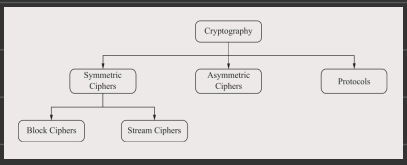
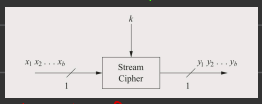
XOR: $x + y \bmod 2 \equiv x \oplus y$

Chapter 2: Stream Ciphers

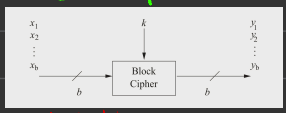


→ Symmetric cryptography → Block Ciphers & Stream Ciphers (Distinguishable)

Stream Cipher:



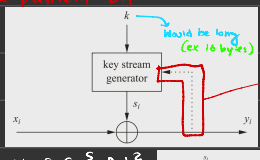
Block Cipher:



→ Stream Ciphers: Encrypt bits individually by adding a bit from a key stream to a plaintext bit.
- ↳ Synchronous: key stream depends only on the key → MOST PRACTICAL (small, fast, less resources = efficient)
- ↳ Asynchronous: key stream depends on key and ciphertext

→ Block Ciphers: Encrypt a block of b plaintext bits at a time with the same key
- ↳ constructed s.t. encrypt. of any plaintext bit in a block depends on every other bit in the block
- ↳ In practice: Block Length = 128 bits (AES) or 64 bits (DES) → Used more than Stream Ciphers

→ How does encryption of single bit work? Each bit x_i encrypted by adding key stream bit s_i modulo 2



→ Stream Cipher Encryption & Decryption: Plaintext, ciphertext and key stream consist of $x_i, y_i, s_i \in \{0,1\}$

↳ Encryption: $y_i = e_{s_i}(x_i) \equiv x_i + s_i \bmod 2 \equiv x_i \oplus s_i$

↳ Decryption: $x_i = d_{s_i}(y_i) \equiv y_i + s_i \bmod 2 \equiv y_i \oplus s_i$ → Cancels out key bit s_i

Why are e_{s_i} and d_{s_i} the same F_2?

→ Correctness Proof: WTS decrypt f^* produces x_i again. We know $y_i \equiv x_i + s_i \bmod 2$.

$$d_{s_i}(y_i) \equiv y_i + s_i \bmod 2$$
$$\equiv (x_i + s_i) + s_i \bmod 2$$
$$\equiv x_i + 2s_i \bmod 2$$
$$\equiv x_i + 0 \bmod 2$$
$$\equiv x_i \bmod 2 \quad \text{Q.E.D}$$

Why is Modulo 2 Addition a Good Encryption F_2?

→ If we do $\bmod 2$ → only possible values are $\{0,1\}$ → Treat as 2 Bool F_2

→ Why is XOR useful? Encrypt some $x_i = 0$. Depending on s_i, y_i is either 0 or 1.

If s_i is random, 50% chance for 0 and 1. → same thing for $x_i = 1$

↓ proves XOR is balanced

x_i	s_i	$x_i + s_i \bmod 2$ (XOR)
0	0	0
0	1	1
1	0	1
1	1	0

Key → magic box → ∞ bitstream

E.g.

Example 2.1. Alice wants to encrypt the letter A, where the letter is given in ASCII code. The ASCII value for A is $65_{10} = 1000001_2$. Let's furthermore assume that the first key stream bits are $(s_0, \dots, s_6) = 0101100$.

Alice	Oscar	Bob
$x_0, \dots, x_6 = 1000001 = A$		$y_0, \dots, y_6 = 1101101 = m$
$\oplus$		$\oplus$
$s_0, \dots, s_6 = 0101100$		$s_0, \dots, s_6 = 0101100$
$y_0, \dots, y_6 = 1101101 = m$ →		$x_0, \dots, x_6 = 1000001 = A$

What exactly is the nature of Key Stream? → determines security

→ Key Stream is the central issue for Stream Ciphers
- ↳ NOT the bits themselves → HARD TO GENERATE
- → Algorithm generating unbounded # of secret bits
- → Actual key is used as input/seed

Random Looking key streams better.

Random Numbers and an Unbreakable Stream Cipher:

→ True Random Number Generators (TRNGs): Output cannot be reproduced and based on physical process that can't be reproduced → Ex: Rolling of dice by humans, coin flipping
- ↳ often needed for generating session keys distributed between Alice and Bob

→ Pseudorandom RNGs (PRNGs): Generate sequences computed from an initial seed value in following way: s_0 = seed, $s_{i+1} = f(s_i)$, $i \in \mathbb{N}$
- ↳ Generalization of this is PRNGs of the form: $s_{i+1} = f(s_i, s_{i-1}, \dots, s_{i-t})$ where t is a fixed integer

→ Ex: Linear Congruential Generator: s_0 = seed, $s_{i+1} = a s_i + b \bmod m$, $i \in \mathbb{N}$ and a, b, m are integer constants

Ex: rand() f^* in C: $s_0 = 12345$, $s_{i+1} = 1103515245\, s_i + 12345 \bmod 2^{31}$, $i \in \mathbb{N}$ → Pros: Good stat. properties ⇒ approximates seq. of true random numbers

* PRNGs are not really random cuz they can be computed and are deterministic.

→ Cryptographically Secure Pseudorandom number generators (CSPRNGs): Type of PRNG which is unpredictable ⇒ Given n output bits of key stream $s_i, s_{i+1}, \dots, s_{i+n-1}$ where $n \in \mathbb{Z}^+$, it is computationally infeasible to compute $s_{i-1}, s_{i-2}, \dots$ or $s_{i+n}, s_{i+n+1}, \dots$ with better than 50% success
- ↳ Also computationally infeasible to generate $s_{i-1}, s_{i-2}, \dots$ NOT NECESSARY FOR CRYPTOGRAPHY

→ Def^n for Unconditional Security: Cryptosystem is unconditionally secure if it cannot be broken with as computational resources

→ Ex: Say we use symmetric alg. w/ key length = 10,000 bits and only attack is BFA. Assume attacker has 2^{10000} computers where each checks 1 key.
- ∴ cipher is computationally secure but not unconditionally

→ Only way to make U.S. cipher is a One-Time-Pad

→ One-Time Pad: A stream cipher s.t.:
1. key stream $s_0, s_1, \dots$ is generated by a TRNG
2. key stream is only known to legit parties
3. Every key stream bit s_i is only used once

Limitations: key length = length of plaintext ⇒ slow ⇒ impractical

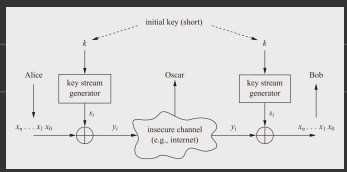
→ All known practical cryptographic algorithms are not U.S.!
- ↳ We aim for Computational Security

Proof of why OTPs are U.S.:

Every ciphertext bit we get:
$y_0 \equiv x_0 + s_0 \bmod 2$
$y_1 \equiv x_1 + s_1 \bmod 2$
...

Impossible to derive unique equations since s_i is truly random

* Practical Stream Ciphers.



→ Computational Security: Cryptosystem is computationally secure if the best known attacking algo requires $\geq t$ operations
↳ Design cryptographic schemes for which it is assumed to be C.S. (For Symmetric: one prays there is no attack. Complexity better than BFA)
↳ Advantage over OTP: Alice & Bob need to only exchange a smaller key where we use PRNGs
sufficient against BFA

Building Key Streams From PRNGs:

→ Ex: Linear Congential Generator: S_0 = seed, $S_{i+1} = AS_i + B \bmod m$, $i \in \mathbb{N}$, $S_i, A, B \in \{0, 1, \ldots, m-1\}$
→ publically known
(A, B) each have length of 100 bits ⇒ 200 bits total

→ Alice encrypts: $y_i \equiv s_i + x_i \bmod 2 \equiv s_i \oplus x_i$
↳ Oscar knows first 300 bits of plaintext and knows cipher text

$S_1 = (s_1, \ldots, s_{100})$
$S_2 = (s_{101}, \ldots, s_{200})$
$S_3 = (s_{201}, \ldots, s_{300})$

⇒ Can figure out first 300 bits of key stream which immediately give first 3 output symbols of PRNG:

Oscar can generate:
$s_2 \equiv As_1 + B \bmod m$
$s_3 \equiv As_2 + B \bmod m$
$\Rightarrow$
$A \equiv (s_2 - s_3)/(s_1 - s_2) \bmod m$
$B \equiv s_2 - s_1 (s_2 - s_3)/(s_1 - s_2) \bmod m$

$\gcd((s_1 - s_2), m) \neq 1$ ⇒ we get multiple sol^n since system is over $\mathbb{Z}_m$

→ However k^{th} piece of plaintext ⇒ key uniquely detected in all cases → .. Few pieces of plaintext ⇒ compute key

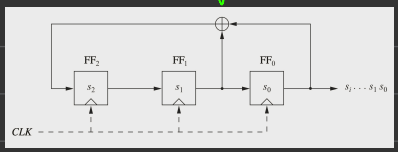
Building Key Streams Using CSPRNGs:

→ CSPRNGs ensures key stream is unpredictable → not practical
→ Most practical stream ciphers use Shift Registers w/ Feedback and some built with pseudorandom f^n based on 32 bit addition, rotation and XOR (add-rotate-XOR operations)

Linear Feedback Shift Registers (LFSRs):

→ Consists of Flip-Flops (clocked storage elements) and a Feedback Path
Feedback Network computes input for last FF as XOR sum of certain FFs in register.
# of storage elements = degree of LFSR

→ Ex: LFSR of degree 3 with initial values: S_2, S_1, S_0



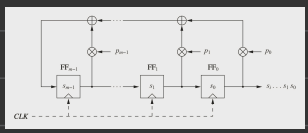
Each clock tick → internal state bits shifted by one to right
Leftmost computed in Feedback Path.

Seq. of states

clk	FF_2	FF_1	$FF_0 = s_i$
0	1	0	0
1	0	1	0
2	1	0	1
3	1	1	0
4	1	1	1
5	0	1	1
6	0	0	1
7	1	0	0
8	0	1	0

output by XOR
↳ input cycle
⇒ period = 7
0010111 0010111 ..

→ In general output bit is computed as: $s_{i+3} \equiv s_{i+1} + s_i \bmod 2$, where $i \in \mathbb{N}$



→ General form of LFSR of degree m has m flip-flops and m feedback path locations, all combined by XOR.
→ Feedback coefficients $(p_{m-1} \ldots p_0, p_1)$ state whether feedback path is active or not.
↳ $p_i \in \begin{cases} 1, \text{if active} \\ 0, \text{if corresponding } S_i \text{ not used} \end{cases}$

Assume LFSR loaded with $s_{m-1}, s_{m-2}, \ldots, s_1, s_0$
Then: $s_m \equiv s_{m-1} p_{m-1} + \cdots + s_1 p_1 + s_0 p_0 \bmod 2$

→ In general output seq described as: $s_{m+i} = \sum_{j=0}^{m-1} p_j s_{i+j} \bmod 2$, $s_i, p_j \in \{0, 1\}$, $i \in \mathbb{N}$
→ Theorem: The maximum seq length generated by an m degree LFSR is $2^m - 1$
→ Length of period = length of LFSR
↳ Proof: State of LFSR uniquely determined by m register bits. Given state, LFSR determines next state. As soon as LFSR assumes prev. state, repeats. Since m-bit vector only assumes $2^m - 1$ nonzero states, max. seq length before repitition is $2^m - 1$

→ Ex

Example 2.4. LFSR with maximum-length output sequence
Given an LFSR of degree $m = 4$ and the feedback path ($p_3 = 0, p_2 = 0, p_1 = 1, p_0 = 1$), the output sequence of the LFSR has a period of $2^m - 1 = 15$, i.e., it is a maximum-length LFSR. ◇

Example 2.5. LFSR with non-maximum output sequence
Given an LFSR of degree $m = 4$ and ($p_3 = 1, p_2 = 1, p_1 = 1, p_0 = 1$), then the output sequence has period of 5; therefore, it is not a maximum-length LFSR. ◇

→ LFSRs specified by polynomials using notation. LFSR w/ feedback coeff vector $\begin{bmatrix} p_{m-1} \\ p_{m-2} \\ \vdots \\ p_0 \end{bmatrix}$ is represented by:
$P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_1 x + p_0$
→ Pros: Maximum length LFSRs have primitive polynomials that are irreducible and can be easily compiled.

→ Linear Dependencies between I/O of LFSRs can be easily analyzed but make highly insecure cipher
→ Known-Plaintext Attack on LFSRs: Key k is feedback coeff vector $(p_{m-1}, \ldots, p_0)$ → Attacks occurs when Oscar knows piece of plaintext & ciphertext (Assume Oscar knows degree m of LFSR)

(0,1,2)	(0,1,3,4,24)	(0,1,2,3,5,8,46)	(0,1,5,7,68)	(0,2,3,5,90)	(0,1,2,3,6,8,112)
(0,1,3)	(0,3,25)	(0,5,47)	(0,2,5,6,69)	(0,1,2,3,5,6,7,91)	(0,2,3,5,113)
(0,1,4)	(0,1,2,6,26)	(0,1,2,4,5,7,48)	(0,1,3,5,70)	(0,2,5,6,92)	(0,2,3,6,7,8,114)
(0,2,5)	(0,1,2,5,27)	(0,4,5,6,49)	(0,1,3,5,71)	(0,2,93)	(0,1,2,3,5,7,115)
(0,1,6)	(0,3,28)	(0,2,3,4,50)	(0,1,2,3,4,6,72)	(0,1,5,6,94)	(0,2,5,6,116)
(0,1,7)	(0,2,29)	(0,1,3,6,51)	(0,2,3,4,73)	(0,1,2,4,5,6,95)	(0,1,2,5,117)
(0,2,3,4,8)	(0,1,4,6,30)	(0,3,52)	(0,1,3,4,7,74)	(0,2,3,4,6,7,96)	(0,2,5,6,118)
(0,4,9)	(0,3,31)	(0,1,2,6,53)	(0,1,3,6,75)	(0,6,97)	(0,8,119)
(0,3,10)	(0,1,2,3,5,7,32)	(0,2,3,4,5,6,54)	(0,2,4,5,76)	(0,1,2,3,4,7,98)	(0,1,2,5,6,7,120)
(0,2,11)	(0,1,4,6,33)	(0,1,2,6,55)	(0,2,5,6,77)	(0,4,5,7,99)	(0,1,5,8,121)
(0,1,4,6,12)	(0,1,2,5,6,7,34)	(0,2,4,7,56)	(0,1,2,7,78)	(0,2,7,8,100)	(0,1,2,6,122)
(0,1,3,4,13)	(0,2,35)	(0,2,3,5,57)	(0,2,3,4,79)	(0,1,6,7,101)	(0,2,123)
(0,1,3,5,14)	(0,1,2,4,5,6,36)	(0,1,5,6,58)	(0,1,2,3,5,7,80)	(0,3,5,6,102)	(0,5,6,7,124)
(0,1,15)	(0,1,2,3,4,5,37)	(0,1,3,4,5,6,59)	(0,4,81)	(0,2,3,4,5,7,103)	(0,1,2,3,5,7,125)
(0,2,3,5,16)	(0,1,5,6,38)	(0,1,60)	(0,1,4,6,7,8,82)	(0,2,3,4,5,6,8,9,104)	(0,2,4,7,126)
(0,3,17)	(0,4,39)	(0,1,2,5,61)	(0,2,4,7,83)	(0,1,2,4,5,6,105)	(0,1,127)
(0,1,2,5,18)	(0,3,4,5,40)	(0,3,5,6,62)	(0,1,3,5,7,8,84)	(0,1,5,6,106)	(0,1,2,7,128)
(0,1,2,5,19)	(0,3,41)	(0,1,63)	(0,1,2,8,85)	(0,1,2,3,5,7,107)	
(0,3,20)	(0,1,2,3,4,5,42)	(0,1,3,4,64)	(0,2,5,6,86)	(0,1,2,3,4,5,6,7,9,10,108)	
(0,2,21)	(0,3,4,6,43)	(0,1,3,4,65)	(0,1,5,7,87)	(0,2,4,5,109)	
(0,1,22)	(0,2,5,6,44)	(0,2,3,5,6,8,66)	(0,1,3,4,5,8,88)	(0,1,4,6,110)	
(0,5,23)	(0,1,3,4,45)	(0,1,2,5,67)	(0,3,5,6,89)	(0,2,4,7,111)	

With 2 pairs of plain & ciphertext bits, Oscar constructs first $2m$ key stream bits: $S_i = x_i \oplus y_i$
let known plaintext be x_0, x_1, x_{2m-1} and ciphertext be $y_0, y_1, \ldots, y_{2m-1}$
→ GOAL: Find key

Major Consequences: As soon as Oscar knows $2m$ output bits of an LFSR of degree m, the p_i coeff. can be exactly constructed by solving a system of linear equations

$s_{m+i} = \sum_{j=0}^{m-1} p_j s_{i+j} \bmod 2$

We get a diff. equation $\forall i \in \mathbb{N}$ that are linearly independent.
Oscar can generate m equations for first m values of i.

$$\begin{array}{llll} i = 0, & s_m & \equiv p_{m-1}s_{m-1} + \ldots + p_1 s_1 + p_0 s_0 & \bmod 2 \\ i = 1, & s_{m+1} & \equiv p_{m-1}s_m + \ldots + p_1 s_2 + p_0 s_1 & \bmod 2 \\ \vdots & \vdots & \vdots & \vdots \\ i = m-1, & s_{2m-1} & \equiv p_{m-1}s_{2m-2} + \ldots + p_1 s_m + p_0 s_{m-1} & \bmod 2 \end{array}$$

Computed Feedback coeff. he can build LFSR and load it w/ any m consecutive output bits that he already knows

Can be solved using Gaussian Elimination, Matrix Inversion

Clock LFSR and produce entire output seq.

Salsa20

→ Salsa20: Family of sub-efficient stream ciphers using a Pseudorandom f^n based on add-rotate-XOR operations (can only broken by BFA; has 20 rounds)

↳ supports key lengths of 128 and 256 bits

→ Core of Salsa20: F^n with 512 bit input and output ⇒ For e(k) and d(y), processes a key, a nonce, and a 64-bit block # and generates 512 bit block of key stream

↳ result is XOR added w/ text (cipher or plain) → Since Salsa20 generates 512 output bits, one can encrypt/decrypt 512 bits at once

→ Main Purpose: 2 key streams produced by cipher should be diff even if key was not changed → otherwise, attacker has known plaintext from first encryption can lead to compromise of key stream same with 2nd encryption

→ Encryption and Decryption is as follows: Let k be a 256 bit seq Let n be 8 byte nonce

Let x be an I-byte message for $I \in \{0, 1, \ldots, 2^{70}\}$

↳ Encryption of msg x yields I-byte ciphertext y: $y = \text{Salsa20}_k(n) \oplus x$

↳ Decryption: $x = \text{Salsa20}_k(n) \oplus y$ → Since each block only depends on key, nonce & block #, key stream blocks computed independently & in parallel → Faster speed

W/o changing nonce, encryption is highly deterministic → used for initialization vectors, ensures same key is reused w/o compromising security by guaranteeing uniqueness. Also prevents attackers from reusing prev. ciphertexts

→ 512-bit internal state of Salsa20 consists of 17 32-bit words y_i, arranged in 4×4 matrix.

u_0	u_1	u_2	u_3
u_4	u_5	u_6	u_7
u_8	u_9	u_{10}	u_{11}
u_{12}	u_{13}	u_{14}	u_{15}

↓

c_0	k_0	k_1	k_2
k_3	c_1	n_0	n_1
p_0	p_1	c_2	k_4
k_5	k_6	k_7	c_3

→ To start encryption process, initial state of Salsa20 is constructed as follows:

1. 8 32-bit words formed by key $K = [k_0, k_1, \ldots, k_7]$, 2 words indicate stream position $[p_0, p_1]$, 2 words come from nonce $n = [n_0, n_1]$ and 4 words are a constant $C = [c_0, c_1, c_2, c_3]$ → "expand 32-byte K" → if 128 bit key of 4 words $K = [k_0, k_1, k_2, k_3]$ used ⇒ it would be of form $[k_0, k_1, k_2, k_3, k_0, k_1, k_2, k_3]$ (Counter of current 512 bit block within range of all 2^{64} 512-bit blocks of keystream.)
2. QR(a,b,c,d) (Quarter Round F^n) is the core operation and is as follows:

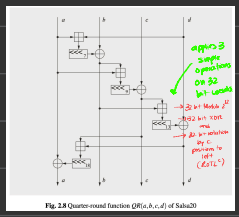


Fig. 2.8 Quarter-round function QR(a, b, c, d) of Salsa20

↳ 4 word output computed from 4 bit input as follows

$b = b \oplus \text{ROTL}^7(a+d)$

$c = c \oplus \text{ROTL}^9(b+a)$

$d = d \oplus \text{ROTL}^{13}(c+b)$

$a = a \oplus \text{ROTL}^{18}(d+c)$

→ Double Round: 1 QR of a round + 2 consecutive rounds

↳ ODD # Round: QR applied to each 4 columns

↳ EVEN # Round: QR applied to each 4 rows

Ex:

First odd Round:

$(u_0, u_4, u_8, u_{12}) = QR(v_0, v_4, v_8, v_{12})$

$(u_5, u_9, u_{13}, u_1) = QR(v_5, v_9, v_{13}, v_1)$

$(u_{10}, u_{14}, u_2, u_6) = QR(v_{10}, v_{14}, v_2, v_6)$

$(u_{15}, u_3, u_7, u_{11}) = QR(v_{15}, v_3, v_7, v_{11})$

2nd Round

$(z_0, z_1, z_2, z_3) = QR(u_0, u_1, u_2, u_3)$

$(z_5, z_6, z_7, z_4) = QR(u_5, u_6, u_7, u_4)$

$(z_{10}, z_{11}, z_8, z_9) = QR(u_{10}, u_{11}, u_8, u_9)$

$(z_{15}, z_{12}, z_{13}, z_{14}) = QR(u_{15}, u_{12}, u_{13}, u_{14})$

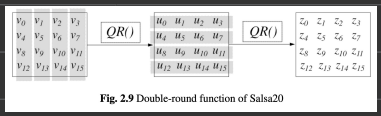


Fig. 2.9 Double-round function of Salsa20

→ Implementation: Salsa20 is an ARX cipher (internally only does additions, XOR, rotation) → With 20 rounds, requires ≈ 4-14 cycles / byte on average for long key streams

ChaCha

→ Follows same design basic principles as Salsa20 (8/12/20 rounds)

→ Encryption & Decryption: ChaCha generates 512 bit block from its inputs (key, nonce, block no)

↳ Hands ⊕ 512 bit text for encryption/decryption

↳ Each 512 bit key stream block computed independently & in parallel

→ Core Function of ChaCha20: Initial state includes 128 bit constant c, 256 bit key k, 64 bit counter p and a 64-bit nonce n → 4×4 matrix of 32 bit words

↳ Constant c given by ASCII version "expand 32-byte K"

Let K be 32 byte/16-byte seq Let n be 8 byte nonce

Let x be an I byte msg for some $I \in \{1, \ldots, 2^{70}\}$

$y = \text{ChaCha20}_k(n) \oplus x$ → Encryption

$x = \text{ChaCha20}_k(n) \oplus y$ → Decryption

c_0	c_1	c_2	c_3
k_0	k_1	k_2	k_3
k_4	k_5	k_6	k_7
p_0	p_1	n_0	n_1

(different order of values)

→ ChaCha20 uses quarter round f^n QR(a,b,c,d) on its 32 bit values a,b,c,d → applies 4 additions mod 2^{32}, 4 XORs and 4 rotations repeatedly on the state words in a diff. order.

$a = a + b$

$d = \text{ROTL}^{16}(d \oplus a)$

$c = c + d$

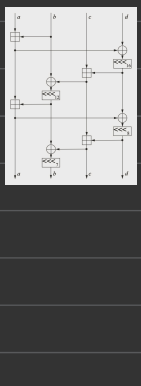
$b = \text{ROTL}^{12}(b \oplus c)$

$a = a + b$

$d = \text{ROTL}^{8}(d \oplus a)$

$c = c + d$

$b = \text{ROTL}^{7}(b \oplus c)$



→ Performs 10 iterations of a double round

Denote input by $(v_0, v_1, \ldots, v_5)$ and output $(v_0, v_1, \ldots, v_5)$ of first odd round of double-round computed on columns.

$(u_0, u_4, u_8, u_{12}) = QR(v_0, v_4, v_8, v_{12}),$

$(u_1, u_5, u_9, u_{13}) = QR(v_1, v_5, v_9, v_{13}),$

$(u_2, u_6, u_{10}, u_{14}) = QR(v_2, v_6, v_{10}, v_{14}),$

$(u_3, u_7, u_{11}, u_{15}) = QR(v_3, v_7, v_{11}, v_{15}).$

→ 2nd round of double-round yields intermediate output $(z_0, \ldots, z_5)$

$(z_0, z_5, z_{10}, z_{15}) = QR(v_0, v_5, v_{10}, v_{15}),$

$(z_1, z_6, z_{11}, z_{12}) = QR(v_1, v_6, v_{11}, v_{12}),$

$(z_2, z_7, z_8, z_{13}) = QR(v_2, v_7, v_8, v_{13}),$

$(z_3, z_4, z_9, z_{14}) = QR(v_3, v_4, v_9, v_{14}).$