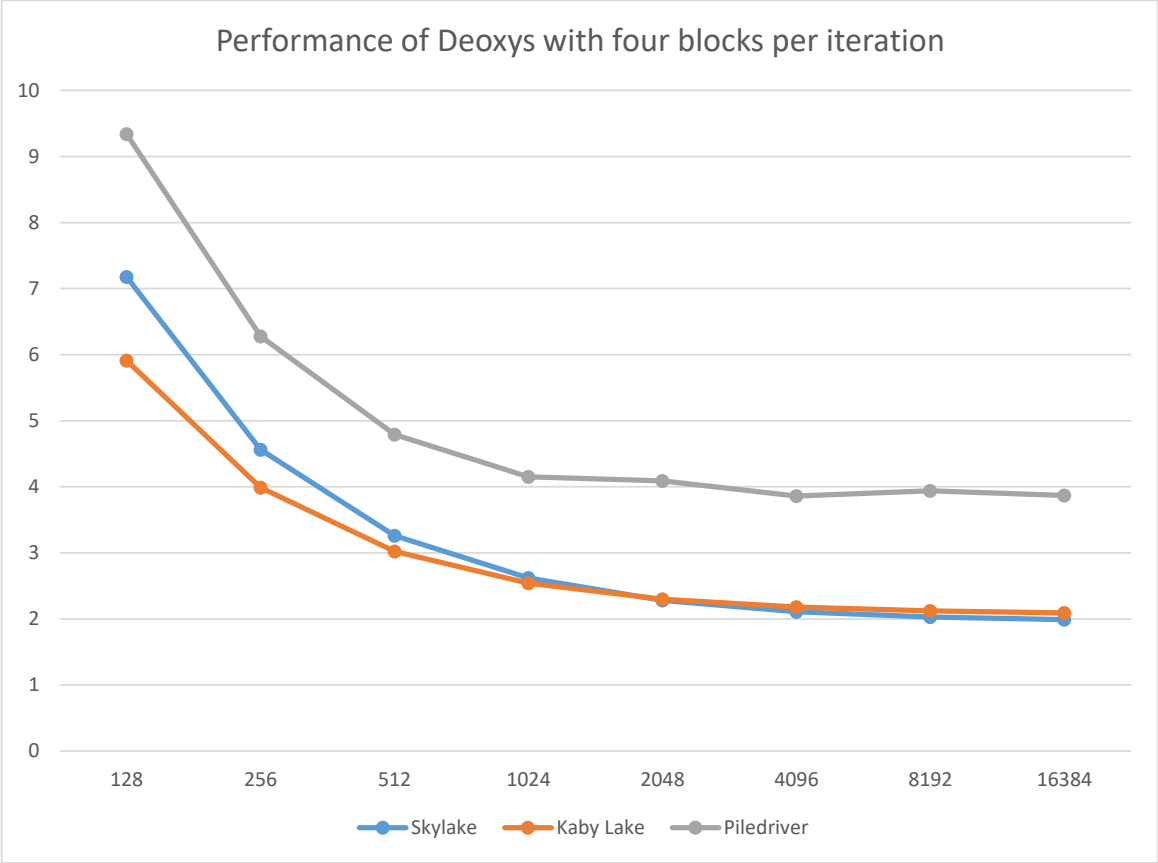


Figure 8.5: Performance of Deoxys with four blocks processed per iteration. Exact numbers can be seen in table A.5 .

It can be seen here that despite the Kaby Lake processor not being completely idle while running the scheme, Kaby Lake outperforms Skylake for smaller message sizes, while the opposite is true for larger message sizes. For both processors, the performance gets better for larger messages, which is not a surprise. Consider the formula for performance in table 7.10. The larger number of bytes, the less any other term than plaintext processing matters. The Piledriver shows as being substantially worse on all accounts, with the 4096 byte message actually producing the best results. The Piledriver processor also displays a larger variance in the test results, as can be seen in table A.10 in appendix.

If inserting the corner cases, messages of 128 bytes and 16384 bytes respectively, we get the following number of cycles per byte as expected performance. Recall that there are 16 bytes to a block.

$$\begin{aligned} 287 + 161 \left\lfloor \frac{1024}{4} \right\rfloor + 81 \left\lfloor \frac{8}{4} \right\rfloor + 144(1024 \mod 4) + 72(8 \mod 4) \\ = 287 + 161 \cdot 256 + 81 \cdot 2 = 41665 \\ \frac{41665}{16384} = 2.54303 \\ 287 + 161 \left\lfloor \frac{8}{4} \right\rfloor + 81 \left\lfloor \frac{8}{4} \right\rfloor + 144(8 \mod 4) + 72(8 \mod 4) \\ = 287 + 161 \cdot 2 + 81 \cdot 2 = 771 \\ \frac{771}{128} = 6.02343 \end{aligned} \tag{8.6}$$

These number is quite a bit higher than the actual performance, which is difficult to explain. Benchmarks of the encryption function and give the conclusion that the theory matches practice quite well, but in the case of a practical application, the scheme is substantially faster than it is theoretically supposed to be. This result becomes even stranger, considering that I have also benchmarked the encryption loop with everything else stripped away, which has the expected performance. With an encryption loop and a tag generation loop that relies on the message, both of which individually benchmark to the theoretical performance, as can be seen in appendix C.8, this result comes across as impossibly good.

The results obtained for Deoxys with four blocks per iteration can be seen in figure 8.6.

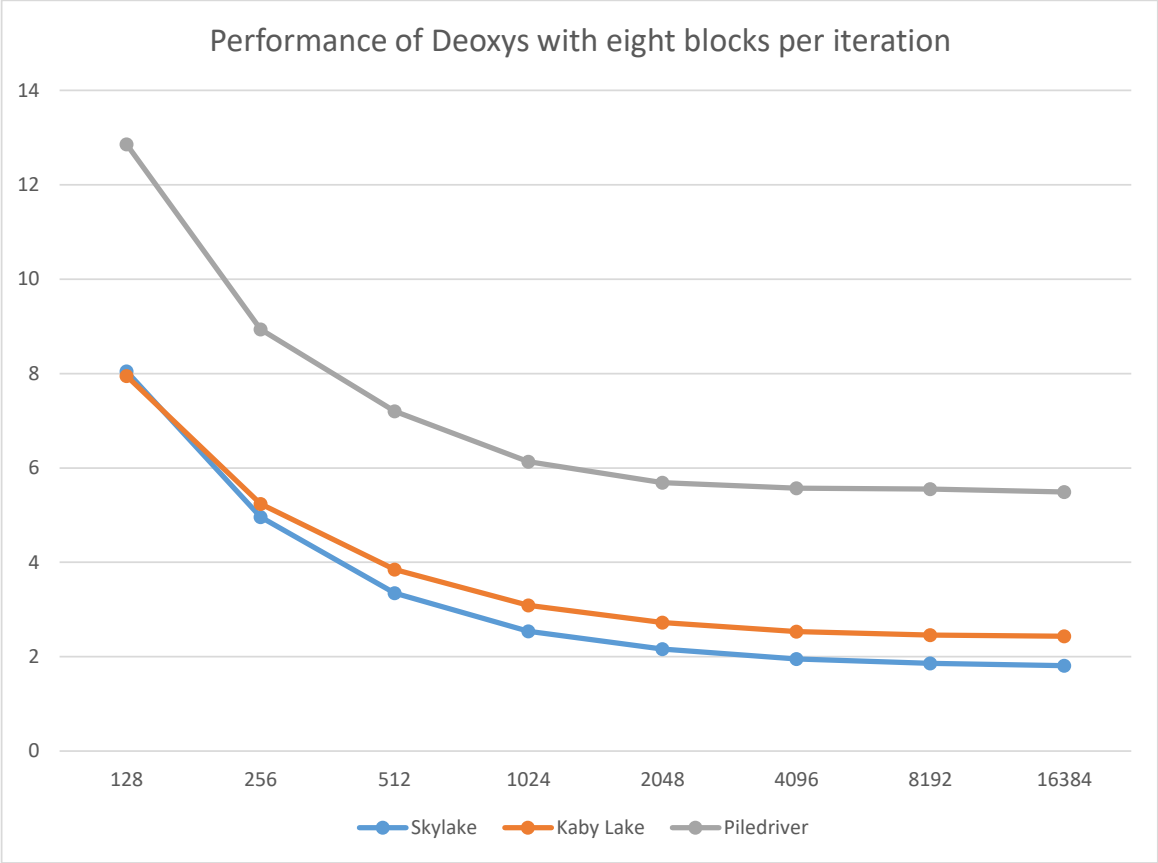


Figure 8.6: Performance of Deoxys with eight blocks processed per iteration. Exact numbers can be seen in table A.6 .

It shows that having eight blocks per iterations is faster for message sizes of 1024 bytes and above, while slower for message sizes of 512 bytes and below. The performance in terms of cycles per byte gets better for larger messages for the same reason as above. What's notable here is that while a message-size dependent difference is displayed for Skylake, Kaby Lake is consistently worse with eight blocks per iteration as opposed to four.

A general observation is that aside from COLM with eight blocks per iteration, Skylake consistently outperforms Kaby Lake, which could mean that the Skylake processor is more idle, the compiler on Skylake generates more optimal code, which I especially believe is the case for Deoxys with eight blocks per iteration. The Kaby Lake processor seems to have an advantage for smaller message sizes, while Skylake processes larger message sizes. In the case of COLM with eight blocks per iteration, Kaby Lake consistently outperforms Skylake. The variance tables show that the results on Skylake are the most reliable, while the Kaby Lake results have little variance as well, making the results reliable as well. The Piledriver processor is consistently outperformed. This could be the result of an old computer not being as good as the compiler thinks it is, the result of an older compiler producing non optimal code and the result of a not fully

idle processor.

8.6 Compiler differences

The first compiler used for Kaby Lake tests (GCC 4.9.2) was a bit older than the one used on the Skylake tests, and was updated to GCC 5.2.0 to match the GCC version used on Skylake as closely as possible. Despite that, the results don't necessarily get faster from the change in compiler. The results on Deoxys with eight blocks per iteration using this compiler were much slower than expected, and change of compiler took off around 1.22 cpb off. Similarly, the compiler update made Deoxys with four blocks per iteration 0.13 cpb faster. COLM with four blocks per iteration is also slightly faster, if only by 0.015 cpb, which may be noise. $COLM_0$ with eight blocks per iterations is substantially slower on the new compiler by 0.6 cpb. All benchmarks on Kaby Lake with GCC 4.9.2 as the compiler can be seen in table 8.2. For $COLM_{127}$, little difference is seen between the older and the newer compiler, the newer compiler producing slightly faster code. The test results on Kaby Lake using GCC 4.9.2 can be seen in table 8.2 .

Scheme \length	128	256	512	1024	2048	4096	8192	16384	GCC 5.2.0 results
$COLM_0$ (4)	5	3.64	2.96	2.58	2.46	2.35	2.29	2.17	table A.1
$COLM_0$ (8)	4.68	3.43	2.8	2.41	2.21	2.13	2.06	2.07	table A.2
$COLM_{127}$ (4)	6.36	4.81	3.89	3.41	3.19	3.13	3.07	3.05	table A.3
$COLM_{127}$ (8)	12.62	7.99	5.53	4.17	3.51	3.19	3.01	2.93	table A.4
Deoxys (4)	6.31	4.18	3.19	2.71	2.45	2.35	2.28	2.26	table A.5
Deoxys (8)	9.79	6.82	5.32	4.59	4.21	4.03	3.94	3.91	table A.6

Table 8.2: Performance of schemes on Kaby Lake using GCC 4.9.2 as the compiler, all numbers are in cycles per byte.

8.7 Deoxys with Skinny

Deoxys with Skinny was tested with message sizes of $2^r \cdot 1024, r \in 0, ..., 7$ and a fixed AD size of 1024. The deviation from the regular testing procedure was caused by the fact that Skinny processes 64 blocks at the time, meaning that the smallest possible message for fully parallel execution would be 1024 bytes. The compiler used in the Kaby Lake test was GCC 5.2.0. The results can be seen in table 8.3.

Processor \length	1024	2048	4096	8192	16384	32768	65536
Skylake	50.01	42.32	30.54	27.52	25.89	25.21	24.91
Kaby Lake	68.06	56.97	43.22	38.79	36.49	35.45	34.97

Table 8.3: Performance of Deoxys with Skinny, all numbers are in cycles per byte.

These numbers do not look efficient at all. There can be multiple causes for this. The packing used to convert the message into a bitsliced format is a heavy operation, which precomputation of tweaks sought to reduce. The precomputed tweaks and another heavy data structure in the implementation pushes the used memory to over 50 KB for the smallest message, leading to potential cache misses, which forces the processor to fetch data from slower memory, reducing speed. There may also be other performance reducing factors in the implementation.

The discrepancy between Skylake and Kaby Lake seems huge, but it could very well be indicative of the discrepancy between the testing environments. The Skylake tests were performed on a dedicated testing machine that was almost fully idle. The Kaby Lake tests were performed on a computer running Windows 10, and while as many intrusive processes as possible were terminated to reduce interference, there were still background processes which may have interfered with the processor and in particular the memory management to a higher degree than on the Skylake processor.

8.8 ARM

Like on the Skylake and Kaby Lake processors, the scenarios tested on ARM were idealized. Any references to COLM in this section is implicitly $COLM_0$. The COLM tests used a message length which was a multiple of 64, the number of bytes used in four blocks plus an additional block at the end, and the Deoxys tests used a message length which was a multiple of 64. The performance numbers can be read in table 8.4.

Scheme \length	128	256	512	1024	2048	4096	8192	16384
COLM	23.54	15.43	10.95	8.57	7.41	6.78	6.45	6.3
Deoxys	13.4	9.3	7.34	6.31	5.78	5.53	5.41*	5.37*

Table 8.4: Average performance of schemes on ARM, all numbers are in cycles per byte. Two results are marked with an asterisk, meaning that a smaller sample size was considered. This is due to an overzealous testing script. The average performance from the full test of a message length of 8192 bytes is 5.53 cpb, and the average performance from the full test for a message length of 16384 6 cpb.

The schemes are shown to be somewhat slower on the ARM processors than on the other processors. There are a few possible reasons for this. The compiler is a cross compiler on the Skylake machine, and the code may be a suboptimal for the ARM processor. The ARM processor may also take longer time to process the cryptographic operations, the xors in Deoxys or the field doubing in COLM. Of further note are the Deoxys tests with large message sizes, which increase in number of cycles per byte over the trials, which can be seen in the below graphs.

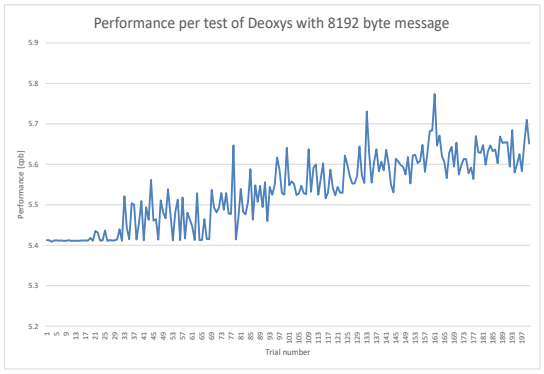


Figure 8.7: Performance over trials of Deoxys with a 8 KB input.

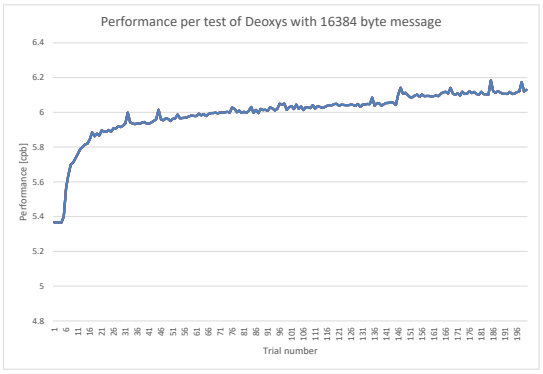


Figure 8.8: Performance over trials of Deoxys with a 16 KB input.

The first five tests of COLM with a message length of 16384 bytes have an average performance of 5.37 cpb, and the performance gets progressively worse afterwards, with every trial after the 88th being over 6 cpb. A similar, though not as large, tendency can be seen for the test with a message length of 8192 bytes, where the first 32 trials have an average performance of 5.41 cpb. These numbers are closer to the realistic case, where the encryption algorithm is only run once. By comparison, the encryption algorithm has been run $5 \cdot 10^5$ times after the fifth test, and $3.2 \cdot 10^6$ times after the 32nd test, which is unlikely to happen in a realistic scenario.

The spikes are caused by the uncertainty in the measurement, and would appear for tests of any scheme on any processor, especially on the Piledriver. There are two options for the increase of cpb over trials. One is thermal throttling, meaning that the processor speed is decreased because of the heat it generates. This would not be a problem for neither Skylake, Kaby Lake nor Piledriver because they have better ventilation than the phone. Another option is that the processor believes that some memory still needs to be used after ending the encryption loop, and it is kept active. The latter possibility is found to be unlikely due to a second trial producing similar results in terms of performance with no change in memory use throughout the test described in appendix E.

8.9 Observations and summary

Of general observations, it is shown that $COLM_{127}$ is slower than $COLM_0$. This was to be expected as $COLM_{127}$ is $COLM_0$ with intermediate tags, and the extra steps should affect performance, even if the theoretical calculation state otherwise. $COLM_{127}$ was also shown to perform much worse for short messages using eight blocks per iteration as opposed to four, while being slightly faster for longer messages. It shows in table 8.5 that if a message is over 2048 bytes, which is the minimum message length for generation of

an intermediate tag, less performance is lost using $COLM_{127}$ compared to using $COLM_0$, while the opposite is true for smaller messages.

Processor \length	128	256	512	1024	2048	4096	8192	16384
Skylake (4)	1.7	1.36	1.02	0.9	1.27	0.81	0.77	0.78
Kaby Lake (4)	1.35	1.15	0.93	0.87	0.77	0.82	0.81	0.89
Skylake (8)	7.21	3.95	2.26	1.32	0.91	0.64	0.47	0.46
Kaby Lake (8)	7.09	3.65	1.99	1.04	0.58	0.37	0.25	0.19

Table 8.5: Increase in cycles per byte using $COLM_{127}$ compared to $COLM_0$ with the same input and number of blocks per iteration.

A potential improvement to $COLM_{127}$, or more generally, $COLM_\tau$ would be to align τ with the number of iterations. The current value, $\tau = 127$, is a prime, meaning that the number of iterations must be checked every time a tag is generated. A change to this process could be to set $\tau = 124$ or $\tau = 120$, substituting the iterations with two layers of iterations and eliminating the check.

A general observation is that on the tests where all other elements are equal; compiler, processor and number of iterations, Deoxys performs better than $COLM_0$. Recall the performance formulas for COLM (table 6.7) and Deoxys (table 7.10). In the tests performed, the bulk of the work is processing blocks in parallel. Serial processing of blocks, where COLM theoretically outperforms Deoxys, has not been tested. According to calculations, COLM contains more overhead, but $COLM_0$ still outperforms Deoxys for messages of less than 2048 byte with four blocks per iteration and for messages less than 1024 byte for eight blocks per iteration.

Of further note is that all instances of Deoxys, except Deoxys with eight blocks per iteration on Kaby Lake, perform better than what should be possible. The encryption loop is processed twice, which should lead to a performance of at least 2.44 cpb for four blocks per iteration, and 2.24 cpb for eight blocks per iteration. This means that the compiler or the processor does something that allows the Deoxys implementation to be faster than twice the encryption loop, when the program is stripped down to the encryption loop only.

On the Piledriver architecture, the best performance figures are mostly seen with four blocks per iteration, except for longer messages using $COLM_0$ as the encryption algorithm. The numbers are slower than in an ideal scenario, likely due to a sub-optimal testing environment and the fact that it is in an ill-maintained, old computer.

Another observation is that Deoxys with Skinny performs badly compared to the benchmark and the expected performance. The working hypothesis for why this is the case, is that memory accesses take up a lot of time, and having two 25 KB data structures that are accessed on each iteration, which exceed the L1 cache of both the Skylake and the Kaby Lake machine, means that cache misses are bound to happen, which can be detrimental to performance. This is especially the case for the Kaby Lake, which ran on a suboptimal, though consistent, testing environment. There may be other issues, and it should be noted that all array accesses in the algorithm are sequential, making branch

prediction easier. The performance estimate of 12.15 plus overhead for the smallest message size isn't even close to being reached for the largest message size.

CHAPTER 9

Further work

Finding out why Deoxys with Skinny produces performance numbers that are worse than expected would be the first point to investigate. The working hypothesis would be that memory optimization is required to improve upon the performance, a hypothesis which is backed by Skinny being much slower on Kaby Lake, as can be seen in table 8.3, which was a consistent testing environment, but never fully idle.

Another point to work on would be to implement $COLM_{120}$ or $COLM_{124}$ and test that against both $COLM_0$ and $COLM_{127}$. While the frequency of intermediate tags will go up, the omission of the check for an additional tag may change it to improve performance.

Testing the AMD Piledriver architecture, or another AMD architecture, against the Intel architectures could also be of interest. The state of the testing machine might have the effect of making the results look worse than how the architecture could perform. A similar case can be made for the testing environment. While processor boost was turned off all testing machines, the Piledriver and Kaby Lake machines run Windows 8.1 and Windows 10 respectively, with all that entails, meaning that other processes could interfere with the results. In both cases, those machines were used because they were the only ones available with those architectures. Consistent testing environments would definitely lead to better results.

Investigating less than ideal cases could also show how the algorithms perform against each other in a more general case. Every case tested in this thesis has aligned perfectly with the algorithm. Attempting to test on different, non-ideal, message sizes could give a more general perspective than the highly optimized tests performed in this thesis.

While Deoxys with AES is a case of the whole being greater (or in this case faster) than the sum of its parts, it would still be interesting to investigate why this is the case, and how it could be applied in other cases.

CHAPTER 10

Conclusion

I have found that in the cases tested, COLM is a bit slower than Deoxys with AES, and that $COLM_0$ is a bit faster than $COLM_{127}$. The latter is no surprise at all, as the difference between the two algorithms is the presence of intermediate tags in $COLM_{127}$ (and indeed in $COLM_\tau$). I have also found that the loss in performance going from $COLM_0$ to $COLM_{127}$ is smaller for larger messages, and I believe the loss would be even smaller, for small and large messages alike, if the modulo operation that controls the intermediate tag generation could be removed. As for Deoxys vs $COLM_0$, the former is more effective for longer messages, and the latter is more effective for shorter messages, with the breaking point being between 1024 and 2048 bytes, depending on the number of blocks processed per iteration of the encryption loop. The tests on the Piledriver architecture show that $COLM_0$ is faster with eight blocks per iterations for the shorter messages, and in all other cases, it is faster with four blocks per iteration. Of further note, the Piledriver tests also show that an old, ill-maintained computer and a sub-optimal testing environment leads to large variance in results, and probably also worse results than what the Piledriver architecture would actually be capable of performing.

Going back to COLM vs Deoxys, we see that the performance numbers are better for longer messages with Deoxys and better for shorter messages with COLM. While there is a gap between theory and practice, the field doubling in COLM being substantially more efficient than estimated, this can be explained by COLM performing worse than expected when accounting for field doubling, while Deoxys performs better than the individual pieces should allow.

The implementation of Deoxys with Skinny is functionally successful, but requires work in terms of efficiency, and takes up a lot of internal memory. Testing Deoxys with a dedicated tweakable block cipher, such as Skinny, shows how it performs against the standard AES cipher, and shows that it is possible to insert a lightweight cipher in an AEAD framework. I believe that performance can be gained by optimizing for memory in this implementation. Furthermore, Skinny was tested in a high-performance setting, while the cipher is designed for lightweight applications.

Of the ciphers tested, both ciphers are found to outperform AES-CCM in terms of performance, and while neither have the same performance numbers as AES-GCM, both of them are nonce-misuse resistant, allowing for use even in settings where poor randomness, or complete lack thereof, are used in cipher initialization.

Bibliography

[1] Ross Anderson, Eli Biham, and Lars Knudsen. "Serpent: A Proposal for the Advanced Encryption Standard, 1998". In: *AES submission* (1998).

[2] Elena Andreeva et al. "COLM v1 (2016)". In: *Submission to the CAESAR competition* (2016).

[3] ARM. *ARM Architecture Reference Manual ARMv8 for ARMv8-A architecture profile*. `https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile`. PDF can be downloaded from link. Accessed: 2017-07-03.

[4] ARM. *ARM NEON Intrinsics Reference*. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0073a/IHI0073A_arm_neon_intrinsics_ref.pdf`. PDF can be downloaded from link. Accessed: 2017-08-02.

[5] Christof Beierle et al. "The SKINNY family of block ciphers and its low-latency variant MANTIS". In: *Annual Cryptology Conference*. Springer. 2016, pages 123–153.

[6] Mihir Bellare and Chanathip Namprempre. "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2000, pages 531–545.

[7] Mihir Bellare and Chanathip Namprempre. "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm". In: *Journal of Cryptology* 21.4 (2008), pages 469–491.

[8] Mihir Bellare, Phillip Rogaway, and David Wagner. "A conventional authenticated-encryption mode". In: *manuscript, April* (2003).

[9] Dan J. Bernstein. *CAESAR use cases*. `http://kortlink.dk/qwvp`. Accessed: 2017-07-02. Link shortened for convenience.

[10] Ritam Bhaumik and Mridul Nandi. "OleF: An Inverse-Free Online Cipher." In: *IACR Cryptology ePrint Archive* 2016 (2016), page 1090.

[11] Eli Biham. "A fast new DES implementation in software". In: *FSE*. Volume 1267. Springer. 1997, pages 260–272.

[12] Andrey Bogdanov, Martin M Lauridsen, and Elmar Tischhauser. "AES-Based Authenticated Encryption Modes in Parallel High-Performance Software." In: *IACR Cryptology ePrint Archive* 2014 (2014), page 186.

[13] Philip Bulman. *NIST Announces Encryption Standard Finalists*. http://csrc.nist.gov/archive/aes/round2/AESpressrelease-990809.pdf. Accessed: 2017-08-02.

[14] *CAESAR call for submissions, final (2014.01.27)*. https://competitions.cr.yp.to/caesar-call.html. Accessed: 2017-07-02.

[15] CPU-World. *AMD A6-4455M specifications*. http://kortlink.dk/r23d. Link shortened because the Latex compiler did not like the original link. Accessed: 2017-08-07.

[16] Ian Cutress. *Intel's 'Tick-Tock' Seemingly Dead, becomes 'Process-Architecture-Optimization'"*. http://www.anandtech.com/show/10183/intels-tick-tock-seemingly-dead-becomes-process-architecture-optimization. Accessed: 2017-08-02.

[17] Bernd Dammann. *The Memory Hierarchy*. Handout from the course 02614 High-Performance Computing in January 2016.

[18] Morris Dworkin. *Recommendation for block cipher modes of operation. methods and techniques*. Technical report. NATIONAL INST OF STANDARDS and TECHNOLOGY GAITHERSBURG MD COMPUTER SECURITY DIV, 2001.

[19] Morris J Dworkin. "Recommendation for block cipher modes of operation: Galois/-Counter Mode (GCM) and GMAC". In: *Special Publication (NIST SP)-800-38D* (2007).

[20] Craig Gentry. "Computing arbitrary functions of encrypted data". In: *Communications of the ACM* 53.3 (2010), pages 97–105.

[21] P. Gutmann. *Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. https://tools.ietf.org/html/rfc7366. Accessed: 2017-07-16.

[22] R. Housley. *Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP)*. https://tools.ietf.org/html/rfc4309. Accessed: 2017-08-07.

[23] R. Housley. *Using AES-CCM and AES-GCM in the CMS*. https://tools.ietf.org/html/rfc5084. Accessed: 2017-08-17.

[24] Intel. *6th Generation Intel Core Processor Family Datasheet, Vol. 1*. https://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-1.html. Accessed: 2017-08-02.

[25] Intel. *Datasheet, Vol. 1: 7th Gen Intel Processor Family for S Platforms*. https://www.intel.com/content/www/us/en/processors/core/7th-gen-core-family-desktop-s-processor-lines-datasheet-vol-1.html. Accessed: 2017-08-02.

[26] Intel. *Intel Advanced Encryption Standard (AES) New Instructions Set*. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf. Pages 31-32. Accessed: 2017-07-07.

[27] Intel. *Intel Intrinsics Guide*. http://kortlink.dk/qwy9. Accessed: 2017-07-03. Link shortened to avoid Latex issues.

[28] Intel. *Intel Intrinsics Guide*. https://software.intel.com/sites/landingpage/IntrinsicsGuide. Accessed: 2017-07-25.

[29] *Information technology – Security techniques – Authenticated encryption*. Standard. International Organization for Standardization, February 2009.

[30] Jérémy Jean. *Skinny 8-bit Sbox*. https://www.iacr.org/authors/tikz/. Can be found under "Block ciphers" -> "Superposition TWEAKEY of order 3 (TK3)". Accessed: 2017-07-08.

[31] Jérémy Jean. *Skinny 8-bit Sbox*. https://www.iacr.org/authors/tikz/. Can be found under "Block ciphers" -> "Skinny 8-bit Sbox". Accessed: 2017-07-04.

[32] Jérémy Jean. *Skinny Round Function*. https://www.iacr.org/authors/tikz/. Can be found under "Block ciphers" -> "Skinny round function". Accessed: 2017-07-04.

[33] Jérémy Jean. *Skinny TWEAKEY Schedule*. https://www.iacr.org/authors/tikz/. Can be found under "Block ciphers" -> "Skinny TWEAKEY Schedule". Accessed: 2017-07-04.

[34] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. "Tweaks and keys for block ciphers: the TWEAKEY framework". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2014, pages 274–288.

[35] Jérémy Jean et al. "Deoxys v1. 41". In: (2016).

[36] Nigel Jones. *Efficient C Tip 13*. http://embeddedgurus.com/stack-overflow/2011/02/efficient-c-tip-13-use-the-modulus-operator-with-caution/. Title shortened Latex issues. Accessed: 2017-07-30.

[37] Antoine Joux. "Authentication failures in NIST version of GCM". In: *NIST Comment* (2006), page 3.

[38] Ilmari Karonen (possible pseudonym). *Why shouldn't I use ECB encryption?* https://crypto.stackexchange.com/questions/20941/why-shouldnt-i-use-ecb-encryption. Accessed: 2017-07-02.

[39] Lars Knudsen. *Cryptology I*. Course notes for 01410 Cryptology 1, spring 2014. 2014, pages 28–29.

[40] Stefan Kölbl and Skinny authors. *Optimized implementation for Skinny128*. https://github.com/kste/skinny_avx/tree/master/skinny128/avx2.

[41] John Leyden. *The perfect CRIME? New HTTPS web hijack attack explained*. http://www.theregister.co.uk/2012/09/14/crime_tls_attack/. Accessed: 2017-07-28.

[42] *List of AMD accelerated processing unit microprocessors*. https://en.wikipedia.org/wiki/List_of_AMD_accelerated_processing_unit_microprocessors. Section Comal: "Trinity" (2012). Accessed: 2017-08-02.

[43] David McGrew and John Viega. “The Galois/counter mode of operation (GCM)”. In: *Submission to NIST Modes of Operation Process* 20 (2004).

[44] J. Postel. *RFC 793 - TRANSMISSION CONTROL PROTOCOL*. https://tools.ietf.org/html/rfc793. Section 3.1. Accessed: 2017-08-01.

[45] RokerHRO (pseudonym). *Authenticated Encryption EaM (image)*. https://en.wikipedia.org/wiki/File:Authenticated_Encryption_EaM.png. Accessed: 2017-07-02.

[46] RokerHRO (pseudonym). *Authenticated Encryption EtM (image)*. https://en.wikipedia.org/wiki/File:Authenticated_Encryption_EtM.png. Accessed: 2017-07-02.

[47] RokerHRO (pseudonym). *Authenticated Encryption MtE (image)*. https://en.wikipedia.org/wiki/File:Authenticated_Encryption_MtE.png. Accessed: 2017-07-02.

[48] Kristian Riisgaard. *Is Dahlmeier putting together the greatest single season of all time?* https://www.reddit.com/r/biathlon/comments/5xti89/is_dahlmeier_putting_together_the_greatest_single/del4zq6/. Posted under pseudonym ”A_Guy_Named_Kris”. Accessed: 2017-07-11.

[49] G.S. Sheaffer. *Division algorithm for floating point or integer numbers*. US Patent 5,784,307. July 1998. URL: https://www.google.com/patents/US5784307.

[50] Nigel Smart. *Cryptography made simple*. Springer, 2015, page 217.

[51] Peter Soderquist and Miriam Leeser. “Floating-Point Division and Square Root: Choosing the Right Implementation”. In: (January 1997).

[52] John Stokes. *SIMD architectures*. https://arstechnica.com/features/2000/03/simd/. Accessed: 2017-07-03.

[53] Elmar Tischhauser. Personal communication.

[54] Serge Vaudenay. “Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS”. In: *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*. Volume 2332. Lecture Notes in Computer Science. Springer, 2002, pages 534–546. DOI: 10.1007/3-540-46035-7_35. URL: http://www.iacr.org/cryptodb/archive/2002/EUROCRYPT/2850/2850.pdf.

[55] Wikichip. *Kaby Lake*. https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake. Section: Memory hierarchy, Accessed: 2017-08-07.

[56] Wikichip. *Kaby Lake*. https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake. Section: Memory hierarchy, Accessed: 2017-08-07.

[57] Wikipedia. *ARM Cortex-A57*. https://en.wikipedia.org/wiki/ARM_Cortex-A57. Accessed: 2017-08-07.

APPENDIX A

Performance and variance tables

A.1 Performance tables

Processor \length	128	256	512	1024	2048	4096	8192	16384
Skylake	4.9	3.51	2.87	2.41	2.23	2.15	2.1	2.08
Kaby Lake	4.98	3.62	2.94	2.53	2.4	2.29	2.24	2.15
Piledriver	8.58	6.53	5.35	4.74	4.55	4.51	4.56	4.79

Table A.1: Performance of $COLM_0$ with four blocks per iteration, all numbers are in cycles per byte.

Processor \length	128	256	512	1024	2048	4096	8192	16384
Skylake	5.18	3.85	3.1	2.71	2.51	2.41	2.39	2.33
Kaby Lake	5.5	4.15	3.46	3.05	2.85	2.74	2.68	2.67
Piledriver	13.61	9.81	5.7	4.86	4.44	4.36	4.37	4.36

Table A.2: Performance of $COLM_0$ with eight blocks per iteration, all numbers are in cycles per byte.

Processor \length	128	256	512	1024	2048	4096	8192	16384
Skylake	6.6	4.87	3.84	3.3	3.5	2.96	2.86	2.85
Kaby Lake	6.23	4.77	3.87	3.4	3.17	3.11	3.05	3.04
Piledriver	9.73	7.14	5.83	5.33	5.06	4.99	5.01	4.94

Table A.3: Performance of $COLM_{127}$ with four blocks per iteration, all numbers are in cycles per byte.

Processor \length	128	256	512	1024	2048	4096	8192	16384
Skylake	12.39	7.8	5.36	4.03	3.42	3.05	2.86	2.79
Kaby Lake	12.59	7.93	5.45	4.09	3.43	3.11	2.93	2.86
Piledriver	21.74	13.76	9.06	7.01	5.85	5.32	5.04	4.9

Table A.4: Performance of $COLM_{127}$ with eight blocks per iteration, all numbers are in cycles per byte.

Processor \length	128	256	512	1024	2048	4096	8192	16384
Skylake	7.18	4.56	3.26	2.62	2.28	2.11	2.03	1.99
Kaby Lake	5.91	3.99	3.02	2.54	2.3	2.18	2.12	2.09
Piledriver	9.34	6.28	4.79	4.15	4.09	3.86	3.94	3.87

Table A.5: Performance of Deoxys with four blocks per iteration, all numbers are in cycles per byte.

Processor \length	128	256	512	1024	2048	4096	8192	16384
Skylake	8.05	4.96	3.35	2.54	2.16	1.95	1.86	1.81
Kaby Lake	7.95	5.24	3.85	3.09	2.72	2.53	2.46	2.43
Piledriver	12.86	8.94	7.2	6.13	5.69	5.57	5.55	5.49

Table A.6: Performance of Deoxys with eight blocks per iteration, all numbers are in cycles per byte.

A.2 Variance tables

	128	256	512	1024	2048	4096	8192	16384
Skylake	$2.3 \cdot 10^{-4}$	$2.5 \cdot 10^{-4}$	$8.6 \cdot 10^{-5}$	$2.6 \cdot 10^{-6}$	$1.7 \cdot 10^{-5}$	$2.2 \cdot 10^{-6}$	$4.1 \cdot 10^{-6}$	$4.8 \cdot 10^{-7}$
Kaby Lake	$5.8 \cdot 10^{-4}$	$7.1 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$	$4.7 \cdot 10^{-5}$	$2.5 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$4.1 \cdot 10^{-5}$	$2.7 \cdot 10^{-5}$
Piledriver	1.75	0.75	0.024	0.0103	0.011	0.021	0.0203	0.0066

Table A.7: Variance of test results of $COLM_0$ with 8 blocks per iteration.

	128	256	512	1024	2048	4096	8192	16384
Skylake	$2.8 \cdot 10^{-5}$	$5.5 \cdot 10^{-5}$	$3.7 \cdot 10^{-6}$	$2.8 \cdot 10^{-5}$	$7.7 \cdot 10^{-4}$	$9.4 \cdot 10^{-5}$	$1.1 \cdot 10^{-4}$	$1.3 \cdot 10^{-4}$
Kaby Lake	$4.8 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$	$5.9 \cdot 10^{-5}$	$8.5 \cdot 10^{-5}$	$6.7 \cdot 10^{-5}$	$3.7 \cdot 10^{-5}$	$8.7 \cdot 10^{-6}$	$4.9 \cdot 10^{-6}$
Piledriver	0.124	0.024	0.022	0.0378	0.0157	0.0203	0.015	0.0098

Table A.8: Variance of test results of $COLM_{127}$ with 4 blocks per iteration.

	128	256	512	1024	2048	4096	8192	16384
Skylake	$2.9 \cdot 10^{-5}$	$6 \cdot 10^{-5}$	$2.5 \cdot 10^{-6}$	$8 \cdot 10^{-7}$	$3.6 \cdot 10^{-6}$	$5.1 \cdot 10^{-6}$	$2.7 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$
Kaby Lake	$7.8 \cdot 10^{-4}$	$2.9 \cdot 10^{-4}$	$9.1 \cdot 10^{-5}$	$3.3 \cdot 10^{-5}$	$1 \cdot 10^{-4}$	$6.1 \cdot 10^{-6}$	$1.2 \cdot 10^{-5}$	$2.5 \cdot 10^{-6}$
Piledriver	1.649	0.061	0.056	0.026	0.059	0.0112	0.0284	0.0142

Table A.9: Variance of test results of $COLM_{127}$ with 8 blocks per iteration.

	128	256	512	1024	2048	4096	8192	16384
Skylake	$1.4 \cdot 10^{-5}$	$3.4 \cdot 10^{-6}$	$3.6 \cdot 10^{-6}$	$6.3 \cdot 10^{-6}$	$3.5 \cdot 10^{-8}$	$1.9 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$
Kaby Lake	$1.1 \cdot 10^{-3}$	$2.3 \cdot 10^{-4}$	$9.6 \cdot 10^{-5}$	$4.6 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$6.4 \cdot 10^{-6}$	$2.3 \cdot 10^{-5}$	$2.3 \cdot 10^{-5}$
Piledriver	0.22	0.083	0.029	0.02	0.019	0.021	0.015	0.012

Table A.10: Variance of test results of Deoxys with 4 blocks per iteration.

	128	256	512	1024	2048	4096	8192	16384
Skylake	$6.5 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$1.6 \cdot 10^{-5}$	$3.1 \cdot 10^{-5}$	$8 \cdot 10^{-6}$	$4.7 \cdot 10^{-8}$	$1.4 \cdot 10^{-8}$	$3.1 \cdot 10^{-7}$
Kaby Lake	$1.5 \cdot 10^{-3}$	$4.2 \cdot 10^{-4}$	$1.2 \cdot 10^{-4}$	$7.3 \cdot 10^{-5}$	$1.9 \cdot 10^{-5}$	$2 \cdot 10^{-4}$	$1.8 \cdot 10^{-5}$	$3.8 \cdot 10^{-6}$
Piledriver	0.53	0.19	0.35	0.026	0.026	0.041	0.016	0.017

Table A.11: Variance of test results of Deoxys with 8 blocks per iteration.

	128	256	512	1024	2048	4096	8192	16384
$COLM_0$ (4)	$4.6 \cdot 10^{-3}$	$3.2 \cdot 10^{-4}$	$1.8 \cdot 10^{-4}$	$3.3 \cdot 10^{-5}$	$1.4 \cdot 10^{-5}$	$9.4 \cdot 10^{-6}$	$4.4 \cdot 10^{-6}$	$2.8 \cdot 10^{-6}$
$COLM_0$ (8)	0.011	$4.7 \cdot 10^{-3}$	$2.9 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	$6.2 \cdot 10^{-4}$	$3.6 \cdot 10^{-3}$	$3.2 \cdot 10^{-4}$	$1.5 \cdot 10^{-4}$
$COLM_{127}$ (4)	$2.6 \cdot 10^{-4}$	$9.7 \cdot 10^{-5}$	$5.5 \cdot 10^{-5}$	$1.9 \cdot 10^{-5}$	$1.9 \cdot 10^{-5}$	$6.6 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$	$3 \cdot 10^{-6}$
$COLM_{127}$ (8)	$7.1 \cdot 10^{-4}$	$2.4 \cdot 10^{-4}$	$8.7 \cdot 10^{-5}$	$4 \cdot 10^{-5}$	$2.1 \cdot 10^{-5}$	$5.7 \cdot 10^{-6}$	$5.4 \cdot 10^{-6}$	$2.4 \cdot 10^{-6}$
Deoxys (4)	$2 \cdot 10^{-3}$	$2.7 \cdot 10^{-4}$	$7.7 \cdot 10^{-5}$	$2.9 \cdot 10^{-5}$	$1.7 \cdot 10^{-5}$	$1.1 \cdot 10^{-5}$	$4.2 \cdot 10^{-6}$	$2.4 \cdot 10^{-6}$
Deoxys (8)	$1.5 \cdot 10^{-3}$	$4.2 \cdot 10^{-4}$	$1.3 \cdot 10^{-4}$	$1.4 \cdot 10^{-4}$	$4 \cdot 10^{-5}$	$1.4 \cdot 10^{-5}$	$6.8 \cdot 10^{-6}$	$4.7 \cdot 10^{-6}$

Table A.12: Variance table for tests of ciphers on Kaby Lake using GCC 4.9.2 as compiler.

Processor length	1024	2048	4096	8192	16384	32768	65536
Skylake	$3.9 \cdot 10^{-4}$	$9.2 \cdot 10^{-5}$	$7.7 \cdot 10^{-3}$	$2.5 \cdot 10^{-4}$	$5.9 \cdot 10^{-5}$	$1.8 \cdot 10^{-3}$	$7 \cdot 10^{-6}$
Kaby Lake	0.039	$7 \cdot 10^{-4}$	$4 \cdot 10^{-4}$	$2 \cdot 10^{-3}$	$8.1 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	$3.9 \cdot 10^{-3}$

Table A.13: Variance of test results of Deoxys with Skinny.

	128	256	512	1024	2048	4096	8192	16384
COLM	$2.9 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	$2.3 \cdot 10^{-4}$	$1 \cdot 10^{-4}$	$9.2 \cdot 10^{-6}$	$3.7 \cdot 10^{-5}$	$3.7 \cdot 10^{-6}$	$1.7 \cdot 10^{-4}$
Deoxys	$3.9 \cdot 10^{-3}$	$9.4 \cdot 10^{-4}$	$4.6 \cdot 10^{-4}$	$6.3 \cdot 10^{-4}$	$6.4 \cdot 10^{-4}$	$3.9 \cdot 10^{-4}$	$7.5 \cdot 10^{-3}$	0.019

Table A.14: Variance table of tests on ARM, all numbers are in cycles per byte.

APPENDIX B

Intrinsics used in implementation

All values are taken from [28] .

Name	Description	Input	Output
___m128i	128 bit vector		
_mm_aesenc_si128	One round of AES encryption on 128 bits	two ___m128i: Plaintext, key	___m128i
_mm_aesenclast_si128	Final round of AES encryption on 128 bits	two ___m128i: Plaintext, key	___m128i
_mm_aesdec_si128	One round of AES decryption on 128 bits	two ___m128i: Ciphertext, key	___m128i
_mm_aesdeclast_si128	Final round of AES decryption on 128 bits	two ___m128i: Ciphertext, key	___m128i
_mm_aesimc_si128	Inverse mix columns on 128 bits	one ___m128i	___m128i
_mm_aeskeygenassist_si128	Function used in making AES key schedule	one ___m128i and an integer	___m128i
_mm_xor_si128	xor of two 128 bits variables	two ___m128i	___m128i
_mm_and_si128	and of two 128 bits variables	two ___m128i	___m128i
_mm_add_epi8	Addition of 16 bytes in a 128 bit vector	two ___m128i	___m128i
_mm_loadu_si128	Loads into 128 bit vector	Pointer to source	___m128i
_mm_storeu_si128	Stores from 128 bit vector to standard data type	Pointer to destination, source	
_mm_setzero_si128	Sets 128 bit vector to zero	none	___m128i
_mm_set_epi64x	Sets 128 bit vector from standard data	two 64 bit variables	___m128i
_mm_set1_epi8	Sets bytes in 128 bit vector	single byte	___m128i
_mm_cmpgt_epi32	Compares four 32 bit elements in 128 bit vectors	two ___m128i	___m128i
_mm_shuffle_epi32	Reorganizes four 32 bit elements, according to mask	___m128i and 8 bit mask	___m128i
_mm_shuffle_epi8	Reorganizes vector according to mask	two ___m128i: Source, mask	___m128i
_mm_slli_epi64	Shifts two 64 bit elements left by n elements	integer n and ___m128i	___m128i
_mm_srli_epi64	Shifts two 64 bit elements right by n elements	integer n and ___m128i	___m128i
_mm_slli_epi32	Shifts four 32 bit elements left by n elements	integer n and ___m128i	___m128i
_mm_srli_epi32	Shifts four 32 bit elements right by n elements	integer n and ___m128i	___m128i
_mm_slli_si128	Shifts bytes in vector by n elements	integer n and ___m128i	___m128i

Table B.1: List of Intel intrinsic operations used in my implementations with descriptions.

Name	Latency	Throughput
_mm_aesenc_si128	(4/7/7/8)	1
_mm_aesenclast_si128	(4/7/7/8)	1
_mm_aesdec_si128	(4/7/7/8)	1
_mm_aesdeclast_si128	(4/7/7/8)	1
_mm_aesimc_si128	(8/14/14/14)	2
_mm_aeskeygenassist_si128	(12/10/10/10)	(12/8/8/8)
_mm_xor_si128	1	0.33
_mm_and_si128	1	0.33
_mm_add_epi8	1	(0.33/0.5/0.5/0.5)
_mm_loadu_si128	1	(0.25/0.33/0.33/0.5)
_mm_storeu_si128	1	(0.25/0.33/0.33/0.5)
_mm_setzero_si128	1	0.33
_mm_set_epi64x	N/A	N/A
_mm_set1_epi8	N/A	N/A
_mm_cmpgt_epi32	1	0.5
_mm_shuffle_epi32	1	(1/1/1/0.5)
_mm_shuffle_epi8	1	(1/1/1/0.5)
_mm_slli_epi64	1	1
_mm_srli_epi64	1	1
_mm_slli_epi32	1	1
_mm_srli_epi32	1	1
_mm_slli_si128	1	(1/1/1/0.5)

Table B.2: List of Intel intrinsic operations used in my implementations with performance figures.

Functions names can be read at [4] .

Name	Functionality	Input
uint8x16_t	vector of 16 eight bit elements	
vld1q_u8	Loads data from array into vector	Pointer to source
vst1q_u8	Stores data from vector into array	uint8x16_t
vaeseq_u8	One round of AES decryption, except MixColumns, on 128 bits	Two uint8x16_t: Plaintext and key
vaesdq_u8	One round of AES decryption, except MixColumns, on 128 bits	Two uint8x16_t: Ciphertext and key
vaesmcq_u8	MixColumns on 128 bit vector	uint8x16_t
vaesimcq_u8	inverse MixColumns on 128 bit vector	uint8x16_t
vshlq_n_u8	Left shift of elements in 128 bit vector	uint8x16_t vector and an integer
vqtbl1q_u8	Shuffle elements of 128 bit vector according to mask	two uint8x16_t: Source and mask
vandq_u8	Bitwise and of two 128 bit vectors	two: uint8x16_t

Table B.3: List of ARM intrinsic operations used in my implementations.

APPENDIX C

Theoretical performance of auxiliary functions

C.1 COLM key schedule

The COLM key schedule has ten calls to the `_mm_aeskeygenassist_si128` function, all of which are performed sequentially. There are also calls to a subfunction called `key_exp_assist`, which can be seen in listing C.1

```
__m128i key_exp_assist(__m128i t1, __m128i t2)
{
  __m128i t3 = _mm_slli_si128(t1,0x04);
  t1 = _mm_xor_si128(t1,t3);
  t3 = _mm_slli_si128(t1,0x04);
  t1 = _mm_xor_si128(t1,t3);
  t3 = _mm_slli_si128(t1,0x04);
  t1 = _mm_xor_si128(t1,t3);
  t2 = _mm_shuffle_epi32(t2,0xFF);
  return _mm_xor_si128(t1,t2);
}
```

Listing C.1: `key_exp_assist`.

Eight operations, all of which are performed sequentially. This takes eight cycles. Ten times `key_exp_assist` and ten times `_mm_aeskeygenassist_si128` runs to 200 cycles on Skylake, and 180 cycles on the other Intel processors.

Processing of the extra keys needed in decryption requires nine calls to `_mm_aesimc_si128` which run in parallel, meaning that it will take 18 cycles plus 6 on Skylake and plus 12 on the other processors.

C.2 COLM Field doubling

The field code can be seen at listing 5.3. Field doubling has nine operations. The logical operations must run sequentially, and all other operations can't be parallelized, meaning that it takes nine cycles.

C.3 COLM ρ function

Two sequential xors and a field doubling, leading to 11 cycles in total.

C.4 COLM single block encryption

The single block encryption function in COLM is rather simple, being ten sequential single encryptions and an xor. This takes 41 cycles on Skylake, 71 cycles on Broadwell and Haswell and 81 cycles on Ivy Bridge. An additional cycle is added if a load is required.

C.5 Parallel COLM encryption

If one round of encryption with a full pipeline is executed, every encryption operation only needs to be counted to take the number of cycles listed in the throughput times the numbers of blocks. Consider figure 5.3, with a depiction of the encryption pipeline. The next operation can be started after one cycle, and this holds for every operation. This means that the running time is number of encryptions plus latency minus throughput.

C.6 Deoxys key schedule

The Deoxys key schedule contains an xor followed by a loop of `_mm_shuffle_epi8`, the Skinny LFSR and an xor. This loop runs for 14 iterations. The Skinny LFSR can be seen in listing C.2

```
1 __m128i LFSR2(__m128i k)
2 {
3   __m128i mask_high = _mm_set1_epi8(0xfe);
4   __m128i mask_low  = _mm_set1_epi8(0x01);
5   __m128i tmp = _mm_xor_si128( _mm_and_si128( mask_high  , _mm_slli_epi32( k
        , 1 ) ) ,
6             _mm_and_si128( mask_low, _mm_srli_epi32( k, 7 ) ) );
7   return _mm_xor_si128(_mm_and_si128( mask_low, _mm_srli_epi32( k, 5 ) ),tmp
        );
8 }
```

Listing C.2: Skinny LFSR.

Line 3 and 4 of the LFSR holds multiple commands, and dissecting those leads to

```
1   sl1 = _mm_slli_epi32( k, 1 )
2   sl7 = _mm_srli_epi32( k, 7 )
3   sl5 = _mm_srli_epi32( k, 5 )
4   a1 = _mm_and_si128( mask_high  , sl1 )
```

```
5  a2 = _mm_and_si128( mask_low, sl7 )
6  a3 = _mm_and_si128( mask_low, sl5)
7  __m128i tmp = _mm_xor_si128( a1 ,a2 );
8  return _mm_xor_si128(a3,tmp)
```

Listing C.3: Dissection of line 3 and 4 in the LFSR.

The shift operations can't be parallelized, and the two final xors must run sequentially. The three ands can be run at the same time, taking only one cycle. This leads to a total of six cycles for the LFSR.

Knowing how many cycles the LFSR takes, I can now conclude that the Deoxys key schedule takes 113 cycles in total, no matter which processor the key schedule is executed on.

C.7 Deoxys single block encryption

Using the arrays in the Deoxys implementation, it is possible to group xor operations together to parallelize these operations. The encryption operations must be performed sequentially. There are two sequential xors and seven shuffles in the initial step, taking nine cycles in total. Grouping the xor operations for the tweaks and keys correctly reduces the process to taking five cycles. The remaining 14 encryption operations must be performed sequentially. This runs to a total of 70 cycles on Skylake, 112 cycles on Broadwell and Haswell and 126 cycles on Ivy Bridge. An additional cycle is added if loading is necessary.

C.8 Parallel Deoxys encryption

The same argument applies here, as the argument in parallel COLM encryption. There are xor's in between, but the benchmark of the Deoxys encryption loops, described in section 8.2.3, show that the closest model of the pipelines is the one depicted in figure 7.3. The easiest calculation method here is to go for a round based approach, where every round takes the number of cycles stated in the latency plus another round for the xor. It's two times latency plus one round for the xor for Deoxys with eight blocks per iteration on Skylake. There are 14 rounds, and the last one is without an xor at and latency minus throughput is added at the end.

C.9 Deoxys arrays

The tables required in the Deoxys imeplentation have 22 shuffles and, depending on the number of blocks per iteration, three or seven sequential adds eight times. Every operation takes one cycle, so the number of cycles is 78 if eight blocks per iteration and 46 if four blocks per iteration.

APPENDIX D

Results of intermediate or auxiliary tests

D.1 Subprocesses

More in depth results of the tests of the different subfunctions can be found here

```
0.128722 cpb
0.128916 cpb
0.128719 cpb
0.128818 cpb
0.12861 cpb
0.128663 cpb
0.128542 cpb
0.128788 cpb
0.128547 cpb
0.128762 cpb
```

Listing D.1: Performance of field doubling.

```
2.14801 cpb
2.14773 cpb
2.14752 cpb
2.14959 cpb
2.14757 cpb
2.14762 cpb
2.14759 cpb
2.14759 cpb
2.14814 cpb
2.14805 cpb
```

Listing D.2: Performance of $COLM_0$ encryption loop - 4 blocks per iteration.

```
2.10503 cpb
2.10477 cpb
2.10472 cpb
2.10485 cpb
2.10498 cpb
2.10491 cpb
2.10486 cpb
2.10491 cpb
```

```
2.1047 cpb
2.10487 cpb
```

Listing D.3: Performance of $COLM_0$ encryption loop - 8 blocks per iteration.

```
1.22246 cpb
1.22258 cpb
1.22247 cpb
1.22242 cpb
1.22259 cpb
1.22244 cpb
1.22243 cpb
1.22245 cpb
1.22247 cpb
1.22244 cpb
```

Listing D.4: Performance of Deoxys encryption loop - 4 blocks per iteration.

```
1.12216 cpb
1.12227 cpb
1.12218 cpb
1.12228 cpb
1.12216 cpb
1.12226 cpb
1.12231 cpb
1.12228 cpb
1.12229 cpb
1.1222 cpb
```

Listing D.5: Performance of Deoxys encryption loop - 8 blocks per iteration.

```
4.009432 cycles per byte
4.010084 cycles per byte
4.014967 cycles per byte
4.061212 cycles per byte
4.013759 cycles per byte
4.040649 cycles per byte
4.170899 cycles per byte
4.046730 cycles per byte
4.061649 cycles per byte
4.044251 cycles per byte
```

Listing D.6: Performance of Deoxys encryption with Skinny.

D.2 Serial implementation of $COLM_0$

Using the same testing method as I have used for testing every other algorithm, I find an average performance of the serial implementation of $COLM_0$ to be 3.37 cycles per byte, with a variance of $3.97 \cdot 10^{-6}$. To find more detailed results, see the file list in appendix F.

APPENDIX E
Test of ARM memory use

The memory usage of the Deoxys cipher was tested at an attempt to find out why there is a decrease in performance for Deoxys with the longest messages. Two reasons for this even were considered, thermal throttling or a memory leak. The memory use during a test of Deoxys with a 16 KB message was recorded using the `top` command to record every two seconds, and the results showed that no change in memory used throughout the process, aside from within the first two seconds. The first and last ten records are shown below:

```
7395  6  12% R     1    844K      4K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
// --------------- SPLIT until last 10 -------------------------
 7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
7395  6  12% R     1    908K    260K  fg u0_a198  ./timing-arm
```

Listing E.1: ARM memory usage.

In terms of performance, it shows that the algorithm becomes slower over time.

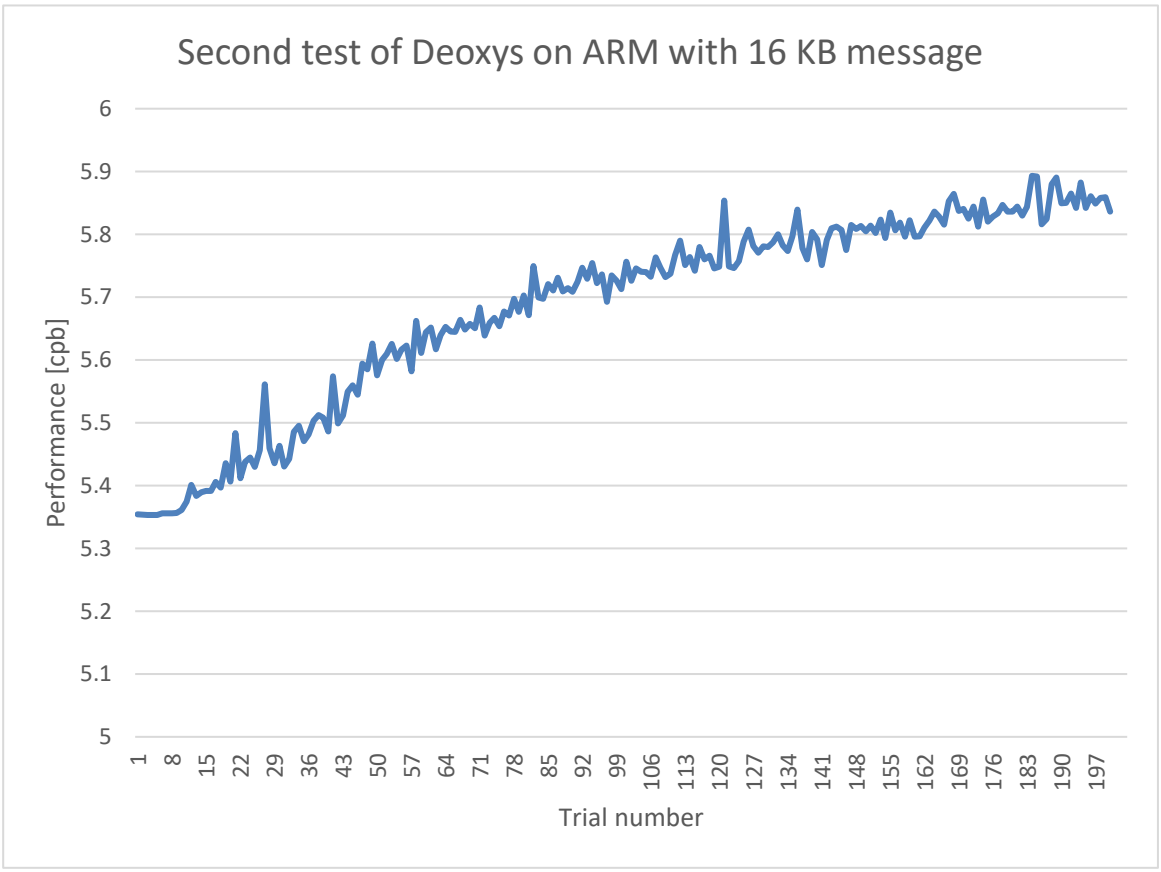


Figure E.1: Performance graph for Deoxys on ARM with a message size of 16384 KB.

APPENDIX F

List of zipped folders appended to this thesis

DeoxysSkinny.rar Implementation of Deoxys with Skinny. Contains the files for an ideal case only version, and a folder called ”final” for a version that takes ciphertexts of length divisible by 16.

Results.rar Test results. Folders are named after the algorithms tested, with some exceptions. ”Skinny” is Deoxys with Skinny on Kaby Lake (GCC 5.2.0) and Skylake. ”New compiler” are the tests of $COLM_0$ and Deoxys with AES on Kaby Lake with GCC 5.2.0. ”ARM_res” contains the results of the tests on ARM. For the remaining results, the processor is named in the folders and in the case of the Kaby Lake tests of $COLM_{127}$, two subfolders, each named after the GCC version used.

COLM0_SER.rar Contains the test implementation of $COLM_0$ with sequential processing.

COLM_ARM.rar Implementation of $COLM_0$ on ARM

Deoxys_ARM.rar Implementation of Deoxys with AES on ARM

COLM0_8.rar Implementation of $COLM_0$ with Intel instructions, eight blocks per iteration.

COLM0_4.rar Implementation of $COLM_0$ with Intel instructions, four blocks per iteration.

COLM127_4.rar Implementation of $COLM_{127}$ with Intel instructions, four blocks per iteration.

COLM127_8.rar Implementation of $COLM_{127}$ with Intel instructions, eight blocks per iteration.

Deox4.rar Implementation of Deoxys with AES with Intel instructions, four blocks per iteration.

Deox8.rar Implementation of Deoxys with AES with Intel instructions, eight blocks per iteration.