COMP 5074 Cryptography and Data Protection (2022)

Lecture 7: Authenticated encryption with associated data

Dr. Yee Wei Law ⟨yeewei.law@unisa.edu.au⟩

Friday 4th November, 2022

⚠ Template and resources

This lecture handout uses the *same* template as the Research Paper *except* the font size has been increased to 14 pt for on-Zoom readability.

The LaTeX code and other resources used in this document are available for you to use, but please make sure you have enough original content.

Contents

List of acronyms

AEAD	Authenticated encryption with associated data
DES	Data Encryption Standard
FIPS	Federal Information Processing Standard
IETF	Internet Engineering Task Force
IND	Indistinguishability
IV	Initialisation vector
LSB	Least significant bit/byte
MAC	Message authentication code
MSB	Most significant bit/byte
NIST	National Institute of Standards and Technology
SHA	Secure Hash Algorithm
SIMD	Single-instruction multiple-data
SPN	Substitution permutation network
XKCP	eXtended (or Xoodoo) Keccak Code Package
XOF	Extendable-output function
XOR	Exclusive-or
WSL	Windows Subsystem for Linux

1. Introduction

We covered a number of cryptographic primitives so far: 1️⃣ stream ciphers and block ciphers (used in conjunction with modes of operation) for protecting data confidentiality, 2️⃣ hash functions for protecting integrity.

This brings us naturally to authentication.

Traditionally, we would cover message authentication codes, but recent trends highlight the advantages of combining authentication with encryption in the form of *authenticated encryption*.

- We first encountered *authenticated encryption* in Lecture 2 where it was defined as a symmetric-key encryption scheme that is *CCA-secure* and *unforgeable*.
- A secure way of constructing an authenticated encryption is the *generic composition* paradigm called *encrypt-then-MAC*.
- There are many scenarios where a ciphertext is accompanied by some metadata or associated data (e.g., packet header) that must be authenticated alongside the ciphertext, resulting in the need for *authenticated encryption with associated data* (AEAD).

  Quiz 1

  Does the associated data need to be confidential?

- In 2017, NIST initiated a process to solicit, evaluate and standardise lightweight AEAD algorithms, and in 2021, after two review rounds, NIST announced ten finalists.

This lecture covers two of the AEAD finalists: Ascon and Xoodyak.

- Ascon is based on the duplex construction, which we studied in the previous lecture.
- Designed by the Keccak team, Xoodyak is based on the full-state keyed duplex construction [DMVA17], which is an extension of the duplex construction.
- Studying Ascon and Xoodyak thus supports a natural progression of learning.

2. ASCON

The Ascon specification [DES21] provides details on

- Three authenticated encryption schemes: Ascon-128, Ascon-128a and Ascon-80pq.
- Two extendable-output functions (XOFs): Ascon-Hash and Ascon-Hasha.

The Ascon specification recommends pairing

- Ascon-128 with Ascon-Hash; and
- Ascon-128a with Ascon-Hasha.

The Ascon family of algorithms is parameterised by key length k, rate (i.e., data block size) r, internal round numbers a and b; see Table 1.

Table 1: Ascon parameters [DES21, Table 1].

Algorithm	k	r	a	b	Nonce	Tag
Ascon-128	128	64	12	6	128	128
Ascon-128a	128	128	12	8	128	128
Ascon-80pq	160	64	12	6	128	128

Note:

- While Ascon-128 and Ascon-128a use 128-bit keys, Ascon-80pq uses 160-bit keys to provide resistance against quantum key search based on Grover's algorithm (which can reduce an ordinarily $O(N)$ search to $O(\sqrt{N})$ using quantum computing, where N is the number of records to search [Gro96]).
- All three schemes use a 128-bit nonce and a 128-bit message authentication code (MAC) tag.

As with any security scheme, the security strength and the computational efficiency are of the utmost concern.

In terms of security:

- All three Ascon authenticated encryption schemes offer in theory 128-bit security.
- The best attacks compromise 7 out of the full 12 rounds. For example, Rohit and Sarkar's attack [RS21] can recover a secret key at a data complexity of 2^{63}, time complexity of $2^{115.2}$, and requires 2^{69} bits of memory.

In terms of computational efficiency:

- Learning the lessons from the Advanced Encryption Standard (AES, see Lecture 4), Ascon was designed to resist cache-timing attacks.

  This requires Ascon to avoid table look-ups, and the standard technique for this is *bistslicing* (see Sec. A.1).

  All Ascon algorithms lend themselves to efficient bitsliced implementations on 64-bit platforms [DES21, Sec. 1].
- Reference and optimised C and assembly implementations of Ascon are provided by the authors on GitHub.

To describe the building blocks of Ascon, we start with an overview:

- All members of the Ascon family operate on a state of 320 bits.
- A state, denoted by S, is divided into an outer part S_r of r bits and an inner part S_c of c bits. This is consistent with the sponge and duplex constructions (see Lecture 6).

  Quiz 2

  Based on the information (including Table 1) thus far, what is the value of capacity c for Ascon-128?
- A state is split into 5 64-bit registers, i.e.,

$$S = S_r \| S_c = x_0 \| x_1 \| x_2 \| x_3 \| x_4, \tag{1}$$

  where $x_0, \ldots, x_4$ denote the content of the 5 registers.
- When interpreted as a byte array, the most significant byte (MSB) of S is the 0th byte, which is also the MSB of x_0; while the least significant byte (LSB) is the 39th byte, which is also the LSB of x_4:

$$x_0 = S_0 \| \cdots \| S_7, \qquad \cdots \qquad , x_4 = S_{32} \| \cdots \| S_{39}.$$

Table 2: Symbols and notation for discussing Ascon in Sec. 2.

Symbols/notation	Meaning
S, S_r, S_c	320-bit state, r-bit outer state and c-bit inner state
$x_0, \dots, x_0$	The 5 64-bit words of the state
$x_{i,j}$	The jth ($0 \leq j \leq 63$) bit of x_i, with $j = 0$ marking the LSB
K	Secret key of length ≤ 160 bits
N, T	Nonce and MAC tag of 128 bits
P, C, A	Plaintext, ciphertext and associated data (subscript i indexes r-bit blocks)
$\mathscr{E}_{k,r,a,b}$, $\mathscr{D}_{k,r,a,b}$	Encryption/decryption using key length k, rate r, number of initialisation and finalisation rounds a, and number of intermediate rounds b
p (p^a)	Permutation function of Ascon (p applied a times)
$\perp$	Message authentication error/failure
$\lvert x \rvert$	Number of bits in bitstring x
$\lfloor x \rfloor_k$	The first (most significant) k bits of bitstring x
$\lceil x \rceil^k$	The last (least significant) k bits of bitstring x

Using the symbols and notation in Table 2, we denote the encryption process by

$$\mathscr{E}_{k,r,a,b}(K, N, A, P) = (C, T). \tag{2}$$

and the decryption process by

$$\mathscr{D}_{k,r,a,b}(K, N, A, C, T) \in \{P, \perp\}. \tag{3}$$

The ensuing discussion covers operations involved in $\mathscr{E}_{k,r,a,b}(K, N, A, P)$ and $\mathscr{D}_{k,r,a,b}(K, N, A, C, T)$ in terms of

- initialisation (Sec. 2.1),
- processing of associated data (Sec. 2.2),
- processing of plaintext/ciphertext (Sec. 2.3),
- finalisation (Sec. 2.4).

As each of the preceding processing stages involves the permutation p, Sec. 2.5 discusses p in detail.

2.1. Initialisation

The initial value of the 320-bit state, S, depends on the secret key K, nonce N, and a 64-bit initialisation vector (IV) specifying the values of k, r, a and b [DES21, Sec.

2.4.1]:

$$\mathrm{IV}_{k,r,a,b} \leftarrow k\|r\|a\|b\|0^{160-k} = \begin{cases} 80400c0600000000 & \text{for Ascon-128,} \\ 80800c0800000000 & \text{for Ascon-128a,} \\ a0400c06 & \text{for Ascon-80pq;} \end{cases} \tag{4}$$

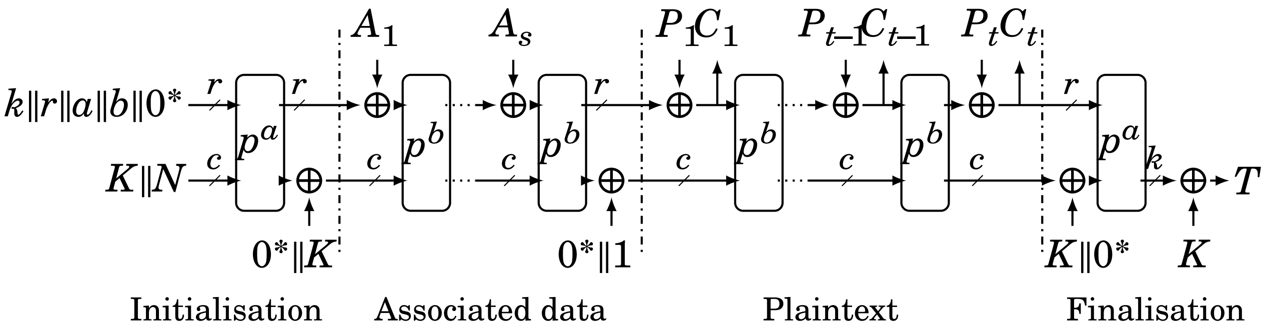
$$S \leftarrow \mathrm{IV}_{k,r,a,b} \,\|K\|N. \tag{5}$$

Quiz 3

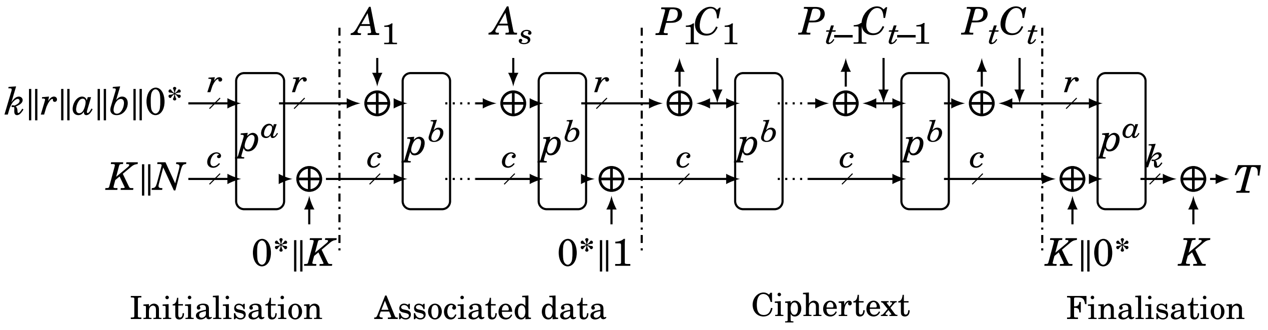
What is the difference between "←" and "="?

To initialise S, a rounds of the round transformation p (more on this later) are applied to S, followed by an XOR of K (see the "Initialisation" part of Figure 1):

$$S \leftarrow p^a(S) \oplus (0^{320-k}\|K) \tag{6}$$



(a) Encryption $\mathscr{E}_{k,r,a,b}(K,N,A,P)$



(b) Decryption $\mathscr{D}_{k,r,a,b}(K,N,A,C,T)$

Figure 1: Every member of the Ascon family applies permutation p $(2a+b)$ number of rounds to a 320-bit state: 1️⃣ a rounds during initialisation, 2️⃣ b rounds when processing associated data and plaintext, and 3️⃣ a rounds during finalisation. Diagrams made using TikZ code from [Jea16].

2.2. Processing of associated data

For processing associated data, A, it is padded with $1\|0^{r-1-(|A| \mod r)}$, so that the result is a multiple of r-bit blocks [DES21, Sec. 2.4.2].

Quiz 4

Suppose A is 13 bytes long, how many zero bits are there in the pad?

Suppose padding results in s blocks of A, then each of these blocks is XORed with S_r (recall Eq. (1)); and in concatenation with S_c, the result is fed to b rounds of permutation p [DES21, Sec. 2.4.2]:

$$S \leftarrow p^b((S_r \oplus A_i) \| S_c), \qquad 1 \le i \le s. \tag{7}$$

Above, note the index of A goes from 1 to s, following the notation in the Ascon specification [DES21].

After b rounds of permutation p, as expressed by Eq. (7), a 1-bit domain separation constant is XORed to S:

$$S \leftarrow S \oplus (0^{319} \| 1). \tag{8}$$

Eqs. (7)–(8) are captured in the "Associated data" part of Figure 1.

2.3. Processing of plaintext/ciphertext

For processing plaintext, P, it is padded in exactly the same way A is padded, as described in the previous subsection [DES21, Sec. 2.4.3].

Suppose padding results in t blocks of P, the first block P_1 is XORed with the state S to produce the first ciphertext block C_1, which is then permuted to update the state. The process is subsequently repeated for $P_2, \ldots, P_t$ [DES21, Sec. 2.4.3]:

$$C_i \leftarrow \begin{cases} S_r \oplus P_i & \text{if } 1 \le i < t, \\ \lfloor S_r \oplus P_i \rfloor_{|P| \bmod r} & \text{if } i = t; \end{cases} \tag{9}$$

$$S \leftarrow \begin{cases} p^b(C_i \| S_c) & \text{if } 1 \le i < t, \\ C_i \| S_c & \text{if } i = t. \end{cases} \tag{10}$$

In Eq. (9), the last ciphertext block C_t is truncated to the length of the unpadded last plaintext block-fragment so that its length is between 0 and $r-1$ bits, and the ciphertext is exactly as long as the plaintext.

Eqs. (9)–(10) are captured in the "Plaintext" part of Figure 1(a).

For processing ciphertext, C, it is padded in exactly the same way A is padded, as described in the previous subsection [DES21, Sec. 2.4.3].

Suppose padding results in t blocks of C, the following statements capture the decryption process [DES21, Sec. 2.4.3]:

$$P_i \leftarrow \begin{cases} S_r \oplus C_i & \text{if } 1 \le i < t, \\ \lfloor S_r \rfloor_{|C_t|} \oplus C_t & \text{if } i = t; \end{cases} \tag{11}$$

$$S \leftarrow \begin{cases} p^b(C_i \| S_c) & \text{if } 1 \le i < t, \\ \left(S_r \oplus (P_t \| 1 \| 0^{r-1-|C_t|})\right) \| S_c & \text{if } i = t. \end{cases} \tag{12}$$

Eqs. (11)–(12) are captured in the “Ciphertext” part of Figure 1(b).

2.4. Finalisation

The finalisation stage of Ascon produces a MAC tag [DES21, Sec. 2.4.4]:

$$S \leftarrow p^a \left(S \oplus (0^r \| K \| 0^{c-k}) \right), \tag{13}$$

$$T \leftarrow \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}. \tag{14}$$

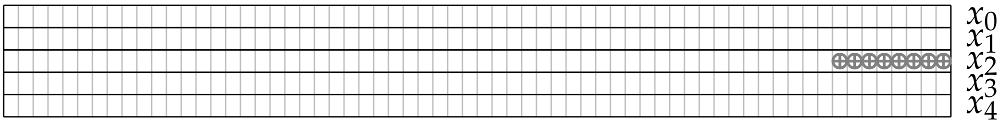
The preceding equations are captured in the “Finalisation” part of Figure 1.

2.5. Ascon permutation

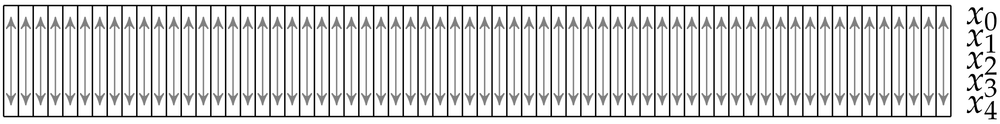
The complexity of the process depicted in Figure 1 lies mainly in the permutation p, which takes the form of a *substitution permutation network* (SPN, see Sec. A.3) [DES21, Sec. 2.6].

The Ascon permutation p is a combination of three successive transformation layers: p_C, then p_S, and finally p_L.

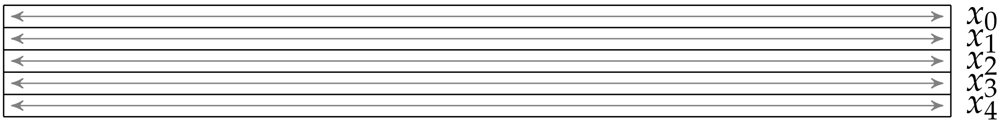
As an overview, Figure 2 illustrates the directions of diffusion effected by the three transformation layers.



(a) Round constant addition p_C



(b) Substitution layer p_S with 5-bit S-box $\mathcal{S}(x)$



(c) Linear layer with 64-bit diffusion functions $\Sigma_i(x_i)$

Figure 2: The three transformations that constitute the permutation p [DES21, Figure 3].

The transformation layers are discussed in turn:

p_C: This layer XORs x_2 (one of the 5 64-bit words in Eq. (1)) with round constant c_i [DES21, Sec. 2.6.1], i.e.,

$$x_2 \leftarrow x_2 \oplus c_i. \tag{15}$$

For $p^a = p^{12}$, the 12 64-bit round constants are 0xf0, 0xe1, 0xd2, ..., 0x4b.

For $p^b = p^8$, the 8 64-bit round constants are 0xb4, 0xa5, 0x96, ..., 0x4b.

For $p^b = p^6$, the 6 64-bit round constants are 0x96, 0x87, 0x78, ..., 0x4b.

Note:

- The last digit of the round constant increments while the other digit decrements with each round.
- The last round constant is 0x4b in every case.

ℹ Detail: Design rationales [DES21, Sec. 5.2.1]

The word x_2 was chosen to enable efficient bitsliced S-box implementations.

Round-dependent round constants thwart slide attacks [BW99].

The low entropy/uncertainty of the round constants is meant to show that the constants are not used to implement any backdoor.

p_S: This is the nonlinear substitution layer, which updates the state with 64 parallel applications of the 5-bit S-box in Figure 3 to each bitslice of the five register words $x_0, \dots, x_4$.

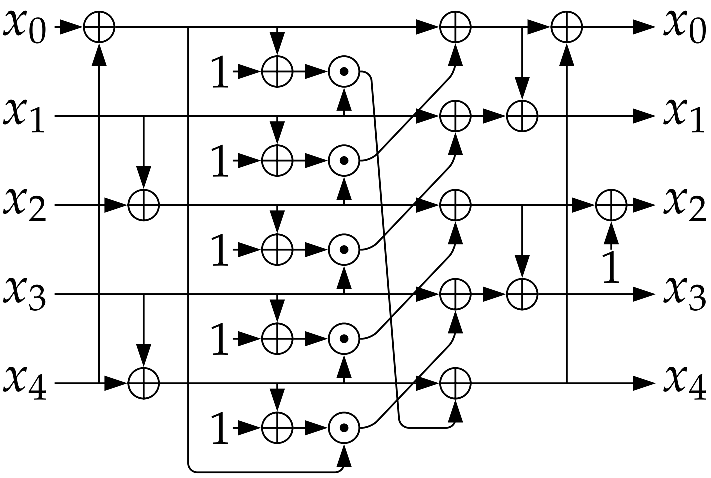


Figure 3: The 5-bit S-box of Ascon [DES21, Figure 4].

ℹ Detail: Design rationales [DES21, Sec. 5.2.2]

Compared to the χ step mapping of Keccak (see Lecture 6), p_S has been specifically designed to 1️⃣ provide higher differential and linear *branch numbers* (3 as opposed to 2), 2️⃣ have no fixed points (as opposed one), and 3️⃣ make each output bit depend on more input bits (4 as opposed to 3).

The branch number is a measure of diffusion [DR20, Ch. 9]. The higher the branch number, the more resistant the scheme is against differential or linear cryptanalysis.

The low algebraic degree of 2 theoretically makes the 5-bit S-box more prone to algebraic attacks, but 1️⃣ a practical attack has yet to be found, and 2️⃣ the simple S-box enables efficient masked implementation (see Sec. A.2) of countermeasures to side-channel analyses.

p_L: This is the linear diffusion layer, which provides diffusion *within each* 64-bit register word:

$$x_0 \leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28), \tag{16a}$$
$$x_1 \leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39), \tag{16b}$$
$$x_2 \leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6), \tag{16c}$$
$$x_3 \leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17), \tag{16d}$$
$$x_4 \leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41). \tag{16e}$$

Above, Σ is not to be confused with summation.

To appreciate (16) as a linear transformation, consider rewriting (16a) as a matrix equation [RAD⁺20, Sec. 2.1]:

$$x_0 \leftarrow \Sigma_0(x_0) = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \vdots & 1 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & \vdots & \vdots & 1 & \vdots & \vdots & \vdots & \vdots & \vdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \vdots & \vdots & \vdots & \vdots & 1 & \vdots & \vdots & \vdots & 0 \\ 0 & \vdots & \vdots & \vdots & \vdots & \vdots & 1 & \vdots & \vdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 1 & 0 & \cdots & 1 & 0 \\ 0 & \cdots & 0 & 1 & \cdots & 0 & 1 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{0,63} \\ \vdots \\ x_{0,29} \\ x_{0,28} \\ \vdots \\ x_{0,20} \\ x_{0,19} \\ \vdots \\ x_{0,1} \\ x_{0,0} \end{bmatrix} \text{ in GF}(2). \tag{17}$$

To understand the pattern of the 64×64 matrix in the preceding equation, notice the last row has ones in the entries corresponding to $x_{0,0}$, $x_{0,19}$ and $x_{0,28}$, as required by (16a). The penultimate row equals the last row left-shifted by by one column, and so on.

ℹ Detail: Design rationales [DES21, Sec. 5.2.3]

The Σ functions were chosen to be similar to the Σ functions in SHA-2 (see Lecture 6), but with one less rotation for efficiency.

The rotation constants (how many bits to rotate through) were chosen to achieve good diffusion after three rounds.

3. XOODYAK

XOODYAK was designed by the KECCAK team under the leadership of the co-inventor of the AES, Joan Daemen; see `https://keccak.team/xoodyak.html`.

In short, XOODYAK is the CYCLIST mode of operation [DHP+21] on top of XOODOO [DHVAVK18], a family of permutation functions parameterised by the number of rounds. Formally,

Definition 1: XOODYAK [DHP+21, Definition 2]

XOODYAK is CYCLIST $[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$ with

- f being XOODOO [12] with width $b = 48$ bytes $= 384$ bits;
- $R_{\text{hash}} = 16$ bytes $= 128$ bits, specifying the block size of a hash, when CYCLIST is used in the hash mode;
- $R_{\text{kin}} = 44$ bytes $= 352$ bits, specifying the block size of an input, when CYCLIST is used in the keyed mode;
- $R_{\text{kout}} = 24$ bytes $= 192$ bits, specifying the block size of an output, when CYCLIST is used in the keyed mode;
- $\ell_{\text{ratchet}} = 16$ bytes $= 128$ bits, specifying the number of bytes of the state to be overwritten with zeros.

The parameters satisfy the constraint (in unit bytes):

$$\max\left(R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}\right) + 2 \leq b,$$

where the term 2 (bytes) is to account for the bits used for padding and domain separation [DHP+21, Sec. 2.2].

The update from KECCAK's width of 320 bits to 384 bits — a perfect fit for 12 32-bit microprocessor registers — was an explicit design decision [DHVAVK18, Sec. 5.7] partly inspired by Gimli [BKL+17].

Definition 1 conveniently sets out the agenda for the ensuing discussion:

- First, we discuss the CYCLIST mode of operation alonsgide its parameters in Sec. 3.1.
- Then, we discuss the XOODOO family of permutations in Sec. 3.2.
- Finally in Sec. 3.3, we summarise the key security and computational efficiency features of XOODYAK.

  In the XOODYAK specification [DHP+21], the key features of XOODYAK are among the first items discussed, but having knowledge of the algorithmic components could help us understand these features better, hence the order of discussion.

3.1. CYCLIST

Cyclist is based on the *full-state keyed duplex construction* [DMVA17], an extension of the *duplex construction* covered in Lecture 6.

The full-state keyed duplex construction, denoted $\mathrm{KD}^f_{\mathbf{K}}$, is illustrated in Figure 4.

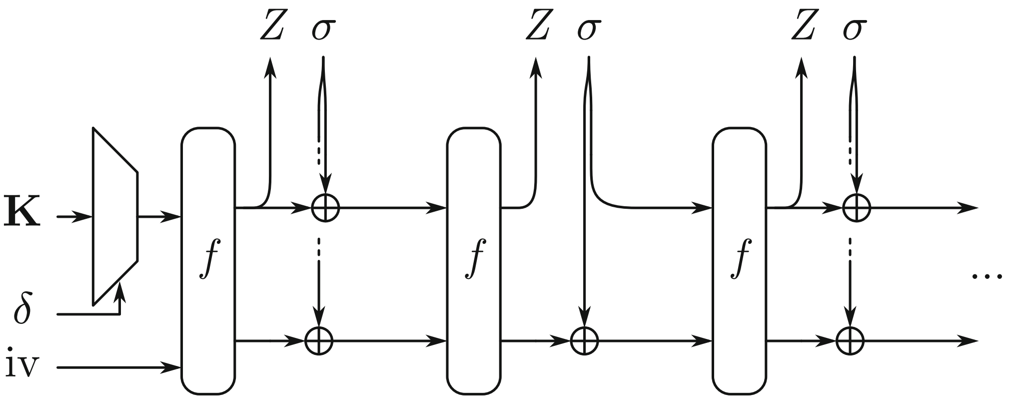


Figure 4: An example of a sequence of function calls to a full-state keyed duplex object $\mathrm{KD}^f_{\mathbf{K}}$: 1️⃣ $Z = \mathrm{KD.Init}(\delta, \mathrm{iv}, \sigma, \mathrm{false})$, 2️⃣ $Z = \mathrm{KD.Duplexing}(\sigma, \mathrm{true})$, 3️⃣ $Z = \mathrm{KD.Duplexing}(\sigma, \mathrm{false})$. Diagram from [DMVA17, Fig. 1].

In Figure 4,

- The input $\mathbf{K}$ is an array/matrix consisting of u keys of size k bits.
- The input δ indexes one of the u keys in $\mathbf{K}$.
- the input iv is an initialisation vector. Together, a k-bit key and an iv are as long as the width b of the permutation f.
- Initialisation is performed by function call KD.Init(δ, iv, σ, flag), which initialises the state to $f(\mathbf{K}[\delta] \| \mathrm{iv})$, where σ is a user-provided string and "flag" is set to true when the outer state is to be overwritten with the outer part of σ.
- Intermediate processing is performed by duplexing call KD.duplexing(σ, flag), where "flag" is as previously defined.

In essence, Cyclist is a *duplex object* extended with an interface for *absorbing* strings of arbitrary length, their encryption and *squeezing* output of arbitrary length.

Clearly, the terminology of *absorbing* and *squeezing* of the *sponge construction* (see Lecture 6) remains applicable here, since it is what the duplex construction is based on.

Cyclist has two modes:

- hash mode — in this mode, Cyclist = sponge;
- keyed mode — in this mode, Cyclist = full-state keyed duplex.

Our focus here is the keyed mode.

Cyclist's accommodation for the two modes above is reflected by its designers' emphasis on the programming/user interface; see the Keccak team's ▶ YouTube video on "Xoodyak, a lightweight cryptographic scheme".

In fact, the Xoodyak specification includes specification of the internal and external interfaces of Cyclist [DHP+21, Algorithms 2 and 3].

For an example of the external interface, consider encrypting this sequence: $A_1 \| P_1$, A_2, P_3, where A_* denotes associated data that needs to be authenticated but not encrypted, and P_* denotes plaintext that needs to be authenticated *and* encrypted. The following function calls are applicable:

Cyclist (K, ID, counter)
Absorb (nonce)
Absorb (A_1), $C_1 \leftarrow$ Encrypt (P_1), $T_1 \leftarrow$ Squeeze (t)
Output (C_1, T_1) and wait for the next message
Absorb (A_2), $T_2 \leftarrow$ Squeeze (t)
Output (T_2) and wait for the next message
$C_3 \leftarrow$ Encrypt (P_3), $T_3 \leftarrow$ Squeeze (t)
Output (C_3, T_3) and wait for the next message

Above ☝,

- A Cyclist object is instantiated with a secret key K, an optional ID for the key, and a counter which plays the role of the iv in Figure 4.

  The counter is "absorbed" in a trickled way starting with the most significant digits (not bits, so a basis of 2 to 2^8 inclusive is applicable) to limit the number of power traces available with distinct inputs [DHP+21, Sec. 3.2.2].

  The idea of trickle-feeding a counter to the Cyclist processing pipeline is credited to Taha and Schaumont [TS14]; see Figure 5.

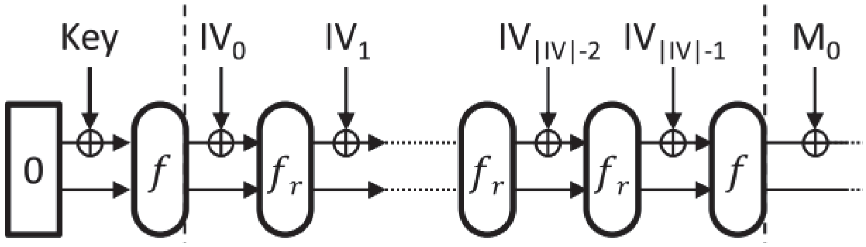


Figure 5: Taha and Schaumont's countermeasure to side-channel analysis, where an IV is trickled-fed to the outer state of Keccak [TS14, Fig. 3]. f_r in the figure is a round-reduced version of the Keccak-f permutation for making space for the incoming IV substrings.

- C_* denotes ciphertext, T_* denotes a MAC tag, and t denotes the desired length of a MAC tag.

- The process is stateful, i.e., a state is maintained going from one function call to the other.
- At any time in keyed mode, the function Ratchet () can be invoked [DHP+21, Sec. 3.2.5], to cause part of the state to be overwritten with zeroes, thereby making it computationally infeasible to compute the state value before the call to Ratchet (). This mitigates the impact of recovering the internal state, e.g., after a side-channel attack.

  This ratchet mechanism (a mechanism that allows movement in only one direction) is a distinct security feature of Xoodyak.

  The ratchet mechanism is paired with the key derivation mechanism implemented by the function SqueezeKey (), which works like Squeeze () but in the key space for the purpose of deriving rolling subkeys [DHP+21, Secs. 2.2 and 3.2.6].

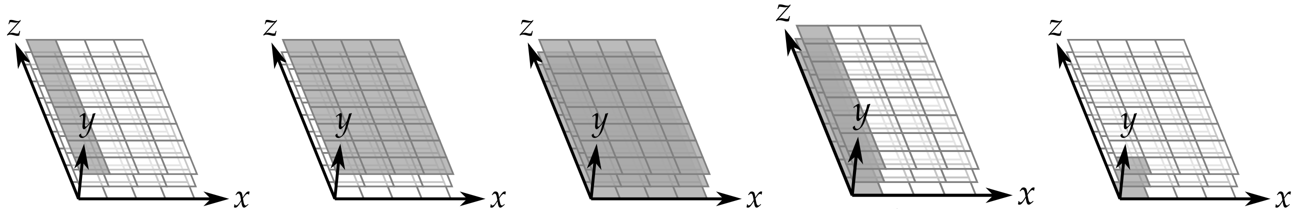
  Using the rolling subkeys derived from the long-term key instead of the long-term key itself provides resilience against side-channel attacks by making the secret key a moving target.

3.2. Xoodoo

Xoodoo is a family of permutations parameterised by the number of rounds n_r, hence the notation Xoodoo $[n_r]$.

Xoodoo is iterated, i.e., it iteratively applies a round function to a state.

A Xoodoo state consists of 3 equally sized horizontal *planes*, each one consisting of 4 parallel 32-bit *lanes*; see Figure 6.



lane plane state sheet column

Figure 6: Parts of a Xoodoo state [DHVAVK18, Figure 2]. Planes are indexed from $y = 0$ (bottom) to $y = 2$ (top). Every lane shown here is 8 bits long but should be understood as 32 bits long. The terminology here is inherited from Keccak, but "row" and "slice" are not applicable.

In analysis, a state bit is indexed based on the Cartesian coordinate system in Figure 6, where $0 \leq x,y \leq 3$ and $0 \leq z \leq 31$, and denoted by $\mathbf{A}[x,y,z]$.

In implementation, the state is stored in a flattened form, i.e.,

$$S = S[0] \| S[1] \| \cdots \| S[b-1],$$

which is related to $\mathbf{A}[x,y,z]$ by

$$\mathbf{A}[x,y,z] = S[\,(x+4)\cdot 32 + z\,],$$

i.e., $\mathbf{A}$ is serialised into S lane-first (lane as defined in Figure 6), then by x and y. ☝ Compare this with what was said about the Keccak state in Lecture 6.

A round function in the Xoodoo permutation consists of 5 step mappings:

1. the mixing layer θ,
2. the "western" plane shifting layer ρ_{west},
3. the round-constant addition layer ι,
4. the nonlinear layer χ,
5. the "eastern" plane shifting layer ρ_{east}.

Compare the step mappings of Xoodoo with those of Keccak:

1. the mixing layer θ,
2. the intra-lane translation layer ρ
3. the intra-slice transposition layer π,
4. the nonlinear layer χ,
5. the round-constant addition layer ι.

Table 3: Symbols and notation for discussing Xoodyak in Sec. 3.

Symbols/notation	Meaning
$\mathbf{K}$, u	An array/matrix of user keys and number of user keys in $\mathbf{K}$
n_r	Number of rounds
$\mathbf{A}_y$	Plane (defined in Figure 6) y of state $\mathbf{A}$
$\overline{\mathbf{A}_y}$	Bitwise complement of $\mathbf{A}_y$
$\mathbf{A}_y \lll (\Delta x, \Delta z)$	Rotation of $\mathbf{A}_y$ moving bit at (x,z) to $(x+\Delta x, z+\Delta z)$
$\mathbf{A}_y + \mathbf{A}_{y'}$	Bitwise sum (XOR) of planes $\mathbf{A}_y$ and $\mathbf{A}_{y'}$
$\mathbf{A}_y \cdot \mathbf{A}_{y'}$	Bitwise product (AND) of planes $\mathbf{A}_y$ and $\mathbf{A}_{y'}$

Based on the symbols and notation in Table 3, the step mappings are defined as follows.

Step mapping θ:

$$\mathbf{A}_y \leftarrow \mathbf{A}_y + \left(\sum_{j=0}^{2} \mathbf{A}_j\right) \lll (1,5) + \left(\sum_{j=0}^{2} \mathbf{A}_j\right) \lll (1,14), \qquad y = 0,1,2. \quad (18)$$

Equivalently [ZZS21, Sec. 2.2],

$$\mathbf{A}[x,y,z] \leftarrow \mathbf{A}[x,y,z] \oplus \left(\sum_{j=0}^{2} \mathbf{A}[x-1,j,z-5]\right) \oplus \left(\sum_{j=0}^{2} \mathbf{A}[x-1,j,z-14]\right). \quad (19)$$

Like Keccak's θ, Xoodoo's θ a column parity mixing layer [SD18]; it is linear and the design rationales for both are similar [DHVAVK18, Sec. 7.3.1].

To achieve a dense *parity-folding matrix* (see Definition 2) for good diffusion, and so that θ can be inverted for decryption, θ needs to operate on columns of odd size [SD18, Corollary 2 and Sec. 7], explaining the y-dimension of the Keccak and Xoodoo state.

Definition 2: Parity-folding matrix [SD18, Sec. 2.3]

The *column parity* of a matrix $\mathbf{A}$ is a row vector defined as $\vec{1}_m^\top \mathbf{A}$, where $\vec{1}_m$ is an m-dimensional vector of 1's.

For example, for

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix},$$

the column parity is

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

The *expanded column pairty* of $\mathbf{A}$ is a matrix with m rows all equal to the column parity of $\mathbf{A}$, and it is given by $\mathbf{1}_{m\times m}\mathbf{A}$.

If θ is the column parity mixer, then the parity-folding matrix $\mathbf{Z}$ satisfies the equation:

$$\theta(\mathbf{A}) = \mathbf{A} + \mathbf{1}_{m\times m}\mathbf{A}\mathbf{Z}.$$

Step mapping ρ_{west} (see Figure 7):

$$\mathbf{A}_2 \leftarrow \mathbf{A}_2 \lll (0, 11), \tag{20a}$$
$$\mathbf{A}_1 \leftarrow \mathbf{A}_1 \lll (1, 0). \tag{20b}$$

Equivalently [ZZS21, Sec. 2.2],

$$\mathbf{A}[x, 2, z] \leftarrow \mathbf{A}[x, 2, z - 11], \tag{21a}$$
$$\mathbf{A}[x, 1, z] \leftarrow \mathbf{A}[x - 1, 1, z], \tag{21b}$$

🔎 Note intra-plane diffusion does not affect $\mathbf{A}_0$.

ρ_{west} is by design the dispersion (Daemen's term [DR20, p. 133]) layer after θ, so that bits or bytes that are close to each other after θ are moved to positions that are distant [DHVAVK18, Secs. 5.1 and 5.7].

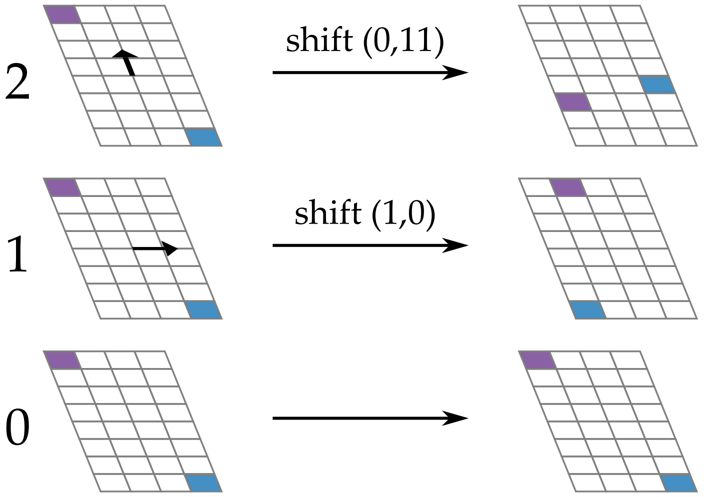


Figure 7: Xoodoo's ρ_{west} step mapping [DHVAVK18, Figure 5].

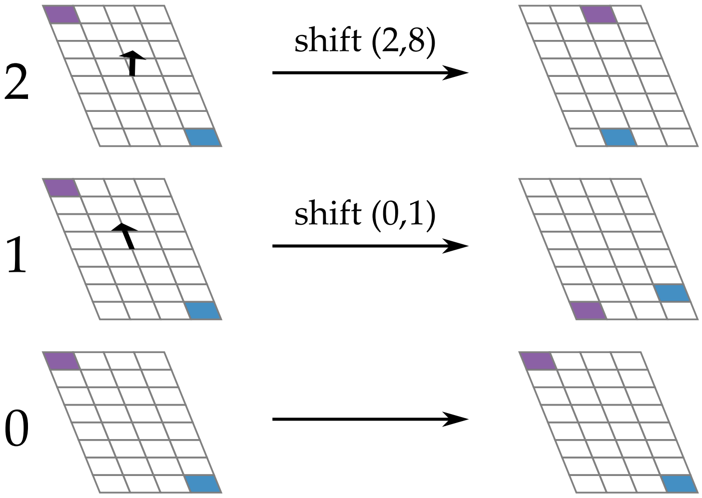


Figure 8: Xoodoo's ρ_{east} step mapping [DHVAVK18, Figure 5].

Step mapping ι:

$$\mathbf{A}_0 \leftarrow \mathbf{A}_0 + \mathbf{C}_i, \qquad -11 \le i \le 0. \tag{22}$$

Above,

- The round constant $\mathbf{C}_i$ is only nonzero for the two least significant bytes of the lane at $(x, y) = (0, 0)$, i.e.,

$$\mathbf{A}[0,0,z] \leftarrow \mathbf{A}[0,0,z] \oplus \mathbf{C}_i, \qquad -11 \le i \le 0.$$

- $i = -11$ applies to the first round, and $i = 0$ applies to the last round.
- The round constants can be found in [DHVAVK18, Table 2] and are omitted here.

Round-dependent round constants thwart slide attacks [BW99].

The round constants were chosen to destroy the translation-invariance / shift-invariance and hence symmetry of the round function [DHVAVK18, Sec. 5.6], helping to thwart cryptanalysis.

Step mapping χ (see Figure 9):

$$\mathbf{A}_2 \leftarrow \mathbf{A}_2 + \overline{\mathbf{A}_0} \cdot \mathbf{A}_1, \tag{23a}$$
$$\mathbf{A}_1 \leftarrow \mathbf{A}_1 + \overline{\mathbf{A}_2} \cdot \mathbf{A}_0, \tag{23b}$$
$$\mathbf{A}_0 \leftarrow \mathbf{A}_0 + \overline{\mathbf{A}_1} \cdot \mathbf{A}_2. \tag{23c}$$

Equivalently [ZZS21, Sec. 2.2],

$$\mathbf{A}[x,y,z] \leftarrow \mathbf{A}[x,y,z] \oplus \left((\mathbf{A}[x,y+1 \bmod 3,z] \oplus 1) \wedge \mathbf{A}[x,y+2 \bmod 3,z] \right) . \quad (24)$$

Unlike KECCAK's χ which operates on 5 bits, XOODOO's χ operates on 3 bits, so XOODYAK has 4×32 3-bit S-boxes.

χ is based on the shift-invariant quadratic mapping of the same name that Daemen analysed in his PhD thesis [Dae95] decades ago, so its properties are well known.

χ is by design involutive (a function that is its own inverse) [DHVAVK18, Sec. 1.2], and has an algebraic degree of 2 [Dae95, Sec. 6.9]. This contributes to 1️⃣ the ease of analysis of XOODOO in terms of its resistance to differential and linear cryptanalyses [Dae95, Sec. 6.9]; and 2️⃣ ease of masked implementations.

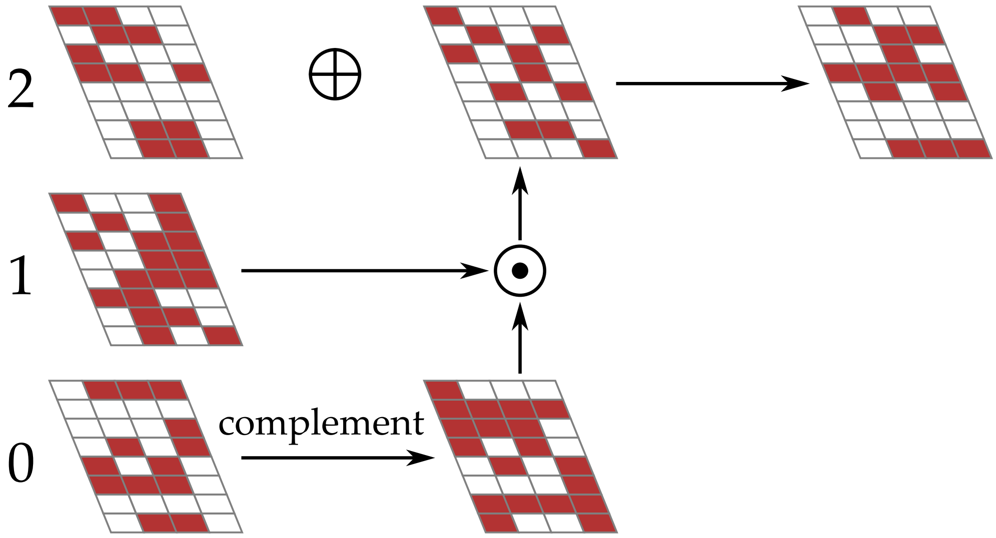


Figure 9: Effect of χ on plane $\mathbf{A}_2$ [DHVAVK18, Figure 3].

Step mapping ρ_{east} (see Figure 8):

$$\mathbf{A}_2 \leftarrow \mathbf{A}_2 \lll (2,8), \quad (25a)$$
$$\mathbf{A}_1 \leftarrow \mathbf{A}_1 \lll (0,1). \quad (25b)$$

Equivalently [ZZS21, Sec. 2.2],

$$\mathbf{A}[x,2,z] \leftarrow \mathbf{A}[x-2,2,z-8], \quad (26a)$$
$$\mathbf{A}[x,1,z] \leftarrow \mathbf{A}[x,1,z-1]. \quad (26b)$$

The designers experimented with only one dispersion layer but found out that did not achieve enough dispersion, so added this second dispersion layer after χ.

Although both ρ_{west} and ρ_{east} do not affect plane $\mathbf{A}_0$, ι only affects $\mathbf{A}_0$. This is a result of trade-off between diffusion and computational efficiency.

3.3. Summary of key features

With knowledge of the algorithmic components of Xoodyak in mind, we now summarise the key security and computational efficiency features of Xoodyak.

In terms of security,

- The design of Xoodyak was based on considerations of a wide range of attacks, e.g., slide attacks [BW99], multi-target attacks [Bih02], side-channel attacks; and has incorporated state-of-the-art countermeasures, e.g., Taha and Schaumont's [TS14].
- Xoodyak protects the secret key through a ratchet and key derivation mechanism (see Sec. 3.1), offering leakage resilience.
- Xoodyak has desirable properties in terms of resistance to differential cryptanlaysis [DHVAVK18, Sec. 1.2], and furthermore it lends itself to efficient masking and threshold countermeasures against differential power analysis and similar attacks [DHP+21].
- The best attack is attributed to Zhou et al. [ZLD+20] who could recover a key in 2^{44} time with negligible memory cost in the nonce-misuse setting if Xoodyak uses 6 rounds instead of the full 12 rounds. Thus, full-round Xoodyak offers nominal 128-bit security.

In terms of computational efficiency,

- The atomic operations (shift, XOR, AND) are simple and lightweight [DHP+21, Sec. 5.1].
- Abundant symmetry enables a high level of code/circuit reuse [DHP+21, Sec. 5.1].
- However, Xoodyak is inherently serial at the construction level [DHP+21, Sec. 1.4].
- Reference and optimised implementations of Xoodyak can be found in the eXtended (or Xoodoo) Keccak Code Package (XKCP) on GitHub.

3.4. Experiments with XKCP

This subsection documents some simple experiments with XKCP.

⚠ For your research paper,

there is no need to reproduce the results below.

To reproduce the experimental results reported in this lecture, these software prerequisites must be satisfied:

- Ubuntu 22.04 LTS environment running on Windows Subsystem for Linux (WSL) version 2.

  Check out this guide to installing Ubuntu on WSL2 on Windows 10, or this guide to installing Ubuntu on WSL2 on Windows 11.

- git, gcc, make and xsltproc in Ubuntu 22.04 on WSL2. These can be installed through command:

  ```
  sudo apt install build-essential xsltproc
  ```

- Visual Studio Code in Windows.

Below are the steps for reproducing the experimental results reported in this lecture:

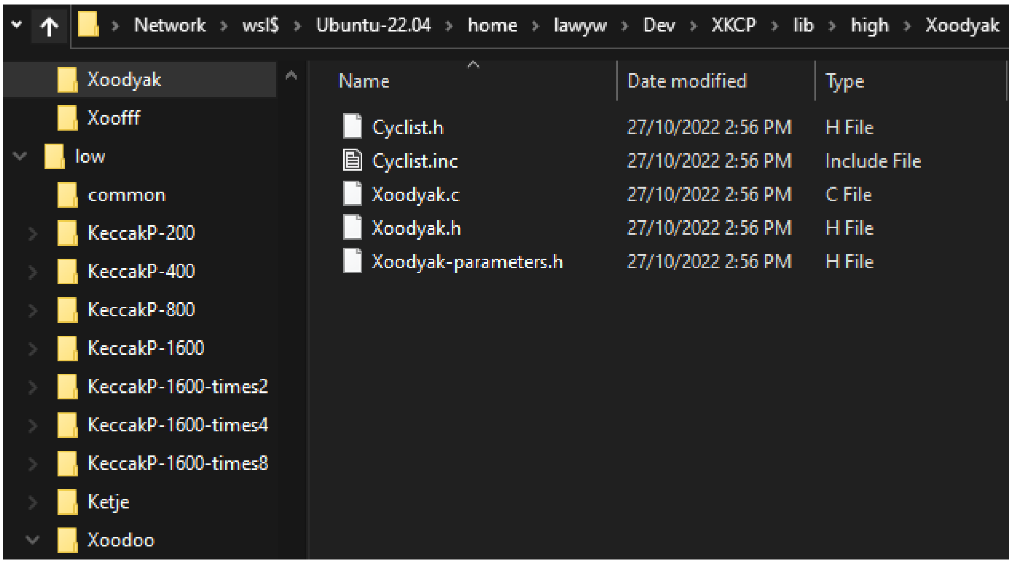
1. This needs only be done once to clone the XKCP repository on GitHub to a local repository:

   ```
   git clone https://github.com/XKCP/XKCP
   ```

   This command — to be executed in the XKCP directory — updates the local repository:

   ```
   git pull
   ```

2. Assuming the local repository is at `/home/lawyw/Dev` in Ubuntu, Figure 10 shows how the local repository can be accessed in Windows.



Name	Date modified	Type
Cyclist.h	27/10/2022 2:56 PM	H File
Cyclist.inc	27/10/2022 2:56 PM	Include File
Xoodyak.c	27/10/2022 2:56 PM	C File
Xoodyak.h	27/10/2022 2:56 PM	H File
Xoodyak-parameters.h	27/10/2022 2:56 PM	H File

Figure 10: Accessing XKCP files in WSL2.

🔎 High-level Xoodyak files lie in the directory `XKCP/lib/high/Xoodyak`, while low-level files lie in the directories 1️⃣ `XKCP/lib/low/common`, 2️⃣ `XKCP/lib/low/Xoodoo`, 3️⃣ `XKCP/lib/low/Xoodoo-times4`, 4️⃣ `XKCP/lib/low/Xoodoo-times8`, 5️⃣ `XKCP/lib/low/Xoodoo-times16`. 👈 The last three directories store parallelised versions of the implementation of Xoodoo, e.g., `times4` means implementation of 4 parallelised instances of Xoodoo.

3. The last comment in the file `Makefile.build` provides the instruction on how to specific a build target:

   ```
   <!-- Target names are of the form x/y where x is taken from the
   first set and y from the second set. -->
   <group all="XKCP">
   <product delimiter="/">
       <factor set="reference reference32bits compact generic32
       generic32lc generic64 generic64lc SSSE3 AVX XOP AVX2 AVX2noAsm
       AVX512 AVX512noAsm ARMv6 ARMv6M ARMv7M ARMv7A ARMv8A AVR8"/>
       <factor set="UnitTests Benchmarks KeccakSum libXKCP.a libXKCP.so
       libXKCP.dylib"/>
   </product>
   </group>
   ```

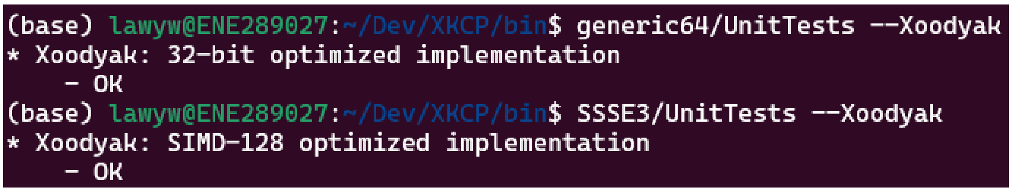
   Let us choose the build targets to be `generic64/UnitTests`, `generic64/Benchmarks`, `SSSE3/UnitTests` and `SSSE3/Benchmarks`:

   ```
   make generic64/UnitTests generic64/Benchmarks
   make SSSE3/UnitTests SSSE3/Benchmarks
   ```

   ⚠ SSSE3 is Intel's Supplemental Streaming SIMD Extension 3 instruction set [Int22], and not supported on contemporary Apple computers.

   Upon successful execution of the `make` commands, these directories will be created: `XKCP/bin/generic` and `XKCP/bin/SSSE3`. Furthermore, each of these directories will contain two executable files: `UnitTests` and `Benchmarks`.

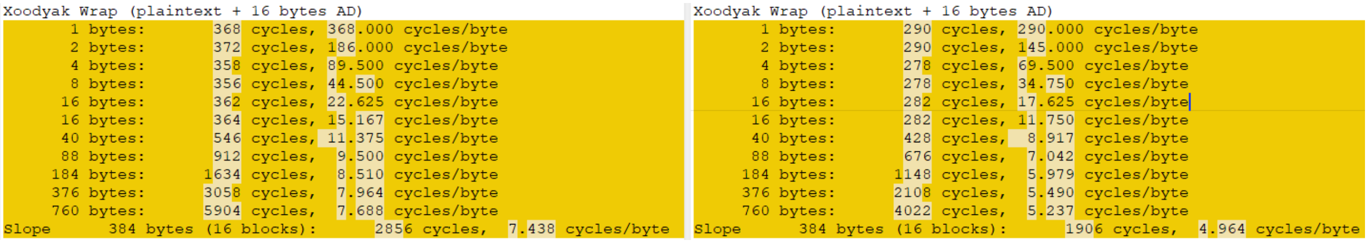
4. Unit tests: The executable `UnitTests` checks if the Xoodyak implementation works as expected. Figure 11 shows the expected output if both the `generic64` and `SSSE3` versions of `UnitTests` completed without issue for Xoodyak.



```
(base) lawyw@ENE289027:~/Dev/XKCP/bin$ generic64/UnitTests --Xoodyak
* Xoodyak: 32-bit optimized implementation
    - OK
(base) lawyw@ENE289027:~/Dev/XKCP/bin$ SSSE3/UnitTests --Xoodyak
* Xoodyak: SIMD-128 optimized implementation
    - OK
```

Figure 11: Successful unit test results for Xoodyak.

5. Benchmarks: Figure 12 compares the outputs of the `generic64` and `SSSE3` versions of `Benchmarks` for Xoodyak.



```
Xoodyak Wrap (plaintext + 16 bytes AD)
       1 bytes:         368 cycles, 368.000 cycles/byte
       2 bytes:         372 cycles, 186.000 cycles/byte
       4 bytes:         358 cycles, 89.500 cycles/byte
       8 bytes:         356 cycles, 44.500 cycles/byte
      16 bytes:         362 cycles, 22.625 cycles/byte
      16 bytes:         364 cycles, 15.167 cycles/byte
      40 bytes:         546 cycles, 11.375 cycles/byte
      88 bytes:         912 cycles,  9.500 cycles/byte
     184 bytes:        1634 cycles,  8.510 cycles/byte
     376 bytes:        3058 cycles,  7.964 cycles/byte
     760 bytes:        5904 cycles,  7.688 cycles/byte
Slope      384 bytes (16 blocks):        2856 cycles,  7.438 cycles/byte
```

```
Xoodyak Wrap (plaintext + 16 bytes AD)
       1 bytes:         290 cycles, 290.000 cycles/byte
       2 bytes:         290 cycles, 145.000 cycles/byte
       4 bytes:         278 cycles, 69.500 cycles/byte
       8 bytes:         278 cycles, 34.750 cycles/byte
      16 bytes:         282 cycles, 17.625 cycles/byte
      16 bytes:         282 cycles, 11.750 cycles/byte
      40 bytes:         428 cycles,  8.917 cycles/byte
      88 bytes:         676 cycles,  7.042 cycles/byte
     184 bytes:        1148 cycles,  5.979 cycles/byte
     376 bytes:        2108 cycles,  5.490 cycles/byte
     760 bytes:        4022 cycles,  5.237 cycles/byte
Slope      384 bytes (16 blocks):        1906 cycles,  4.964 cycles/byte
```

Figure 12: `generic64` vs `SSSE3` benchmark results for Xoodyak.

Thus, the SIMD-optimised version is 33% more efficient than the unoptimised version.

A. Appendix: Glossary

This appendix provides brief explanation of terms that appear in the discussion above but are tangential to the main topics of this lecture.

A.1. Bitslicing

Bitslicing is the standard technique to avoid table look-ups without compromising efficiency [DR20, Sec. 4.2.2].

The technique was invented by Biham [Bih97] for the Data Encryption Standard (DES) and later adapted to other ciphers. For example, Käsper and Schwabe's bitslicing technique for the AES [KS09] is highly cited.

The method of bitslicing views a processor — say a 64-bit processor — as a single-instruction multiple-data (SIMD) computer that can perform 64 one-bit operations simultaneously, while the 64 bits of each block are set in 64 different words, of which the first bit is associated with the first block, the second bit associated with the second block, etc. [Bih97].

Watch Thomas Pornin's ▶ YouTube video on "BearSSL: SSL for All Things" for a quick introduction to bitslicing.

A.2. Masking

Masking is a class of techniques for randomising processed data to obscure intermediate values from cryptanalysts in their attempts to exploit information leakage for side-channel attacks [SMN21].

For example,

- r is a Boolean mask when XORed with bitstring x to obscure the value of x: $x \oplus r$ [Mes01].
- r is an arithmetic mask when added to bitstring x in a Galois field of cardinality 2^n to obscure the value of x: $x + r \mod 2^n$ [Mes01].

A.3. Substitution-permutation network (SPN)

The SPN is a widely used structure, e.g., it is used by the AES [Bak22, Sec. 2.3.1] although this was not mentioned in Lecture 4.

An SPN effects diffusion and confusion in an iterated approach, i.e., in rounds.

In an SPN, a round typically consists of 1️⃣ a non-linear transformation (usually using S-boxes), followed by 2️⃣ a linear transformation, and then 3️⃣ addition of the round key.

B. References

[Bak22] A. Baksi, *Classical and Physical Security of Symmetric Key Cryptographic Algorithms*, *Computer Architecture and Design Methodologies*, Springer Nature Singapore Pte Ltd, 2022. https://doi.org/10.1007/978-981-16-6522-6. 23

[BKL^{+}17] D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, and B. Viguier, Gimli : A cross-platform permutation, in *Cryptographic Hardware and Embedded Systems – CHES 2017* (W. Fischer and N. Homma, eds.), Springer International Publishing, Cham, 2017, pp. 299–320. 11

[Bih97] E. Biham, A fast new DES implementation in software, in *Fast Software Encryption*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 260–272. https://doi.org/10.1007/BFb0052352. 22

[Bih02] E. Biham, How to decrypt or even substitute DES-encrypted messages in 2^{28} steps, *Information Processing Letters* **84** no. 3 (2002), 117–124. https://doi.org/10.1016/S0020-0190(02)00269-7. 19

[BW99] A. Biryukov and D. Wagner, Slide attacks, in *Fast Software Encryption* (L. Knudsen, ed.), Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 245–259. 9, 17, 19

[Dae95] J. Daemen, *Cipher and hash function design strategies based on linear and differential cryptanalysis*, Ph.D. thesis, KU Leuven, 1995. Available at https://cs.ru.nl/~joan/papers/JDA_Thesis_1995.pdf. 18

[DHP^{+}20] J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, Xoodyak, a lightweight cryptographic scheme, *IACR Transactions on Symmetric Cryptology* **2020** no. S1 (2020), 60–87, see also [DHP^{+}21]. https://doi.org/10.13154/tosc.v2020.iS1.60-87. 24

[DHP$^+$21] J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, Xoodyak, a lightweight cryptographic scheme, Xoodyak specification v2, May 2021, see also [DHP$^+$20]. Available at https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/xoodyak-spec-final.pdf. 11, 13, 14, 19, 23

[DHVAVK18] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer, The design of Xoodoo and Xoofff, *IACR Transactions on Symmetric Cryptology* **2018** no. 4 (2018), 1–38. https://doi.org/10.13154/tosc.v2018.i4.1-38. 11, 14, 16, 17, 18, 19

[DMVA17] J. Daemen, B. Mennink, and G. Van Assche, Full-state keyed duplex with built-in multi-user support, in *Advances in Cryptology – ASIACRYPT 2017* (T. Takagi and T. Peyrin, eds.), Springer International Publishing, Cham, 2017, pp. 606–637. 3, 12

[DR20] J. Daemen and V. Rijmen, *The Design of Rijndael: The Advanced Encryption Standard (AES)*, 2nd ed., *Information Security and Cryptography*, Springer-Verlag, Heidelberg, 2020. https://doi.org/10.1007/978-3-662-60769-5. 10, 16, 22

[DES21] C. Dobraunig, M. Eichlseder, and F. M. M. Schläffer, Ascon v1.2: Submission to NIST, May 2021. Available at https://ascon.iaik.tugraz.at. 3, 4, 6, 7, 8, 9, 10

[Gro96] L. K. Grover, A fast quantum mechanical algorithm for database search, in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, *STOC '96*, Association for Computing Machinery, New York, NY, USA, 1996, p. 212–219. https://doi.org/10.1145/237814.237866. 3

[Int22] Intel Corporation, *Intel® Architecture Instruction Set Extensions and Future Features: Programming Reference*, September 2022, 319433-046. Available at https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-refere
html. 21

[Jea16] J. Jean, TikZ for Cryptographers, https://www.iacr.org/authors/tikz/, 2016. 6

[KS09] E. Käsper and P. Schwabe, Faster and timing-attack resistant AES-GCM, in *Cryptographic Hardware and Embedded Systems - CHES 2009* (C. Clavier and K. Gaj, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 1–17. https://doi.org/10.1007/978-3-642-04138-9_1. 22

[Mes01] T. S. Messerges, Securing the AES finalists against power analysis attacks, in *Fast Software Encryption* (G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 150–164. 22

[RAD+20] K. Ramezanpour, A. Abdulgadir, W. Diehl, J.-P. Kaps, and P. Ampadu, Active and passive side-channel key recovery attacks on ascon, in *NIST Lightweight Cryptography Workshop*, 2020, presentation at `https://csrc.nist.gov/CSRC/media/Presentations/active-passive-side-channel-key-attacks-on-ascon/images-media/session-5-ramezanpour-active-passive-ascon.pdf`. Available at `https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/active-passive-recovery-attacks-ascon-lwc2020.pdf`. 10

[RS21] R. Rohit and S. Sarkar, Diving deep into the weak keys of round reduced Ascon, *IACR Transactions on Symmetric Cryptology* **2021** no. 4 (2021), 74–99. `https://doi.org/10.46586/tosc.v2021.i4.74-99`. 4

[SMN21] P. Socha, V. Miškovský, and M. Novotný, High-level synthesis, cryptography, and side-channel countermeasures: A comprehensive evaluation, *Microprocessors and Microsystems* **85** (2021), 104311. `https://doi.org/10.1016/j.micpro.2021.104311`. 22

[SD18] K. Stoffelen and J. Daemen, Column parity mixers, *IACR Transactions on Symmetric Cryptology* **2018** no. 1 (2018), 126–159. `https://doi.org/10.13154/tosc.v2018.i1.126-159`. 16

[TS14] M. Taha and P. Schaumont, Side-channel countermeasure for sha-3 at almost-zero area overhead, in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2014, pp. 93–96. `https://doi.org/10.1109/HST.2014.6855576`. 13, 19

[ZZS21] Z. Zhang, W. Zhang, and H. Shi, Genetic algorithm assisted state-recovery attack on round-reduced Xoodyak, in *Computer Security – ESORICS 2021* (E. Bertino, H. Shulman, and M. Waidner, eds.), Springer International Publishing, Cham, 2021, pp. 257–274. 15, 16, 18

[ZLD+20] H. Zhou, Z. Li, X. Dong, K. Jia, and W. Meier, Practical Key-Recovery Attacks On Round-Reduced Ketje Jr, Xoodoo-AE And Xoodyak, *The Computer Journal* **63** no. 8 (2020), 1231–1246. `https://doi.org/10.1093/comjnl/bxz152`. 19