Name: Madhav Verma
Subject: Cryptographic & Security Implementations

Question 01 — Gate Count/Hardware requirement Estimation for Grain v1 Stream Cipher

Registers and Storage:
Grain v1 has an 80-bit LFSR and 80-bit NFSR. Using the estimate of 2.5 gates per flip-flop:

- LFSR: $80 \times 2.5 = 200$ gates
- NFSR: $80 \times 2.5 = 200$ gates
- **Total storage: 400 gates**

LFSR Feedback Function:

$$s_{i+80} = s_{62} \oplus s_{51} \oplus s_{38} \oplus s_{23} \oplus (s_{13} \& s_i)$$

This consists of:

- 4 XOR taps combined in a 3-level XOR tree: $3 \times 2 = 6$ gates
- 1 two-input AND: 1 gate
- 1 XOR with the AND output: $1 \times 2 = 2$ gates

Total: $6 + 1 + 2 = \boxed{9 \text{ gates}}$

NFSR Feedback Function:

$$\begin{aligned} b_{i+80} = {} & s_i \oplus b_i \oplus b_{62} \oplus b_{60} \oplus b_{52} \oplus b_{45} \oplus b_{37} \\ & \oplus b_{33} \oplus b_{28} \oplus b_9 \\ & \oplus b_{63}b_{60} \oplus b_{37}b_{33} \oplus b_{15}b_9 \\ & \oplus b_{60}b_{52}b_{45} \oplus b_{33}b_{28}b_{21} \oplus b_{63}b_{45}b_{28}b_9 \\ & \oplus b_{60}b_{52}b_{37}b_{33} \oplus b_{63}b_{60}b_{52}b_{45}b_{37} \\ & \oplus b_{33}b_{28}b_{21}b_{15}b_9 \\ & \oplus b_{52}b_{45}b_{37}b_{33}b_{28}b_{21} \end{aligned}$$

Gate Breakdown:

- **Linear XOR taps:** 11 variables $\Rightarrow$ 10 XOR terms $\Rightarrow$ $10 \times 2 = 20$ XOR gates
- **AND gates:**
  - 3 terms with 2-input ANDs: $3 \times 1 = 3$ gates
  - 2 terms with 3-input ANDs: $2 \times 2 = 4$ gates
  - 2 terms with 4-input ANDs: $2 \times 3 = 6$ gates

  - 2 terms with 5-input ANDs: $2 \times 4 = 8$ gates
  - 1 term with 6-input AND: $1 \times 5 = 5$ gates
  - **Total AND gates:** $3 + 4 + 6 + 8 + 5 = 26$

- **XORs to combine nonlinear terms:** 8 products $\Rightarrow 7 \times 2 = 14$ XOR gates
- **Total gates for NFSR feedback:** 20 (linear XORs) +26 (ANDs) +14 (nonlinear XORs) $= \boxed{\text{60 gates}}$

Output Function z_i:

$$\begin{aligned} z_i = b_1 \oplus b_2 \oplus b_4 \oplus b_{10} \oplus b_{31} \oplus b_{43} \oplus b_{56} \\ \oplus h(s_3, s_{25}, s_{46}, s_{64}, b_{63}) \end{aligned}$$

$$\begin{aligned} h(x_0, x_1, x_2, x_3, x_4) &= x_1 \oplus x_4 \oplus x_0x_3 \oplus x_2x_3 \oplus x_3x_4 \\ &\oplus x_0x_1x_2 \oplus x_0x_2x_3 \oplus x_0x_2x_4 \\ &\oplus x_1x_2x_4 \oplus x_2x_3x_4 \end{aligned}$$

Gate Breakdown:

- 7-input XOR for b terms: 6 XORs = 12 gates
- $h(x)$ function:
  - 2 XORs = 4 gates
  - 3×2-input ANDs = 3 gates
  - 4×3-input ANDs = 8 gates
  - 1×4-input AND = 3 gates
  - 9 XORs to combine = 18 gates
- **Total output logic: 48 gates**

KSA Extra XORs: Injecting z_i into LFSR and NFSR:

- 2 XORs = $2 \times 2 = 4$ gates

Final Gate Count Summary:

- KSA round total: $400 + 10 + 58 + 48 + 4 =$ **520 gates**
- PRGA round total: $400 + 10 + 58 + 48 =$ **516 gates**

Question 02 - Can we say multiplication and division have the same time complexity for n bit integers?

Time Complexity Comparison: Multiplication vs Division

We analyze the time complexity of multiplication and division for n-bit integers, both from theoretical and practical standpoints.

Multiplication

- **Grade-School Algorithm:** Performs $n \times n = O(n^2)$ single-bit multiplications.
- **Karatsuba Algorithm:** A divide-and-conquer approach that reduces the number of recursive multiplications.

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

- **Fastest Known:**
  - **Schönhage-Strassen algorithm:** $O(n \log n \log \log n)$ using FFT in rings.
  - **Fürer's algorithm:** Further improves to $O(n \log n \cdot 2^{O(\log^* n)})$.

Division

- **Long Division Algorithm:** Similar to manual division, with complexity $O(n^2)$.
- **Newton-Raphson Division:**
  - Based on computing the reciprocal using Newton-Raphson iterations:

$$x_{k+1} = x_k(2 - bx_k)$$

    where b is the divisor.
  - Once $\frac{1}{b}$ is approximated, compute $a/b \approx a \cdot (1/b)$.
  - Each iteration roughly doubles precision, and multiplication dominates cost.
  - Total complexity: $O(M(n))$, where $M(n)$ is multiplication time.

Asymptotic Summary

Operation	Naïve Complexity	Best Known Complexity
Multiplication	$O(n^2)$	$O(n \log n)$ (FFT-based)
Division	$O(n^2)$	$O(n \log n)$ (via Newton-Raphson)

Practical Considerations

- Despite similar asymptotic bounds, **division is significantly slower** than multiplication on most hardware.
- On typical CPUs:
  - Multiplication is pipelined and parallelizable.
  - Division involves iterative estimation and correction, requiring more cycles.
- As an example, on x86 CPUs, an integer division may take 20–40 cycles, while a multiplication takes 1–3 cycles.

Answer

While both multiplication and division can be performed in $O(n \log n)$ time using advanced algorithms, division is generally:

- Harder to implement,
- Slower in practice,
- More complex in both hardware and software.

Hence, **multiplication and division do not have the same time complexity in practice**.

Question 01 — Which function is used to compute the clock cycle of a program. Which function call is used here.

Answer: To measure the number of CPU clock cycles consumed by a program, different system calls and intrinsic functions are used depending on the platform.

- **x86 (Linux/Windows):** Use the Time Stamp Counter instruction via `__rdtsc()` intrinsic. It returns the number of cycles since reset.
- **POSIX systems:** Use `clock_gettime(CLOCK_MONOTONIC_RAW, ...)` which gives nanosecond resolution. Multiplying by CPU frequency converts time into cycles.
- **macOS (Apple Silicon):** Use `mach_absolute_time()` which returns raw ticks. Convert to nanoseconds using `mach_timebase_info`.

Which function call is used here: In your provided code setup on macOS, the call used is `clock_gettime(CLOCK_MONOTONIC_RAW, ...)` (or alternatively `mach_absolute_time()` for M1). On x86 machines, `__rdtsc()` would be the preferred function.

Question 02 — RC4 Configuration, KSA/PRGA and Bias in KSA

RC4 configuration:

- **Key size:** Between 5 and 30 bytes (40 to 240 bits).
- **State size:** 256-byte permutation array `S[256]`.
- **Indices:** Two 8-bit indices `i`, `j`.
- **Variables in C:** Typically `unsigned char` or `uint8_t`.
- **Data structure:** Array of 256 bytes storing the permutation, plus index variables.

Key Scheduling Algorithm (KSA):

```
for $i = 0 \ldots 255 : S[i] = i$
    $j = 0$
for $i = 0 \ldots 255 : j = (j + S[i] + K[i]) \bmod 256; \quad \text{swap}(S[i], S[j])$
```

Pseudo-Random Generation Algorithm (PRGA):

$$\begin{aligned} i &= (i+1) \bmod 256 \\ j &= (j + S[i]) \bmod 256 \\ &\text{swap}(S[i], S[j]) \\ Z &= S[(S[i] + S[j]) \bmod 256] \end{aligned}$$

Bias of permutation in KSA (from RC25 slides):

- The KSA does not generate all 256! permutations. For a key of length l, at most 2^{8l} permutations are possible, which is only a tiny subset of all permutations.
- Example: After the first swap, $S[0] = K[0]$ with probability ≈ 0.37, since $S[0]$ may only be touched again with probability $1/256$ per round, and the chance of it remaining untouched for the rest is $(255/256)^{255}$.
- The distribution $P(S[u] = v)$ is biased compared to the ideal uniform $1/256$. Mantin (2001) provided exact formulas showing this non-randomness.
- Empirical results reveal *anomaly pairs* (u, v) where theory and experiment diverge. Their frequency reduces as key length increases; with a 256-byte key no anomaly pairs occur.
- Roos (1995) showed that the most likely value of $S[y]$ after the KSA is

  $$f_y = \frac{y(y+1)}{2} + \sum_{x=0}^{y} K[x],$$

  which shows direct bias toward cumulative key sums.
- These permutation biases propagate into the PRGA, leading to well-known keystream biases such as the Mantin–Shamir $Z_2 = 0$ bias.

Conclusion: RC4 uses a 256-byte state with a simple KSA and PRGA. However, the KSA introduces significant non-randomness, creating biases that leak information about the key and enable practical attacks.

Question 03 — Good Algorithm to Generate a Random Permutation. Can it replace RC4 KSA?

Answer: A well-known method for generating a random permutation is the **Fisher–Yates shuffle**:

$$\text{for } i = N-1 \text{ down to } 1: \quad j = \text{rand}(0, i);\ \text{swap}(S[i], S[j]).$$

- If the random numbers are uniform, every permutation is equally likely.
- The algorithm runs in $O(N)$ time and produces a uniform distribution over all $N!$ permutations.

Why not replace RC4 KSA with this:

- The Fisher–Yates shuffle requires high-quality uniform random numbers. If these come directly from the secret key, the transpositions themselves leak information about the key.
- RC4's KSA is a deterministic mixing function that derives the permutation directly from the key. It is biased, but those biases are understood and can be analyzed.
- Using Fisher–Yates with key-derived randomness would not provide security unless combined with a cryptographically strong pseudorandom function, which effectively replaces RC4 altogether.

Conclusion: Fisher–Yates is a good algorithm for random permutation generation, but it cannot safely replace RC4's KSA because it would reveal structure about the key and compromise security.

Question 04 — Sorting Algorithms: Bubble, Merge, Heap, Quick Sort

Bubble Sort:

- Pseudocode: Repeatedly sweep the array and swap adjacent out-of-order elements.
- Recurrence: $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.
- Comparisons: $\Theta(n^2)$.
- Swaps: Up to $\Theta(n^2)$.

Merge Sort:

- Pseudocode: Divide into halves, recursively sort, then merge.
- Recurrence: $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$.
- Comparisons: At most $n\lceil \log_2 n \rceil - n + 1$.
- Swaps: None in pure merge, but moves are $\Theta(n \log n)$.

Heap Sort:

- Pseudocode: Build a heap, repeatedly extract max, re-heapify.
- Recurrence: $T(n) = \Theta(n) + n \cdot \Theta(\log n) = \Theta(n \log n)$.
- Comparisons: $\Theta(n \log n)$.
- Swaps: $\Theta(n \log n)$.

Quick Sort:

- Pseudocode: Partition around pivot, recurse on subarrays.
- Recurrence (expected): $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.
- Worst case: $T(n) = \Theta(n^2)$ if pivot choice is poor.
- Comparisons: $\Theta(n \log n)$ average.

- Swaps: $\Theta(n \log n)$ average, $\Theta(n^2)$ worst case.

Who does the most comparisons and swaps: Among these, **Bubble Sort** performs the most comparisons and swaps in practice: $\Theta(n^2)$ each. Merge, heap, and quick sort (average case) are $\Theta(n \log n)$ and much more efficient.

Question 05 — Study Triple DES. Why 2DES is not secure but 3DES is.

DES recap:

- Block size: 64 bits. Key size: 56 effective bits (64 with 8 parity bits).
- A DES encryption is denoted $E_K(\cdot)$ and decryption $D_K(\cdot)$.

2DES construction and attack:

$$C = E_{K_2}(E_{K_1}(P)) \quad \text{with} \quad K_1, K_2 \in \{0,1\}^{56}.$$

Naively this suggests about 2^{112} security. However, a *meet-in-the-middle* (MITM) attack breaks 2DES with roughly 2^{57} time and 2^{56} memory using just one known (P, C) pair.

1. For all 2^{56} keys K_1, compute and store $X = E_{K_1}(P)$ in a table keyed by X along with K_1.
2. For all 2^{56} keys K_2, compute $Y = D_{K_2}(C)$ and look up Y in the table. Any hit yields a candidate pair (K_1, K_2).
3. Verify each candidate on a second (P', C') to eliminate false positives.

Complexity: about 2^{56} encryptions for the forward pass, 2^{56} decryptions for the backward pass, table of size 2^{56}. This completely defeats the intended 2^{112} security of 2DES.

3DES constructions:

- **EDE mode (most common):**

$$C = E_{K_3}\big(D_{K_2}(E_{K_1}(P))\big), \quad P = D_{K_1}\big(E_{K_2}(D_{K_3}(C))\big).$$

- **Keying options:**
  - Three-key 3DES: (K_1, K_2, K_3) all independent. Nominal key length 168 bits.
  - Two-key 3DES: $K_1 = K_3 \neq K_2$. Nominal key length 112 bits. Still widely analyzed.

Why 3DES resists simple MITM:

- A direct MITM between the outer E_{K_1} and inner D_{K_2} is obstructed by the additional third stage and the need to match through two layers. The best generic attacks reduce to roughly 2^{112} time for two-key 3DES under standard models, which is far higher than 2DES.
- Known advanced attacks (e.g., van Oorschot–Wiener type) can trade time, memory, and many known pairs, but security remains around 2^{112} for two-key 3DES in classical settings.

Practical security notes:

- **Block size limits:** DES uses 64-bit blocks, so large volumes can hit birthday-bound issues (e.g., "Sweet32"), making 3DES unsuitable for very high data volumes per key.
- **Mode of operation matters:** Use authenticated modes and rekey before large data thresholds to avoid block-collision risks.

Conclusion: 2DES is insecure due to the classic meet-in-the-middle attack requiring about 2^{57} effort. Triple DES avoids this simple MITM and, in the two-key variant, offers about 112-bit security in classical models, though the 64-bit block size makes it legacy for new designs.

Question 06 — From rc25.pdf pages 16,17,18,22,23,25,32,34,35,39,40,41,45,49,50,51,52,58,62: what exam questions can be made?

Answer: Below is a curated set of exam-style questions derived from those pages. They target RC4's data structures, KSA/PRGA, and the families of biases and attacks discussed in the slides.

- **RC4 data structure and initialization**
  - Define the RC4 state. Explain the roles of $S[256]$, the indices i, j, and the expanded key $K[i]$. Show how $K[i]$ is derived from a key of length l.
  - Prove that the number of permutations reachable by RC4's KSA from an l-byte key is at most 2^{8l} and compare this to 256!. Why does this imply the KSA explores only a tiny subset of all permutations?
- **KSA algorithm and non-randomness**
  - Write the KSA pseudocode. Starting from $S[i] = i$ and $j = 0$, compute the first two swaps for a small toy key to illustrate how key bytes influence early locations.
  - Derive or state the formula for $P(S[u] = v)$ immediately after KSA and explain how it differs from the ideal 1/256. What does this say about the non-randomness of the permutation?
  - Explain the concept of *anomaly pairs* where theoretical $P(S[u] = v)$ and experimental values diverge. How does key length affect the number of anomaly pairs?
  - Using the "$S[0]$ sticks" argument, show why $S[0] = K[0]$ with probability approximately $(255/256)^{255} \approx 0.37$ after KSA. State the assumptions behind this approximation.
  - State Roos's formula for the most likely value of $S[y]$ after KSA, $f_y = \frac{y(y+1)}{2} + \sum_{x=0}^{y} K[x]$. Discuss its meaning and limitations.
  - Present the **movement-frequency** results: probability a value v is touched exactly once during KSA, and the expected number of touches E_v. Explain why these help in reversing parts of KSA.
  - Define and illustrate **key collisions** in KSA. Give an example pattern where two distinct keys yield the same permutation. Why is KSA not one-to-one?
  - Outline the **KSA-reversal** idea: given the post-KSA permutation, set up equations of the form $S_N[y] = f_y$ for small y to solve for key bytes. Why is the system "triangular"? What checks are needed to validate a candidate key?

- **PRGA structure and early-round behavior**
  - Write the PRGA pseudocode. Explain why j is "pseudo-random" and show how $j_1 = S[1]$ becomes biased if $S[1]$ is biased from KSA.
  - Discuss the **non-randomness of** j for the first few rounds. Give the distribution for j_1, j_2, and why for $r \geq 3$ the biases shrink.
  - Explain the absence of short cycles and describe the *Finney cycle* thought experiment and why it cannot happen in standard RC4 PRGA initialization.
- **Classical keystream biases and attacks**
  - Prove or outline Mantin–Shamir's second-byte bias $P(Z_2 = 0) \approx 2/256$. Show how this yields a practical distinguisher and the well-known broadcast attack for recovering the second plaintext byte across many encryptions.
  - Explain the negative bias toward 0 in Z_1 and the negative bias toward 2 in Z_2. Give the approximate probabilities and discuss the intuition.
  - State the improved bounds for $P(Z_r = 0)$ for $3 \leq r \leq 255$ and explain how they correct the earlier belief that only Z_2 is biased.
  - Define the **digraph repetition** bias and why patterns of the form $ABTAB$ occur more than in random streams. Estimate how many samples are needed for a distinguisher based on this phenomenon.
  - Present the **Glimpse Theorem** and its corollary for z_r. Explain how it links PRGA outputs to entries of the previous permutation and why this creates predictable structure.
  - Describe the late-byte biases at indices 256 and 257. Give the approximate probabilities and explain why such biases persist even far from the start.
- **Biases tied to key length**
  - Describe the empirical observation that Z_ℓ near $-\ell \pmod{256}$ is more likely when the key length is ℓ bytes. Explain the conditional events such as $S_\ell[j_\ell] = 0$ and how they amplify bias.
  - Explain how one might *predict key length* by measuring the distribution of Z_ℓ events over many traces. What are the practical limitations?
- **Synthesis and defenses**
  - Given the collection of biases, propose a strategy for key recovery in WEP/WPA-like settings that rely on many sessions. Which bytes and rounds would you focus on and why?
  - Suggest mitigations when RC4 must be used: key-whitening, discarding a prefix of keystream, or abandoning RC4 entirely. Discuss pros and cons.

Outcome: These questions cover the specific slide topics you listed and are phrased for exam preparation, from basic definitions to derivations, proofs, and applications.

Question 06 — Answers to the exam-style prompts from the listed RC4 slides

RC4 data structure and initialization

- **State definition:** RC4 maintains a permutation array $S[0..255]$ of bytes and two byte indices i, j. The output each round is a keystream byte Z computed from $S[i], S[j]$.
- **Expanded key $K[i]$:** For a key $k[0..l-1]$ of length l bytes, define $K[i] = k[i \bmod l]$ for $i = 0..255$. K is used only during KSA.
- **Why at most 2^{8l} permutations:** The KSA is fully determined by the sequence $K[0..255]$. Since there are 2^{8l} possible l-byte keys and KSA is deterministic, the number of distinct permutations reachable is $\leq 2^{8l}$, which is negligible compared to $256!$. Hence KSA explores a tiny subset of all permutations.

KSA algorithm and non-randomness

- **KSA pseudocode:** Initialize $S[i] = i$ and $j = 0$. For $i = 0..255$: $j \leftarrow (j + S[i] + K[i]) \bmod 256$; swap $S[i] \leftrightarrow S[j]$.
- **Toy trace for first two swaps:** With $S[0] = 0$, $j = 0$, $K[0] = k_0$, we get $j = k_0$ and swap $S[0]$ with $S[k_0]$. Next, $i = 1$: $j = (k_0 + S[1] + K[1]) \bmod 256 = (k_0 + 1 + k_1) \bmod 256$ and swap $S[1]$ with $S[j]$. This shows early array locations are directly influenced by small partial sums of key bytes.
- **Non-uniform $P(S[u] = v)$:** Ideally $P(S[u] = v) = 1/256$. After KSA, the distribution deviates due to the deterministic j-update and dependence on cumulative key sums; early positions exhibit strong non-uniformity.
- **Anomaly pairs:** Certain (u, v) pairs show experimental probabilities that noticeably differ from theoretical approximations. The number of such pairs tends to decrease as the key length l increases; for very large l the effect fades.
- **$S[0]$ "sticks" argument:** After the first step, $S[0] = K[0]$ (since $j = K[0]$ and $S[0] \leftrightarrow S[K[0]]$). In subsequent rounds, the only way to move $S[0]$ again is if $j = 0$, which happens with probability 1/256 per round. The probability that $S[0]$ remains untouched for the remaining 255 rounds is $(255/256)^{255} \approx 0.37$, hence $S[0] = K[0]$ with sizable probability.
- **Roos's f_y:** For small y, the most likely value of $S[y]$ is $f_y = \frac{y(y+1)}{2} + \sum_{x=0}^{y} K[x] \pmod{256}$. Intuition: early $S[y]$ tends to follow the cumulative drift of index j driven by partial sums of the key. Limitation: it is a most-likely value, not a guarantee, and accuracy lowers as y grows.
- **Movement frequency:** Let "touched" mean either $i = y$ (always true once) or $j = y$. A value v in the permutation is touched a random number of times; one can show the probability of exactly one touch and the expected count E_v depend on v and on $(255/256)^{\cdot}$ factors. Practical use: bytes touched unusually often or rarely give equations helping partial KSA reversal.
- **Key collisions:** KSA is not injective. Different keys can lead to the same final permutation, hence the same keystream. Collisions arise because many distinct K sequences fold to an identical sequence of swaps.
- **KSA reversal overview:** For small y, use $S_N[y] \approx f_y$ to form linear equations in the unknown key bytes $K[0], K[1], \ldots$. Because f_y uses cumulative sums, the system is triangular, enabling back-substitution. Check a candidate key by re-running KSA and comparing the full S; reject if mismatch occurs.

PRGA structure and early-round behavior

- **PRGA pseudocode:** For each output: $i \leftarrow i+1$; $j \leftarrow j+S[i]$; swap $S[i] \leftrightarrow S[j]$; output $Z = S[(S[i]+S[j]) \bmod 256]$.
- **Why j is "pseudo-random":** j is an accumulating index of $S[i]$ values. If S were uniform and independent, j would look uniform. However, KSA biases S initially, so $j_1 = S[1]$ inherits that bias, and j_2 still shows remnants; for $r \geq 3$ the effect decays as swaps mix the permutation.
- **No short cycles; Finney cycle cannot occur:** A hypothetical condition $j = i+1$ and $S[j] = 1$ is preserved and would form a long cycle of length $256 \cdot 255$, but the PRGA starts from $(i,j) = (0,0)$ and the initialization makes this condition unattainable in standard RC4, so the "Finney cycle" cannot happen.

Classical keystream biases and attacks

- **Mantin–Shamir $Z_2 = 0$ bias:** In a random source $P(Z_2 = 0) = 1/256$. In RC4, structural dependencies in the second PRGA step imply $P(Z_2 = 0) \approx 2/256$. Consequences: a strong distinguisher with $O(256)$ samples and a broadcast attack that recovers the second plaintext byte by majority on many encryptions of different messages under different nonces/keys.
- **Negative biases:** Z_1 is slightly less likely to be 0 than $1/256$; Z_2 is slightly less likely to be 2 than $1/256$ (a compensating effect because $Z_2 = 0$ is higher). These come from combinational constraints on (i,j,S) during the first few PRGA steps.
- **Later Z_r bounds ($3 \leq r \leq 255$):** $P(Z_r = 0)$ is not exactly $1/256$; it can be expressed as $1/256 + c_r/256^2$ for some small c_r that decays with r, correcting the old belief that only Z_2 is biased.
- **Digraph repetition bias ($ABTAB$):** Pairs of outputs with a small gap Δ reappear slightly more often than random. This long-term dependency enables distinguishers with on the order of 2^{26}–2^{29} samples depending on the success probability target.
- **Glimpse-style relation:** Events like $P(S_r[j_r] = i_r - z_r) \approx 2/256$ or equivalently $P(z_r = r - S_{r-1}[r]) \approx 2/256$ link outputs to entries of the previous permutation, creating weak but exploitable structure across rounds.
- **Late-byte biases ($r = 256, 257$):** Even far from the start, structure persists. For example, $P(Z_{256} = \text{specific value})$ and $P(Z_{257} = \text{specific value})$ are measurably above $1/256$ by ≈ 0.0002–0.0006 margins, showing that tiny biases survive deep into the stream.

Biases tied to key length

- **Key-length marker around $r = \ell$:** When the key has length ℓ bytes, events around round $r = \ell$ exhibit slightly elevated probabilities, such as $Z_\ell \equiv -\ell \pmod{256}$ occurring more often than $1/256$, especially conditioned on events like $S_\ell[j_\ell] = 0$. Intuition: the KSA's use of $K[i]$ with period ℓ leaves a fingerprint at index ℓ.
- **Predicting key length:** Over many sessions, tally the frequency of $Z_r \equiv -r \pmod{256}$ for r in $[5, 32]$. The r with the largest excess over $1/256$ suggests the key length. Practical limitations include noise, availability of many traces, and variations in implementations.

Synthesis and defenses

- **Key-recovery strategy (many sessions):** Focus on early bytes Z_1, Z_2 and the conditional relations linking z_r to $S_{r-1}[r]$ and to j_r, i_r. Use the $Z_2 = 0$ bias for immediate distinguishing and second-byte inference. Aggregate statistics across many encryptions to infer key bytes using equations derived from early-round biases and KSA reversal heuristics.
- **Mitigations if RC4 must be used:** Discard an initial prefix of the keystream (RC4-dropN), combine with key whitening, and avoid using early bytes in protocols. Best practice in modern systems is to *avoid RC4 entirely* and use authenticated encryption (e.g., AES-GCM or ChaCha20-Poly1305) with strict nonce/IV rules.

Takeaway: RC4's simplicity enables deep analytical results. KSA yields non-uniform permutations tightly tied to cumulative key sums; these imprints propagate into PRGA, creating a family of biases (early and late) that support distinguishers, broadcasts, and even partial key-recovery when many sessions are available.

Question 07 — In ChaCha why do we use $X + (X \lll r)$ instead of just $(X \lll r)$? How do we store the state of ChaCha, how many bits are required, and which data structure do we use.

Why $X + (X \lll r)$ and not only rotation:

- Rotation and XOR are linear over $\mathbb{F}_2$. Linear operations preserve linear relations and make linear and differential attacks easier. A pure rotation $(X \lll r)$ does not create carries. It is linear in the bitwise sense and therefore relatively weak for diffusion alone.
- Modular addition $+$ over 2^{32} is *nonlinear* over $\mathbb{F}_2$ because of carry propagation. When you compute $X + (X \lll r)$, the carry chain mixes distant bits in a data dependent way. This breaks linear relations, improves diffusion speed, and increases resistance to linear and differential cryptanalysis.
- ChaCha uses the ARX design pattern: Addition, Rotation, XOR. Addition injects nonlinearity, rotation repositions bits without loss, XOR blends words. The trio amplifies avalanche in a few rounds while remaining very fast in software.
- Rotational cryptanalysis relies on pure rotational symmetries. Adding a rotated copy of a word destroys simple rotational invariants because of carries, which helps against rotational and slide style attacks.

ChaCha quarter round (for context): Let (a, b, c, d) be four 32 bit words, and $\lll r$ denote left rotation by r bits.

$$\begin{aligned}
a &= a + b, \quad & d &= d \oplus a, \quad & d &= d \lll 16,\\
c &= c + d, \quad & b &= b \oplus c, \quad & b &= b \lll 12,\\
a &= a + b, \quad & d &= d \oplus a, \quad & d &= d \lll 8,\\
c &= c + d, \quad & b &= b \oplus c, \quad & b &= b \lll 7.
\end{aligned}$$

The constants $16, 12, 8, 7$ are chosen to quickly mix bits across positions without introducing simple symmetries.

ChaCha state layout, size, and data structure:

- The ChaCha state is a 4×4 matrix of 32 bit words. Total words = 16. Total state size $= 16 \times 32 = 512$ bits.
- A common C representation uses `uint32_t state[16]`. The algorithm is specified in little endian.
- Initial state words for ChaCha20 (IETF variant):
  - Constants: 4 words spelling the ASCII string "expand 32 byte k".
  - Key: 8 words for a 256 bit key.
  - Counter: 1 word for a 32 bit block counter.
  - Nonce: 3 words for a 96 bit nonce.
- Original ChaCha allowed a 64 bit nonce and 64 bit counter. The IETF profile standardizes a 96 bit nonce and 32 bit counter to align with AEAD constructions like ChaCha20 Poly1305.
- One ChaCha20 block generates 64 bytes of keystream. After 20 rounds (10 column rounds and 10 diagonal rounds), the state is added wordwise to the original state and serialized as output.

Summary: Using $X + (X \lll r)$ introduces carry driven nonlinearity that pure rotation lacks. The state is 16 words of 32 bits each stored as a flat array or a 4×4 matrix. ChaCha20 typically uses a 256 bit key, a 96 bit nonce, and a 32 bit counter.

Question 08 — Why is an IV used when we already have a key. Can we just use only the key.

Purpose of an IV or nonce:

- **Semantic security** requires that encrypting the same message twice under the same key yields different ciphertexts. An IV or nonce achieves this by varying the initialization. Without it, identical plaintexts would map to identical ciphertexts which leaks information.
- In stream ciphers and counter based modes, the IV or nonce selects a fresh keystream for each message. Reusing the same keystream with different plaintexts is a two time pad and immediately leaks the XOR of plaintexts.
- In block cipher modes, the IV prevents structural leakage. For example, CBC with a fixed IV leaks whether the first block repeats. In CTR or GCM, a unique nonce ensures the counter stream never repeats under the same key.

Security consequences of not using an IV or nonce:

- **Stream ciphers and CTR like modes:** If the same IV or nonce repeats under the same key, keystream repeats. Given $C_1 = P_1 \oplus K$ and $C_2 = P_2 \oplus K$, an attacker computes $C_1 \oplus C_2 = P_1 \oplus P_2$ and can recover data with known plaintext or structure.
- **ECB mode** has no IV and reveals equal blocks clearly. ECB is not semantically secure.
- **CBC with fixed IV** leaks equality of first blocks across messages and enables pattern attacks. CBC requires an unpredictable or at least random IV. It must be never reused with the same key.

Requirements vary by mode:

- **CTR and GCM:** Nonce must be unique per key. Randomness is not required if you can guarantee uniqueness, for example by using a counter based nonce or a sequence number with a per key salt. Reuse is catastrophic.
- **ChaCha20 Poly1305:** Uses a 96 bit nonce. It must be unique for a given key. Typical layout uses a per connection salt and a per message counter to ensure uniqueness.
- **CBC:** IV must be unpredictable and unique. Random IV of block size works. It is transmitted alongside the ciphertext.

Why not key only:

- A single fixed key without fresh IV or nonce forces the same starting state for each message. That repeats keystreams or initial blocks and breaks confidentiality. The IV provides per message variability without changing the long term key.
- Keys are long lived and costly to rotate, while IVs or nonces are cheap and can be generated per message. This separation reduces operational risk and supports security proofs.

Best practices:

- Ensure per message uniqueness of IV or nonce under a given key. Prefer counters or sequence numbers that cannot collide.
- Use the mode specific rules. CTR and GCM care about uniqueness. CBC cares about unpredictability. ChaCha20 Poly1305 follows the GCM style uniqueness rule with a 96 bit nonce.
- Never reuse an IV or nonce with the same key unless the mode explicitly allows it, which is rare.

Summary: The IV or nonce is essential for semantic security and to prevent keystream or pattern reuse. Using only a key without IV or nonce is unsafe for almost all practical encryption schemes.

Question 09 — Multi-Precision Libraries (MPI, GMP): Random and Prime Generation; Primality Testing Functions

Random n-bit numbers:

- In GMP, use `mpz_urandomb(mpz_t rop, gmp_randstate_t state, mp_bitcnt_t n)`. This produces a uniform integer $0 \leq rop < 2^n$, i.e., an n-bit random number.
- Alternatively, `mpz_rrandomb` generates a "random-looking" number of exactly n bits using fewer random bits; used for performance, not cryptographic quality.

Random n-bit primes:

- GMP provides `mpz_nextprime(mpz_t rop, const mpz_t op)`. Given op, it returns the next larger probable prime.

- To generate a random n-bit prime: first generate a random n-bit integer x with `mpz_urandomb`, then call `mpz_nextprime(x, x)` to obtain the next prime $\geq x$.

Primality testing in GMP:

- `mpz_probab_prime_p(const mpz_t n, int reps)` returns:
  - 0 if n is definitely composite,
  - 1 if n is "probably prime" (passes several rounds of Miller–Rabin),
  - 2 if n is definitely prime (for small n with deterministic check).
- Internally, GMP uses:
  1. A base-2 Fermat test for small inputs.
  2. Several Miller–Rabin rounds with random bases. Each round reduces error probability to $\leq 1/4$.
  3. A strong Lucas probable prime test for additional assurance.

MPI library (other implementations):

- Follows similar functions: generate uniform n-bit integers, trial division for small primes, and Miller–Rabin or Lucas tests for primality.
- APIs differ, but the cryptographic principle remains: random n-bit integers are created, primality is tested using probabilistic algorithms, and accepted as primes when error probability is negligible.

Conclusion: GMP provides reliable functions for generating random integers and probable primes. Its primality test is Miller–Rabin combined with Lucas, which is industry-standard for key generation in RSA, DSA, and ECC systems.

Question 10 — Wiener Attack on RSA

RSA background:

- Public key: (N, e), where $N = pq$ with large primes p, q.
- Private exponent d satisfies $ed \equiv 1 \pmod{\varphi(N)}$.
- Normally d is large (about same size as N) to avoid vulnerabilities.

Wiener's observation:

- If d is "too small", specifically $d < \frac{1}{3}N^{1/4}$, then RSA is vulnerable.
- Reason: $\frac{e}{N}$ can be approximated by a fraction $\frac{k}{d}$ using continued fractions, since $ed - k\varphi(N) = 1$ with small error.

Attack outline:

algorithm below

1. Compute the continued fraction expansion of $\frac{e}{N}$.
2. Extract convergents $\frac{k_i}{d_i}$ of this expansion.
3. For each candidate (k_i, d_i):
   - Check if $ed_i - 1$ is divisible by k_i. If yes, set $\varphi(N) = (ed_i - 1)/k_i$.
   - Solve $x^2 - (N - \varphi(N) + 1)x + N = 0$. If integer roots p, q exist, factorization is successful.

Why it works:

- The continued fraction convergents of e/N approximate k/d with extraordinary accuracy when d is small.
- For $d < N^{1/4}/3$, the correct $\frac{k}{d}$ appears among the early convergents, enabling efficient recovery.

Complexity:

- Polynomial-time in $\log N$, typically requiring just a few hundred convergents to be tested.
- Much faster than generic factoring of N.

Countermeasures:

- Ensure d is not too small. In practice, use $d \approx \varphi(N)/e$ with $e = 65537$ (standard choice), which produces large d.
- Apply standards like PKCS #1, which prevent using insecure small private exponents.

Conclusion: Wiener's attack breaks RSA if the private exponent d is unusually small ($< N^{1/4}/3$). It exploits continued fraction approximations of e/N and recovers d in polynomial time, making small-exponent RSA keys completely insecure.

Question 11 — Side-Channel Attack on RSA via Square-and-Multiply; How to Prevent

Square-and-multiply method:

- To compute $a^b \bmod N$, where $b = (b_{t-1}b_{t-2}\dots b_0)_2$, we loop over the bits of b.
- For each bit:
  - Always square: $x \leftarrow x^2 \bmod N$.
  - If $b_i = 1$, multiply: $x \leftarrow x \cdot a \bmod N$.
- Total cost depends on the Hamming weight of b. Timing, power, or EM leaks reveal whether multiplications occur at each step.

Side-channel leakage:

- An attacker measures timing or power trace. "Squaring-only" steps look different from "square+multiply" steps.
- Correlating patterns over many runs reveals positions of 1-bits in b, allowing recovery of private key d in RSA.

Countermeasures:

- **Constant-time exponentiation:** Always perform both square and multiply, but mask the result of the multiply depending on the bit:

$$x \leftarrow (x^2 \bmod N) \cdot (a^{b_i} \bmod N) \bmod N.$$

  This ensures identical sequence of operations for all bits.
- **Montgomery ladder:** At each step, update two accumulators using a fixed sequence independent of the bit value. Prevents key-dependent branches and timing.
- **Blinding techniques:**
  - *Message blinding:* Before decryption/signature, compute $a' = a \cdot r^e \bmod N$ with random r. After decryption, divide out r. This makes side-channel traces unrelated to the real input.
  - *Exponent blinding:* Randomize d by computing $d' = d + r\varphi(N)$. Since $a^{d'} \equiv a^d \bmod N$, leakage is obscured.
- **Constant-time arithmetic:** Implement modular multiplication and reduction without data-dependent branches or memory accesses.

Summary: The attack exploits the fact that squaring is always done while multiplication occurs only when exponent bits are 1. Prevention requires eliminating key-dependent branching and ensuring constant-time execution through ladders, blinding, and careful implementation.

Question 12 — Primality Testing: Miller–Rabin vs Solovay–Strassen

Solovay–Strassen test:

- Input: odd integer $n > 2$.
- Pick random base a, $1 < a < n - 1$.
- Compute $x = a^{(n-1)/2} \bmod n$.
- Compute Jacobi symbol $\left(\frac{a}{n}\right)$.
- If $x \equiv \left(\frac{a}{n}\right) \pmod{n}$, a is a strong liar, test passes; otherwise n is composite.

Miller–Rabin test:

- Write $n - 1 = 2^s d$ with odd d.
- Pick random base a, $1 < a < n - 1$.

- Compute $x = a^d \bmod n$.
- If $x = 1$ or $x = n-1$, test passes. Otherwise, square repeatedly: $x \leftarrow x^2 \bmod n$ up to $s-1$ times. If $x = n-1$ at any step, test passes; else n is composite.

Pseudocode (Miller–Rabin):

```
function $MR(n, k)$ :
    if $n$ even return composite
    write $n - 1 = 2^s d$
    for $i = 1..k$ :
        $a \leftarrow$ random in $[2, n-2]$
        $x \leftarrow a^d \bmod n$
        if $x = 1$ or $x = n-1$ continue
        repeat $s-1$ times :
            $x \leftarrow x^2 \bmod n$
            if $x = n-1$ break loop
        else return composite
    return probably prime
```

Error probabilities and complexities:

- **Solovay–Strassen:** If n is composite, at least half of bases a are witnesses. So error probability after k tests is $\leq 2^{-k}$.
- **Miller–Rabin:** If n is composite, at most 1/4 of bases are liars. So error probability after k tests is $\leq 4^{-k}$.
- Both run in $O(k \log^3 n)$ using fast modular exponentiation.

Which is better:

- Miller–Rabin is strictly stronger in error bound (1/4 vs 1/2 per base).
- Solovay–Strassen uses the Jacobi symbol, which adds cost but does not reduce error faster than Miller–Rabin.
- Miller–Rabin is used in practice by libraries such as GMP and OpenSSL. Solovay–Strassen is of more theoretical interest.

Conclusion: Both are fast, probabilistic, one-sided error tests. Miller–Rabin is preferred in practice due to its tighter error bound and efficiency. Solovay–Strassen is historically important but rarely used in cryptographic software today.

Question 13 — Why do we run primality testing for $k = 2^{512} \times 2^{512}$ loops? (Fail probability $= 1/k$).

Background on error probability:

- Probabilistic primality tests (Miller–Rabin, Solovay–Strassen) are *Monte Carlo one-sided* algorithms. If n is prime, they always accept. If n is composite, they may accept with some probability.
- For Miller–Rabin, each round with a random base a detects compositeness with probability at least $3/4$. Therefore, the probability of a composite slipping through t independent rounds is $\leq (1/4)^t = 2^{-2t}$.
- For Solovay–Strassen, error probability per base is at most $1/2$, so after t rounds it is $\leq 2^{-t}$.

What does "$k = 2^{512} \times 2^{512}$" mean?

- Sometimes textbooks or notes phrase the target failure probability as $1/k$ with k extremely large, e.g. 2^{1024}. This comes from multiplying two 512-bit search spaces (key sizes in RSA) and aiming for a failure probability much smaller than the chance of breaking RSA by factoring.
- It does *not* mean literally running 2^{1024} test loops, which is infeasible. Instead, it means: "choose t large enough that the failure probability 4^{-t} (for MR) is less than $1/2^{1024}$".

Concrete calculation:

- We want $4^{-t} < 2^{-1024}$.
- That requires $t > 512$ Miller–Rabin rounds.
- In practice, much fewer rounds suffice because the probability bound is loose. Standards (FIPS 186-4, NIST SP 800-56) recommend around 40–64 rounds for 1024–2048 bit moduli.

Detailed discussion:

- Running too many rounds wastes time, since even a handful of rounds make the chance of error astronomically smaller than hardware failure rates.
- "$2^{512} \times 2^{512}$" is a way of stating "we want the failure probability to be smaller than the inverse of the RSA modulus size" (roughly 1024 bits of security).
- Therefore, we select t such that the probability of misclassifying a composite as prime is negligible compared to any realistic attack on the cryptosystem.

Conclusion: The phrase means setting the number of Miller–Rabin iterations so that the error probability is at most $1/2^{1024}$. This translates to about 512 rounds in theory, but in practice, 40–64 rounds are considered sufficient for cryptographic key generation.

Question 14 — Miller–Rabin on 256-bit primes a, b and composite $n = ab$. Witnesses and non-witnesses.

Setup:

- Let a, b be random 256-bit primes. Then $n = a \cdot b$ is a 512-bit RSA-like modulus.
- We run Miller–Rabin on n with some base w.

- Since n is composite, there are two possibilities for w:
  - w is a *witness*: test detects compositeness, rejects n.
  - w is a *strong liar*: test incorrectly declares n “probably prime”.

Witnesses vs non-witnesses:

- For any odd composite n, at least 3/4 of bases w are witnesses. Thus at most 1/4 of bases are strong liars.
- For semiprimes $n = pq$ with distinct primes p, q, the proportion of strong liars is usually far smaller than 1/4; empirically it is tiny.
- Example: For $n = 15$, bases $\{2, 7, 8, 11, 13\}$ are witnesses while $\{4, 14\}$ are liars. For large semiprimes, liar sets exist but are vanishingly rare.

Failure of MR:

- If by chance we pick only liar bases, MR fails to detect compositeness. The probability of this after t independent bases is $\leq 4^{-t}$.
- For a 512-bit n, a single MR round might rarely misclassify. But running $t = 40$ rounds makes the probability of error $\leq 4^{-40} \approx 2^{-80}$, negligible for cryptographic purposes.

Illustrative run:

- Suppose $n = ab$ with $a = prime_{256}, b = prime_{256}$. Write $n - 1 = 2^s d$ with d odd.
- Pick random w. Compute $x = w^d \bmod n$.
- If $x \not\equiv \pm 1$, square repeatedly. If we never see -1, w is a witness.
- Rarely, for some special w, the sequence cycles without hitting -1, and the test falsely accepts n; such w are strong liars.

Conclusion: Miller–Rabin can in theory fail for $n = ab$, but the probability is extremely small. Most bases are witnesses; a few are non-witnesses (liars). Repeated rounds reduce the error probability exponentially. For 256-bit primes multiplied into a 512-bit modulus, 40 rounds essentially guarantee correct detection of compositeness.

Question 15 — Why did we use `-O3` while compiling? Why `-lgmp`? Why `-march=native`?

`-O3` optimization flag:

- `-O3` tells the compiler (e.g., GCC or Clang) to enable the most aggressive optimizations available.
- This includes:
  - Function inlining: small functions are expanded inline to avoid call overhead.
  - Loop unrolling: repeated operations expanded for fewer jumps and better instruction-level parallelism.

  - Auto-vectorization: transform loops into SIMD instructions if supported by the hardware.
  - Common subexpression elimination, constant folding, and dead code removal.

- The downside: larger binaries, longer compile times, and sometimes less predictable performance due to aggressive reordering.
- In cryptography experiments (timing-sensitive), `-O3` helps maximize performance of repeated arithmetic operations like modular multiplications.

`-lgmp` linking flag:

- GMP (GNU Multiple Precision library) provides high-performance big integer arithmetic.
- The flag `-lgmp` tells the linker to include the GMP library, resolving function calls such as `mpz_mul`, `mpz_mod`, `mpz_probab_prime_p`.
- Without `-lgmp`, you would get "undefined reference" errors when linking.
- This is standard practice: compile with `gcc main.c -lgmp` so that GMP's object code is pulled in.

`-march=native`:

- Instructs the compiler to generate instructions optimized for the host CPU's microarchitecture.
- Enables the use of specialized instructions available on the host (e.g., AVX2/AVX-512 on Intel, NEON on ARM, vectorized rotate and multiply).
- Compiler may choose different instruction scheduling and register usage for the best throughput.
- The trade-off: binaries are not portable. A binary compiled with `-march=native` may not run on a different CPU that lacks those features.
- For cryptographic arithmetic (e.g., modular exponentiation, primality testing), this produces substantial speedups since multi-precision operations heavily benefit from SIMD and specialized multiply instructions.

Conclusion: `-O3` maximizes compiler optimizations, `-lgmp` links the multiple-precision library, and `-march=native` produces host-specific, highly optimized binaries. Together, these ensure fast, efficient big-integer cryptographic computations.

Question 16 — What is inlining and SIMD, why we use them, how they make programs faster.

Inlining:

- A compiler optimization where a function call is replaced directly with the function body.
- Removes function call overhead (stack frame setup, jump, return).

- Allows the compiler to optimize across the boundary — for example, propagate constants into the function, unroll loops, or vectorize combined operations.
- Example:

  ```
  inline int square(int x) { return x*x; }
  ```

  When called, the compiler may replace `square(y)` with `y*y`.
- Useful in performance-critical cryptographic loops where small functions like "modular add" or "rotate" are called millions of times.

SIMD (Single Instruction, Multiple Data):

- An instruction set paradigm where one CPU instruction operates on multiple pieces of data in parallel.
- Examples: Intel SSE/AVX, ARM NEON.
- Instead of adding one pair of 32-bit integers, SIMD can add 4 or 8 pairs simultaneously (depending on width).
- Cryptographic primitives often involve bitwise XOR, rotations, additions, and table lookups — all of which can be vectorized.
- Compilers (`-O3` and `-march=native`) often automatically vectorize loops. Developers can also explicitly use SIMD intrinsics.

Performance impact:

- Inlining reduces branching and makes the code pipeline smoother.
- SIMD increases throughput by exploiting data-level parallelism. For instance, in ChaCha20, 4 blocks can be processed simultaneously using AVX2.
- Combined, they lower latency per operation and boost throughput, which is crucial when testing cryptographic algorithms with very large numbers or long keystreams.

Conclusion: Inlining removes the overhead of function calls and enables better compiler optimization. SIMD executes the same operation on multiple data in parallel. Both techniques are key to high-speed cryptographic implementations where billions of arithmetic or bitwise operations are required.

Question 17 — AES: Internal State, Data Structure, Bit Requirements, and Detailed Steps

Internal state and data structure

- AES operates on a fixed-size **state** of 128 bits arranged as a 4×4 matrix of bytes. Denote bytes as $s_{r,c}$ with row $r \in \{0, 1, 2, 3\}$ and column $c \in \{0, 1, 2, 3\}$.

- Input mapping is **column-major**: if the plaintext bytes are $in[0..15]$, then $s_{r,c} = in[4c+r]$. The same layout is used for round keys and for the final serialization of the state back to ciphertext.
- A natural C structure is `uint8_t state[4][4]` or a flat `uint8_t state[16]` with column-major indexing. Many optimized implementations use four 32-bit words `uint32_t s[4]` for each column.

Block size, key sizes, and number of rounds

- **Block size** is always 128 bits.
- **Key sizes** are 128, 192, or 256 bits, with key-word count $N_k \in \{4, 6, 8\}$.
- **Rounds** N_r are 10, 12, or 14 for 128, 192, or 256-bit keys respectively.

Round functions overview

- Each round transforms the state using:
  1. **SubBytes**: nonlinear byte-wise substitution by an S-box.
  2. **ShiftRows**: cyclic left shifts on each row by offsets $0, 1, 2, 3$.
  3. **MixColumns**: linear mixing of each column in GF(2^8).
  4. **AddRoundKey**: XOR of the state with the round key.
- The **initial round** performs only AddRoundKey. The **final round** omits MixColumns.

SubBytes in detail

- The S-box maps each byte x to $S(x)$. Construction:
  1. Interpret bytes as elements of GF(2^8) with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ (hex 0x11B).
  2. Compute multiplicative inverse x^{-1} in GF(2^8), with convention $0^{-1} = 0$.
  3. Apply a fixed affine transformation over $\mathbb{F}_2^8$: $S(x) = A \cdot x^{-1} \oplus b$, where A is an invertible 8×8 binary matrix and b is a fixed 8-bit vector.
- This gives strong nonlinearity and good resistance to linear and differential cryptanalysis at the byte level.

ShiftRows in detail

- Row 0: no shift. Row 1: shift left by 1 byte. Row 2: shift left by 2 bytes. Row 3: shift left by 3 bytes.
- This step spreads byte dependencies across columns and cooperates with MixColumns to ensure fast diffusion.

MixColumns in detail

- Each column $(s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c})^\top$ is multiplied by an MDS matrix over GF(2^8):

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}.$$

- Multiplication by `02` is `xtime`: left shift by one and conditionally XOR with 0x1B if the original byte had its top bit set. Multiplication by `03` equals `xtime(x) XOR x`.
- The matrix is MDS, so any change to a single input byte affects all four output bytes, maximizing branch number and diffusion.

AddRoundKey

- XOR the state with the 128-bit round key: $s_{r,c} \leftarrow s_{r,c} \oplus k_{r,c}$.
- There are $N_r + 1$ round keys, one for the initial AddRoundKey and one per round.

Key schedule and round keys

- Work with 32-bit words $w[i]$. For AES, $N_b = 4$ words per block, and $N_w = N_b \cdot (N_r + 1)$ total key words: 44, 52, or 60 for 128, 192, or 256-bit keys.
- Initialization: $w[0..N_k - 1]$ are formed by packing the key bytes column-wise.
- For i from N_k to $N_w - 1$:

$$\text{temp} \leftarrow w[i-1].$$

$$\text{if } i \bmod N_k = 0 : \quad \text{temp} \leftarrow \text{SubWord}(\text{RotWord}(\text{temp})) \oplus \text{Rcon}[i/N_k].$$

$$\text{if } N_k = 8 \text{ and } i \bmod N_k = 4 : \quad \text{temp} \leftarrow \text{SubWord}(\text{temp}).$$

$$w[i] \leftarrow w[i - N_k] \oplus \text{temp}.$$

- **RotWord** rotates a 32-bit word by one byte, **SubWord** applies the S-box to each byte, and **Rcon**[j] is $(\text{rc}_j, 0, 0, 0)$ with $\text{rc}_j = 2^{j-1}$ in GF(2^8). First few rc_j: 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36.
- Round keys are taken as consecutive groups of four words and arranged column-wise to match the state layout.

Encryption structure

```
Input: state, round keys $RK[0..N_r]$
    state ← state ⊕ $RK[0]$
    for $r = 1$ to $N_r - 1$ : state ← SubBytes(state)
                               state ← ShiftRows(state)
                               state ← MixColumns(state)
                               state ← state ⊕ $RK[r]$
    Final round $r = N_r$ :    state ← SubBytes(state)
                               state ← ShiftRows(state)
                               state ← state ⊕ $RK[N_r]$
Output ciphertext from state.
```

Decryption structure

- Use inverses: InvShiftRows shifts right by row offsets, InvSubBytes uses the inverse S-box, InvMixColumns multiplies each column by the inverse MDS matrix with coefficients {0E, 0B, 0D, 09}.

```
                state ← state $\oplus$ $RK[N_r]$
for $r = N_r - 1$ down to 1 : state ← InvShiftRows(state)
                              state ← InvSubBytes(state)
                              state ← state $\oplus$ $RK[r]$
                              state ← InvMixColumns(state)
                 Final step:  state ← InvShiftRows(state)
                              state ← InvSubBytes(state)
                              state ← state $\oplus$ $RK[0]$.
```

Bit requirements and memory footprint

- **State**: 128 bits.
- **Round keys**:
  - AES-128: $N_w = 44$ words $\Rightarrow$ 176 bytes of expanded key.
  - AES-192: $N_w = 52$ words $\Rightarrow$ 208 bytes.
  - AES-256: $N_w = 60$ words $\Rightarrow$ 240 bytes.
- **S-box tables**: 256 bytes for S-box and 256 bytes for inverse S-box. Rcon is small (10 bytes used for AES-128).
- **Optional T-tables** optimization: four tables of 256 32-bit words for encryption (about 4 KB each, 16 KB total) and similarly for decryption, trading memory for speed.

Implementation notes and constant-time concerns

- Table lookups can leak through cache timing. Options:
  - Use **bitsliced** or **vectorized** implementations that compute S-box and MixColumns via boolean operations in registers.
  - Use dedicated hardware instructions when available. For example, CPUs with AES instructions compute SubBytes MixColumns and key addition in hardware, reducing timing leakage risk.
- For software without hardware support, prefer constant-time S-box computations or masked table accesses to reduce side-channel leakage.

Why the design gives strong diffusion and confusion

- SubBytes provides nonlinearity and breaks linear relations at the byte level.
- ShiftRows ensures that bytes from the same column are distributed across different columns in the next MixColumns, accelerating spread.
- MixColumns is MDS, so any single-byte change affects all four bytes of a column. Combined with ShiftRows, after two rounds every output byte depends on every input byte.
- AddRoundKey injects key material at every round, preventing slide-style symmetries and ensuring key-dependent transformations throughout.

Summary AES maintains a 128-bit state stored as a 4×4 byte matrix. The round structure is initial AddRoundKey, followed by $N_r - 1$ rounds of SubBytes, ShiftRows, MixColumns, AddRoundKey, with a final round that omits MixColumns. Key expansion produces $N_r + 1$ round keys using RotWord, SubWord, and Rcon. The mathematics in GF(2^8) with carefully chosen S-box and an MDS column mixing matrix provides strong nonlinearity and diffusion coupled with efficient software and hardware realizations.