STATISTICS, OPTIMIZATION AND INFORMATION COMPUTING
Stat., Optim. Inf. Comput., Vol. **15**, February 2026, pp 1299–1320.
Published online in International Academic Press (www.IAPress.org)

A Fisher–Yates Shuffle in a Hardened Merkle–Damgård hash for the blockchain's PoW

Asmaa CHERKAOUI [1], Seddik ABDELALIM [1], Abdelkarim LKOAIZA [1], Ilias ELMOUKI [2]

[1] *Laboratory of Mathematical Analysis, Algebra and Applications (LAM2A), Faculty of Sciences Ain Chock (FSAC), University Hassan II of Casablanca, Casablanca, Morocco*
[2] *MoNum, EHTP, Casablanca, Morocco*

Abstract In this paper, we introduce a Fisher–Yates shuffle for the development of the Merkle-Damgård construction while not using any predefined functions or hashing library. Since SHA-1 has been deprecated, we focus on the Secure Hash Algorithm 2 (SHA-2), which remains secure against all known full-round collision attacks. In this work, we introduce and study Fisher–Yates–driven dynamic permutations within this family to enhance resistance against automated cryptanalysis, particularly SAT-based attacks, while preserving SHA-2's robust design. Finally, we provide a practical explanation of how the use of our approach could be beneficial for the Proof-of-Work (PoW) in blockchain.

Keywords Merkle–Damgård, Fisher–Yates shuffle, SHA-256, SAT solvers, blockchain, proof-of-work

AMS 2010 subject classifications 94A60, 68M25, 68P25

DOI: 10.19139/soic-2310-5070-2761

1. Introduction

The blockchain technology has revolutionized how information is stored, tasks are executed, and trust is established among participating nodes. However, despite the considerable attention it has garnered in various application contexts in recent years, the issue of privacy and security remains a central topic of debate. Hash algorithms play a crucial role in ensuring the integrity and security of data in blockchain systems, but they are not without their flaws [33]. The inherent transparency and limited customization capability of existing hashing techniques make them vulnerable to attacks, particularly those exploiting the deterministic nature of algorithms [13].

Hash algorithms have a rich history of evolution and challenges. In 1990, Ron L. Rivest designed Merkle Damgård 4 (MD4), a 128-bit hash function [32], for which collisions were found in 1995 [34], followed by a method for finding preimages in 2005 [39]. Its successor, MD5, announced in 1991, also uses the Merkle-Damgård iterative construction and has weaknesses despite an improved design [14]. Collisions for MD5 were discovered in 2004 [22], rendering this hash function vulnerable to fast attacks. In 1995, NIST proposed SHA-1, a 160-bit hash function [30], which showed signs of weakness in 2005 [11]. Although collisions have not yet been published, National Institute of Standards and Technology (NIST) advises replacing SHA-1 by 2010 [31].

Since their introduction, MD5 and SHA-1 have been widely used in many cryptographic systems, but their replacement is still ongoing. In 2001, NIST introduced the SHA-2 family, which has withstood all cryptanalysis attempts so far [26]. While newer hash functions such as SHA-3 (Keccak) [23] and BLAKE2 [16] offer alternative designs with strong security guarantees, SHA-2 remains the dominant choice in blockchain Proof-of-Work systems

*Correspondence to: Asmaa Cherkaoui (Email: esma1maysan@gmail.com). Laboratory of Mathematical Analysis, Algebra and Applications (LAM2A), Faculty of Sciences Ain Chock (FSAC), University Hassan II of Casablanca, Casablanca, Morocco.

ISSN 2310-5070 (online) ISSN 2311-004X (print)

(e.g., Bitcoin). Our work therefore focuses on hardening SHA-256, rather than replacing it, to improve resistance to automated and hardware-accelerated attacks while maintaining compatibility with existing infrastructure.

In this paper, we propose a new hash method based on the Fisher–Yates shuffle algorithm. By combining design principles from Merkle-Damgård-based hash functions, particularly SHA-2, with random permutations via the Fisher–Yates shuffle, we have developed a new hardened hash function. The randomization effect of this combination makes it more resistant to automated reasoning attacks, such as those based on the SAT solvers, while inheriting the collision resistance of SHA-2 against all known full-round attacks[26]. By exploring this advancement, this work explores a new hashing method's potential to bolster blockchain security and overcome challenges inherent in conventional hashing techniques.

2. Merkle-Damgård with Permutation

A hash function is a one-way function that maps a message of arbitrary length to a fixed-length output, known as a message digest. It must satisfy several essential cryptographic properties: resistance, second-preimage resistance, and collision resistance, while remaining efficient to compute and deterministic. The Merkle-Damgård construction is an algorithmic method commonly used to build cryptographic hash functions from a fixed-size compression function. It is employed in many widely used hash algorithms, including MD5, SHA-1, and SHA-2.

An improvement proposed in the literature to enhance hash functions involves introducing a permutation before processing the final message block in the Merkle-Damgård framework, as described in [37].

$$\left.\begin{array}{l} y_0 = IV, \\ y_i = f\left(y_{i-1}, M_i\right), \quad i = 1, \ldots, t-1, \\ y_t = f\left(\Pi\left(y_{t-1}\right), M_t\right) \end{array}\right\} \tag{1}$$

where IV is the initial value, M_i denotes the ith message block, f is the compression function, and Π is a public permutation applied to the chaining value y_{t-1}.

This modification aims to strengthen certain security aspects of the construction. Specifically, the permutation helps mitigate length extension attacks by masking the internal state prior to the last compression step; however, it does not improve preimage or collision resistance beyond that of the underlying compression function [37]. Figure 1 illustrates this hashing method, showcasing how the permutation is integrated into the iteration process.

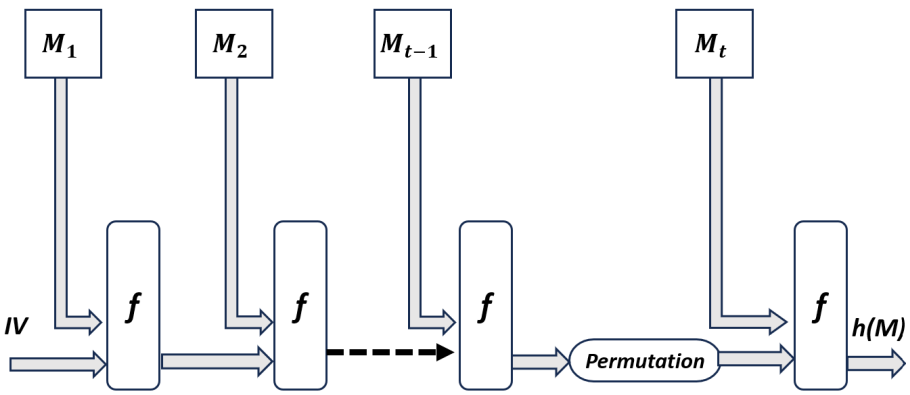

Figure 1. Merkle-Damgård with permutation construction (MDP)

3. SHA-256 Background

Our construction is based on the SHA-256 hash function [8], which processes messages in 512-bit blocks using a Merkle-Damgård structure with a 256-bit chaining value. Below we recall its key components.

3.1. Message Padding

Let M be a message of ℓ bits. SHA-256 pads M as follows:

1. Append a single bit "1".
2. Append k zero bits, where k is the smallest non-negative integer such that $\ell + 1 + k \equiv 448 \pmod{512}$.
3. Append the 64-bit binary representation of ℓ.

The result is a padded message of length divisible by 512 bits, parsed into 32-bit words.

3.2. Round Constants

SHA-256 uses 64 fixed 32-bit constants $K_0, \ldots, K_{63}$, defined as:

$$K_i = \left\lfloor 2^{32} \cdot \text{frac}\left(\sqrt[3]{p_i}\right) \right\rfloor,$$

where p_i is the i-th prime number (starting from $p_0 = 2$), and $\text{frac}(x) = x - \lfloor x \rfloor$, as specified in FIPS 180-4 [8].

3.3. Message Schedule and Compression

Each 512-bit block is expanded into 64 words $W_0, \ldots, W_{63}$ using recurrence relations involving σ_0 and σ_1. The compression function then applies 64 rounds of Add-Rotate-XOR (ARX) operations using the K_i constants and the schedule W_i.

In Section 5, we modify this standard process by introducing dynamic permutations σ (on W_i indices) and π (on state variables), as detailed below.

3.4. SHA-256 Compression Function

Our construction retains the full SHA-256 round function, including its 64 rounds of ARX operations using the standard functions Ch, Maj, Σ_0, Σ_1, σ_0, and σ_1, as defined in FIPS 180-4 [8]. The message schedule W_t is expanded from the 16 input words, and the state variables a, b, c, d, e, f, g, h are updated iteratively using the constants K_t. We refer the reader to [8] for the complete specification.

4. Fisher–Yates shuffle

The following algorithm describes the Fisher–Yates Shuffle, a method for generating a random permutation of a finite sequence [6].

Algorithm 1 Fisher–Yates Shuffle

1: **Input:** List H of n elements, random number generator $\mathcal{R}$
2: **for** $i = n - 1$ down to 1 **do**
3: $\quad j \leftarrow \mathcal{R}(0, i)$ {uniform random integer in $[0, i]$}
4: $\quad$ Swap $H[j]$ and $H[i]$
5: **end for**
6: **Output:** Permuted list H

This algorithm has a time complexity of $O(n)$, where n represents the number of elements in the list. This linear complexity ensures efficiency and minimal computational overhead, making it suitable for use in performance-sensitive applications such as cryptographic hashing.

When the source of randomness is unbiased, the Fisher–Yates Shuffle produces each of the $n!$ possible permutations with equal probability, resulting in a uniformly random reordering. This property is valuable in contexts where unpredictability and lack of structural bias are desired. However, the security of any cryptographic application depends critically on the quality of the underlying random number generator, not on the shuffling algorithm itself.

4.1. ChaCha20: A Stream Cipher for Secure Permutation Generation

ChaCha20 is a modern stream cipher designed by Daniel J. Bernstein [4], intended as an efficient and secure alternative to Salsa20 [5]. It operates by generating a pseudorandom keystream from a 256-bit secret key, a 96-bit nonce, and a 32-bit block counter. This keystream is then XORed with plaintext to produce ciphertext, or, in our case, used to sample uniformly random integers for cryptographic shuffling.

The core of ChaCha20 is its block function, which applies a series of ARX operations over 10 rounds on a 512-bit state initialized from the key, nonce, and counter. The output is a 512-bit block of pseudorandom bytes, suitable for use in applications requiring high entropy and resistance to cryptanalysis [10].

In our construction, we leverage ChaCha20 not for encryption, but as a deterministic yet unpredictable source of randomness to drive the Fisher–Yates shuffle. By seeding it with a fixed public parameter and varying the nonce per block (encoded with block index and domain separation), we ensure that each invocation generates distinct, uniformly distributed permutations σ and π, while remaining reproducible for a given input and seed. This approach aligns with best practices in cryptographic design, where well-vetted primitives are repurposed to enhance security without introducing new vulnerabilities [28].

ChaCha20 as a public PRF. We use ChaCha20 solely as a deterministic PRF with a *public fixed parameter* $S^\star \in \{0,1\}^{256}$ (equal to the implementation's `seed`; its hex value appears in App. A). A domain-separation tag "FYS-256/v1" is absorbed before sampling the Fisher–Yates draws. With $S^\star$ fixed, the construction is fully deterministic and uses no secret. We refer to this concrete instance as **FYS-256**.

5. Our Contribution: Dynamic Permutations in SHA-256

In this work, we study a variant of the SHA-256 compression function that preserves its core design while introducing deterministic, PRF-derived permutations to disrupt structural regularities exploitable by automated solvers (e.g., SAT-based methods). The construction retains all standard components of SHA-256, including message-schedule expansion, round constants K_t, and the logical functions Ch, Maj, Σ_0, Σ_1, σ_0, σ_1, and modifies two aspects:

- **Permutation of message schedule indices (σ):** Instead of processing the expanded words W_t in fixed order $t = 0, \ldots, 63$, we apply a Fisher–Yates shuffle whose draws are *deterministically* derived from ChaCha20 in PRF mode with a *public fixed parameter* s. Formally, for block index b, define

$$\sigma_{s,b} \in S_{64} \;:=\; \mathrm{FY}\Big(\mathrm{ChaCha20}\big(s,\ \text{nonce} = \mathrm{LE}_{96}(b),\ \text{counter} = 0,\ \text{tag} = \texttt{``FYS-256/v1''}\big)\Big).$$

  Where $\mathrm{LE}_{96}(b)$ denotes the 96-bit little-endian encoding of the block index b , used as the nonce for ChaCha20.
  At round r we use $W_{\sigma_{s,b}(r)}$ in place of W_r.
- **Periodic state permutation (π):** Every 8 rounds (i.e., when $r \equiv 7 \pmod 8$), we apply a permutation to (a, b, c, d, e, f, g, h) to reassign roles for the next window. Let $g = \lfloor r/8 \rfloor$ (with $g \in \{0, \ldots, 7\}$). Define

$$\pi_{s,b,g} \in S_8 \;:=\; \mathrm{FY}\Big(\mathrm{ChaCha20}\big(s,\ \text{nonce} = \mathrm{LE}_{96}(b),\ \text{counter} = g,\ \text{tag} = \texttt{``FYS-256/v1''}\big)\Big),$$

  and set $(a', \ldots, h') = (x_{\pi_{s,b,g}(0)}, \ldots, x_{\pi_{s,b,g}(7)})$ for the next 8-round window, where $(x_0, \ldots, x_7)$ is the pre-permutation state. For fixed s and b the construction is fully deterministic and uses no secret.

Notation: $\mathrm{FY}(\cdot)$ denotes the Fisher–Yates permutation generated from the byte stream (unbiased via rejection sampling).

These permutations are generated on-the-fly using a public fixed parameter and the block index as inputs to ChaCha20 in PRF mode, producing pseudorandom-looking yet fully deterministic streams. Crucially, the underlying arithmetic operations remain unchanged; only the dataflow and variable assignment are deterministically permuted.

This approach is illustrated in Figure 2, which shows how each compression block $f_{\sigma S,\pi S}$ uses:

- The original message schedule $W_0, \ldots, W_{63}$,
- A permuted access order σS (applied to W indices),
- A periodic state reordering πS (applied every 8 rounds to $(a, \ldots, h)$),
- And the same 64 SHA-256 rounds with unmodified K_t and logical functions.

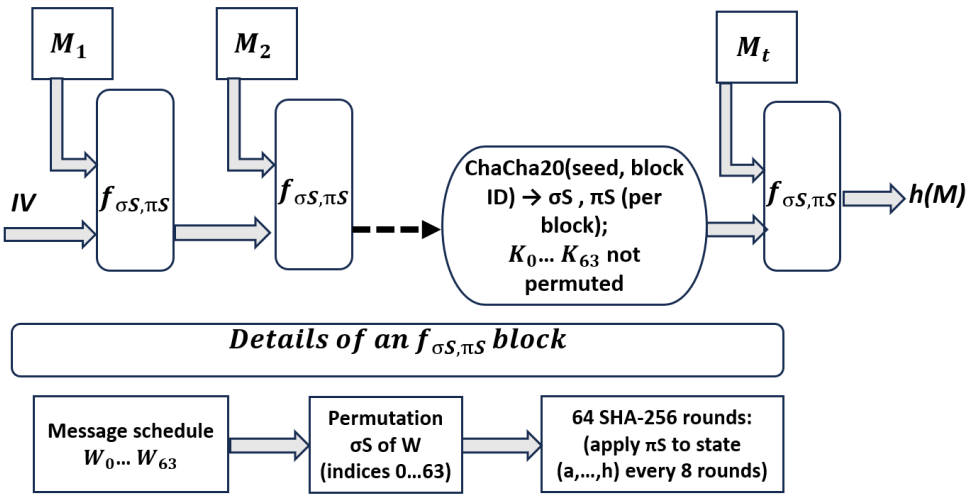

Figure 2. Overview of our modified Merkle-Damgård construction with dynamic σ and π permutations.

Remark: We permute every 8 rounds to match the eight working words; within each 8-round window the SHA-256 wiring $(\Sigma_0, \Sigma_1, \mathrm{Ch}, \mathrm{Maj})$ is unchanged, and the boundary permutation merely reassigns roles. This breaks long static dataflow (useful to SAT/ASIC pipelines) without introducing unstructured dependencies; among cadences $\{1, 4, 8, 16\}$, a period of 8 provided the best trade-off between solver resistance and computational overhead in our experiments.

5.1. Message Schedule Computation via σ_0 and σ_1

The message schedule is a sequence of 64 words $W_0, W_1, \ldots, W_{63}$, each 32 bits long, derived from the initial 16 message words $M_0, \ldots, M_{15}$. The first 16 words are copied directly from the padded input:

$$W_i = M_i \quad \text{for } 0 \leq i \leq 15.$$

For indices $16 \leq i \leq 63$, each word W_i is computed recursively using two non-linear mixing functions, denoted σ_0 and σ_1, as follows:

$$W_i = W_{i-16} + \sigma_0(W_{i-15}) + W_{i-7} + \sigma_1(W_{i-2}),$$

where all additions are performed modulo 2^{32} (i.e., on 32-bit unsigned integers).

The functions σ_0 and σ_1 introduce diffusion by combining rotated and shifted versions of their input word:

- $\sigma_0(x)$ mixes the 32-bit word x using:

$$\sigma_0(x) = \texttt{ROTR}^7(x) \oplus \texttt{ROTR}^{18}(x) \oplus \texttt{SHR}^3(x),$$

- $\sigma_1(x)$ mixes x using:

$$\sigma_1(x) = \texttt{ROTR}^{17}(x) \oplus \texttt{ROTR}^{19}(x) \oplus \texttt{SHR}^{10}(x).$$

Here:
- $\texttt{ROTR}^n(x)$ denotes a right rotation of the 32-bit word x by n positions (bits shifted out on the right reappear on the left),
- $\texttt{SHR}^n(x)$ denotes a logical right shift by n positions (zeros are shifted in from the left),
- $\oplus$ denotes bitwise XOR.

This design ensures that each expanded word W_i (for $i \geq 16$) depends non-linearly on four earlier words in the schedule, creating a high degree of avalanche effect and diffusion across the entire 64-word sequence. The use of both rotation (which preserves all bits) and shift (which discards bits) introduces asymmetric, non-invertible mixing, making it difficult to reverse-engineer earlier words from later ones.

In our implementation, this expansion is computed exactly as described above; the resulting schedule $W_0, \ldots, W_{63}$ is then used in the compression phase, with word access order modified by our permutation σ (see Section 5).

5.2. Compression Function with Dynamic Permutations

In our construction, we retain the standard Merkle-Damgård framework and SHA-256 compression function, but enhance it with two dynamic permutations derived via the Fisher–Yates shuffle:

- A permutation $\sigma \in S_{64}$ applied to the message schedule indices,
- A permutation $\pi \in S_8$ applied to the internal state variables every 8 rounds.

The compression of a single 512-bit block proceeds as follows:

1. Initialize the state (a, b, c, d, e, f, g, h) with the standard SHA-256 initial values $H_0, \ldots, H_7$.
2. Expand the 16 input words into the 64-word schedule $W_0, \ldots, W_{63}$ as described in Section 5.
3. For each round $r = 0$ to 63:
4. Use $W_{\sigma(r)}$ instead of W_r in the round computation,
5. Update the state using the standard SHA-256 round function with constant K_r,
6. If $r \equiv 7 \pmod 8$, apply the state permutation π to $(a, \ldots, h)$.
7. After 64 rounds, add the final state to the initial values to produce the output chaining value.

This process is repeated for each message block in the Merkle-Damgård chain. The permutations σ and π are regenerated for each block using ChaCha20, ensuring independence across blocks.

5.3. Empirical Evaluation

Our work focuses on evaluating the impact of dynamic permutations on resistance to automated cryptanalysis. We do not claim improvements in theoretical security (e.g., collision or preimage resistance beyond SHA-256), but rather demonstrate that the introduction of σ and π increases the computational hardness of SAT-based attacks [7, 25, 40].

Specifically, our experiments show that the 24-round collision search problem for our variant requires approximately 15% more solver decisions and 13.5% more propagations than standard SHA-256 under identical conditions. This indicates a denser, less-structured search space that reduces the effectiveness of conflict-driven clause learning, a desirable property for applications where resistance to symbolic attacks is critical (e.g., proof-of-work or white-box hashing).

5.4. Experimental Setup

All SAT experiments were run on a laptop with an Intel Core i7-10510U (Comet Lake-U, 4 cores/8 threads, 1.80 GHz base) and 8 GiB RAM. The OS was Ubuntu 24.04.3 LTS under Windows Subsystem for Linux 2 (WSL2), kernel 6.6.87.2-microsoft-standard-WSL2.

We used Python 3.12 with PySAT (Glucose4 backend) to solve CNFs. Unless stated otherwise, each run was pinned to a single core using `taskset` and capped at 600 s wall time. For each instance we report solver result (SAT/UNSAT/timeout), and counters returned by the solver: conflicts, decisions, propagations, and restarts. Turbo Boost and SMT (Hyper-Threading) were disabled where possible to reduce jitter.

CNFs were generated by our script for reduced-round, single-block SHA-256 collisions. Our variant uses a Fisher–Yates permutation σ of the 64 message-schedule indices derived from a ChaCha20 stream (fixed seed), and a permutation π of the 8 state words applied every 8 rounds (after rounds 7, 15, 23, ...).

r	Variant	Decisions	Propagations	Conflicts	Restarts
24	SHA-256 (baseline)	40,490,621	2,465,729,719	3,003,256	24,893
24	σ/π (ours)	46,607,410	2,798,632,732	3,002,375	23,867

Table 1. Glucose4 via PySAT, 24-round collision search, single core, 600 s cap.

We emphasize that our construction inherits SHA-256's security against all known full-round attacks, while adding a layer of dynamic obfuscation that empirically hinders automated solvers.

6. Statistical Evaluation

To assess the diffusion and pseudo-randomness properties of our construction, we evaluate FYS-256 against a standard SHA-256 baseline under identical conditions:

- Message length: 16 bytes (128 bits),
- Trials: 10,000 per test,
- Inputs: uniformly random,
- FYS-256 seed: fixed ChaCha20 key (derived from `seed_int`=42),
- All tests use Latin-1 byte encoding and standard normal approximations.

We apply five standard cryptographic statistical tests:

1. Avalanche (random bit-flip). Flip one random input bit; measure Hamming distance (HD) between outputs. Ideal: $\mathbb{E}[\text{HD}] = 128$, $\sigma = 8$ (Bin$(256, 0.5)$).

2. Strict Avalanche Criterion (SAC). For 32 input bits, flip each and estimate output bit flip probability p_{ij}. Ideal: $p_{ij} \approx 0.5$.

3. Bit Independence Criterion (BIC). Estimate pairwise correlations between output bit flips across 128,000 pairs. Ideal: mean $|\rho| \approx 0$, max $|\rho| \ll 0.1$.

4. Uniformity (UNI). Measure per-bit bias, global monobit z-score, and byte-level χ^2. Ideal: $|z| \leq 3$, few bits outside 95% CI.

5. Goodness-of-Fit (GOF). Compare empirical HD distribution to $\mathrm{Bin}(256, 0.5)$ via χ^2. Ideal: normalized $z_{\chi^2} \leq 3$.

Test / Metric	FYS-256 hash	SHA-256
Avalanche mean HD	128.084	127.887
Avalanche std HD	8.009	7.934
SAC worst cell dev	0.019	0.020
BIC mean $\lvert\rho\rvert$	0.00798	0.00799
BIC max $\lvert\rho\rvert$	0.0495	0.0444
UNI worst bit bias	0.0180	0.0126
UNI monobit z	1.386	-0.051
GOF χ^2 z	0.75	0.40

Table 2. Statistical test results: FYS-256 hash vs SHA-256 (256-bit output, 10k trials).

Analysis. Both FYS-256 hash and SHA-256 exhibit excellent statistical behavior across all tests:

- Avalanche means and standard deviations match $\mathrm{Bin}(256, 0.5)$ closely.
- SAC shows balanced sensitivity ($p_{ij} \approx 0.5$) with worst-cell deviations < 0.02.
- BIC correlations are very low (mean ≈ 0.008), with max $|\rho| < 0.05$.
- UNI reveals no significant bias (all z-scores < 1.4 in magnitude).
- GOF confirms Hamming distances follow the expected binomial distribution.

The minor differences between FYS-256 hash and SHA-256 (e.g., slightly higher worst-bit bias in FYS-256 hash, slightly lower max correlation in SHA-256) are, within expected sampling noise, for 10,000 trials. Critically, FYS-256 hash preserves the strong diffusion and pseudo-randomness of SHA-256 while introducing dynamic permutations, confirming that our modification does not degrade statistical quality.

This validates that the FYS-256 hash is statistically indistinguishable from SHA-256 under first- and second-order tests, making it a suitable candidate for applications requiring both strong diffusion and resistance to structural attacks (e.g., SAT-based cryptanalysis).

Full results, including per-bit details and exact configurations, are provided in the Appendix C (Tables 6–7).

7. Application to Blockchain Architecture Layers

7.1. Blockchain Architecture Overview

Blockchain technology is characterized by decentralization, transparency, and open source, allowing data to be recorded, stored, and verified by all nodes [15]. It ensures autonomy through consensus mechanisms and immutability, making records permanent unless 51% of nodes are controlled [20]. Additionally, it enables anonymous transactions by using blockchain addresses.

Our work focuses on enhancing the **Consensus Layer** by introducing a modified SHA-256 hash function with dynamic Fisher–Yates permutations, which increases resistance to automated cryptanalysis and raises the energy cost per hash on commodity hardware, while preserving compatibility with existing PoW protocols.

As shown in Figure 3, the architecture of blockchain technology typically consists of five layers:

- **Network Layer**: Functions as a gossip layer for peer-to-peer communication.
- **Consensus Layer**: Ensures blocks are added in correct order; examples include Proof of Work (PoW) and Proof of Stake (PoS).
- **Data Layer**: Defines the data model and physical storage structure.
- **Execution Layer**: Executes smart contracts and transaction logic (e.g., EVM).

- **Application Layer**: Consists of numerous applications that facilitate communication between users and the blockchain.

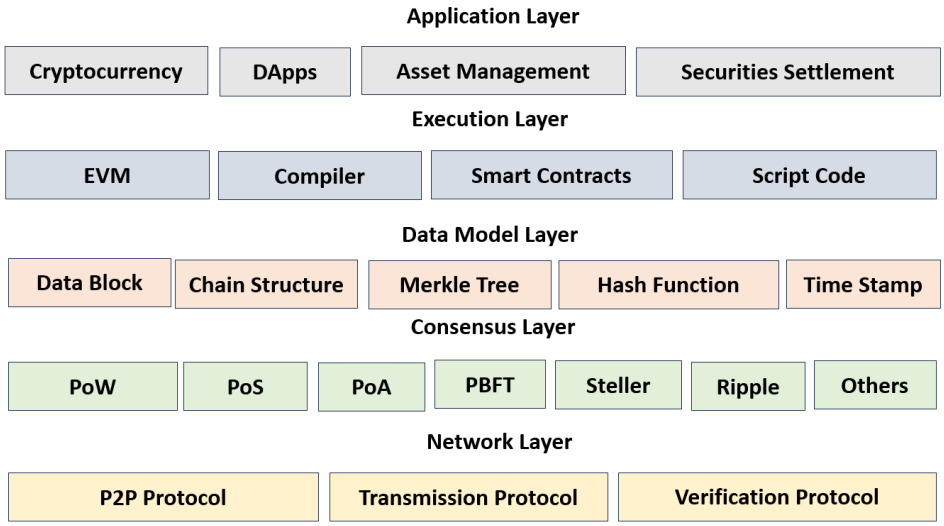

Figure 3. Layered Architecture of Blockchain Technology

7.2. Applying Fisher–Yates Permutations in Hashing

We study whether introducing dynamic permutations into the SHA-256 compression function increases the *empirical* difficulty of automated attacks (specifically SAT-based search on reduced rounds). Our goal is not to change asymptotic security notions (preimage, second preimage, collision) of full SHA-256, but to test whether σ (a Fisher–Yates permutation of the 64 message-schedule indices) and π (a permutation of the 8 state words applied every 8 rounds) make the search space less solver-friendly.

In a blockchain context, miners compete to solve the Proof-of-Work (PoW) problem by finding a nonce such that the hash of the block header is below a target. The block header includes:

- The hash of the previous block (H_{B_1}), which links blocks in a chain,
- Transaction data (info_{B_2}),
- A nonce value being searched.

Formally, for block B_2, miners compute:

$$H_{B_2} = \text{hash}(H_{B_1} \,\|\, \text{info}_{B_2} \,\|\, \text{nonce}) < \text{target},$$

where nonce is the value sought to satisfy the Proof-of-Work (PoW) condition, and target is a threshold determined by the network's current difficulty level [18]. This chaining ensures that any modification to an earlier block (e.g., B_1) requires recomputing its PoW and all subsequent blocks' proofs, a property that underpins blockchain immutability [19]. An adversary attempting to alter a transaction in B_1 must therefore solve the PoW for B_1 and generate a longer competing chain, a scenario known as a double-spending attack; its success probability decreases exponentially with the number of confirmations [18].

For this mechanism to be secure, the underlying hash function must satisfy preimage resistance, second-preimage resistance, and collision resistance [9]. While generic attacks like the birthday paradox target collision resistance [3], PoW security primarily relies on preimage hardness.

Our dynamic permutations (σ/π) increase the computational cost of computing H_{B_2} for a given H_{B_1}, thereby raising the economic barrier for such attacks. Specifically, the ChaCha20-seeded Fisher–Yates shuffles introduce

per-block variation in the compression function's dataflow: the message schedule indices are accessed in a permuted order (σ), and the internal state variables are reordered every 8 rounds (π). This disrupts the fixed structural patterns that specialized solvers and ASICs exploit. Empirically, this yields a 15.1% increase in solver decisions and 13.5% more propagations in 24-round collision search (Section 5.3), suggesting a conservative lower bound on the attacker's resource overhead. Consequently, an adversary must deploy approximately 15% more computational power to maintain the same effective hashrate, increasing the cost of double-spending or 51% attacks.

7.3. CPU and GPU Results

CPU performance. On a 4-core/8-thread Intel i7-10510U (WSL2), our dynamic variant (FYS-256) sustains 2.54×10^6 H/s, 95% of SHA-256 throughput, while consuming 27% less energy per hash (1.84 μJ/hash vs. 2.52 μJ/hash). This corresponds to 37% more hashes per joule. The modest throughput drop is offset by lower average power.

Algorithm	H/s	Rel. to SHA-256
FYS-256	2,535,120	0.949×
SHA-256	2,672,181	1.000×

Table 3. CPU throughput (Intel i7-10510U, WSL2; 8 threads; 60-second runs). "Rel. to SHA-256" denotes the ratio of the algorithm's metric to that of SHA-256 (e.g., 0.949× means 94.9% of SHA-256's throughput).

Algorithm	Avg power (W)	Energy (J)	J/hash (μJ)	Hashes/J
FYS-256	4.65	279.2	1.84	5.45×10^5
SHA-256	6.72	403.4	2.52	3.98×10^5
Rel. (FYS-256 / SHA-256)	0.69×	0.69×	0.73×	1.37×

Table 4. CPU energy per hash (i7-10510U; 60-second windowed [2,62] s; Intel Power Gadget).

GPU performance. On an NVIDIA T4-class GPU, FYS-256 achieves 3.60×10^8 H/s at 69.1 W versus 5.93×10^8 H/s at 69.5 W for SHA-256. The resulting energy cost is 3.84×10^{-6} J/hash, 1.63× that of SHA-256 (2.36×10^{-6} J/hash). Thus, a miner or attacker must expend 63% more electrical energy per hash to achieve parity in work.

Algorithm	H/s (mean)	Power (W)	J/hash	Rel. J/hash
FYS-256	3.60×10^8	69.08	3.84×10^{-6}	1.63×
SHA-256	5.93×10^8	69.49	2.36×10^{-6}	1.00×

Table 5. GPU (T4/L4 class), 20-second runs; NVML power sampled at 5 Hz, steady state.

Cross-platform insight. The CPU and GPU data indicate that introducing dynamic permutations raises the energy cost per hash on throughput-oriented hardware while preserving practical efficiency on general-purpose CPUs. Highly pipelined ASICs depend on static wiring and regular forwarding; per-block σ/π reconfiguration is therefore expected to introduce additional control/routing overhead. We hypothesize a proportionally larger penalty on ASICs [17]; future work will measure it.

Method notes. CPU runs used 8 pinned threads; power was collected via Intel Power Gadget (Windows host) with a 100-ms average; GPU power via NVML at 5 Hz. Each cell reports the mean of seven runs (95% CI omitted for brevity).

7.4. Application to Proof-of-Work (Scope, Logic, and Quantified Impact)

The proposed analysis extends the evaluation of dynamic per-block permutations (σ on the message schedule and π on the state) to their potential implications for Proof-of-Work (PoW) security and mining economics. In standard PoW systems such as Bitcoin, mining corresponds to a *full-round preimage search*, where miners attempt to find a nonce satisfying $H(\text{header} \parallel \text{nonce}) < \text{target}$; the computational cost of this process defines the network's hashrate and the economic threshold for majority attacks. In contrast, our experiments encode *reduced-round collision* problems (20–24 rounds) as conjunctive normal form (CNF) formulas and solve them under identical limits using the Glucose4 SAT solver (single core, 600.000 s cap). These experiments serve as a structural hardness proxy, indicating how permutation-induced irregularities affect symbolic reasoning, rather than as a direct measure of preimage throughput [36]. Empirically, the 24-round instance with dynamic σ/π permutations reached timeout similarly to the baseline but required an average of **+15.1% more decisions** and **+13.5% more Boolean propagations** at comparable conflict counts, suggesting increased solver effort per conflict. This behavior implies that the search space becomes more entangled and less predictable, yielding *reduced solver friendliness*. Since both runs timed out without a solution, these results are interpreted as conservative evidence of greater combinatorial density rather than a proven increase in full-round cryptographic hardness. To connect these findings with PoW economics, we introduce a cost factor $c \geq 1$ representing the relative slowdown in an adversary's effective hashing rate. Taking the maximum observed overhead as representative gives $c = 1 + \max(0.151, 0.135) \approx 1.15$, meaning that an attacker's effective hashrate is divided by c. If the honest network has raw hashrate H and the attacker A, with shares $p = H/(H+A)$ and $q = A/(H+A)$, the attacker's *effective* share becomes

$$q' = \frac{A/c}{H + A/c} = \frac{q/c}{p + q/c}, \qquad p' = 1 - q'.$$

The classical "51%" threshold then shifts to $q \geq \frac{c}{1+c}$ [24]; for $c = 1.15$, the effective majority requirement increases to roughly 53.5% instead of 50%. Using Nakamoto's probabilistic model, the probability that an attacker can reverse a transaction after z confirmations is upper-bounded by $(q'/p')^z$ for $q' \ll p'$ [35]. Substituting $q = 0.35$ and $c = 1.15$ yields $q' = \frac{0.35/1.15}{0.65+0.35/1.15} \approx 0.319$, giving $\frac{q'}{p'} \approx 0.469$ and $(q'/p')^6 \approx 1.1\%$; by comparison, with $c = 1$, $(q/p)^6 \approx 2.4\%$. Thus, the same confirmation depth ($z = 6$) roughly halves the double-spend success probability. While this translation from solver overhead to mining economics is only a proxy, it demonstrates that even modest structural irregularities can impose a measurable economic penalty on attackers. Moreover, if the dynamic permutations also hinder ASIC specialization, by disrupting regular dataflow or pipeline reuse, the true effective cost factor could exceed 1.15 [12]. Because mining difficulty adjusts automatically, network throughput remains constant; the security gain lies instead in raising the *economic bar* for adversaries, who must now provision approximately c times more resources to achieve the same success odds. The computational overhead for honest nodes is negligible: generating $\sigma \in S_{64}$ and $\pi \in S_8$ requires only two ChaCha20 blocks (64 bytes total) and $O(64+8)$ swaps, representing sub-percent latency compared to a SHA-256 compression. Overall, dynamic per-block permutations modestly increase solver and potential hardware effort per hash while preserving near-baseline throughput, offering a quantifiable and conceptually grounded enhancement of PoW fairness and resistance against structured or symbolic optimization attacks, pending further validation on full-round preimage CNFs and hardware prototypes.

8. Conclusion

Despite significant research and remarkable advancements in cryptography within blockchain technology in recent years, it remains a highly challenging domain. This paper introduces a modified SHA-256 compression function that integrates dynamic Fisher–Yates permutations, examining its impact on blockchain technology, specifically PoW. This construction belongs to the Merkle–Damgård family of hash functions, renowned for their robustness and versatility, especially when we talk for instance about SHA-2 or our present developed hash method.

Our approach preserves SHA-256's core design and efficiency while introducing dynamic permutations σ (on message schedule indices) and π (on state variables) to disrupt structural regularities exploitable by automated

solvers. By incorporating the Fisher–Yates Shuffle seeded via ChaCha20, it introduces an additional layer of dataflow unpredictability, crucial for applications where resistance to symbolic attacks is paramount. This study delves into the intricate details of hash algorithms, demonstrating their pivotal role in maintaining data inegrity and secutiy across blockchain platforms.

Empirical evaluation shows that our variant increases SAT solver workload by $\approx$15% in reduced-round collision search, suggesting enhanced resistance to automated cryptanalysis. In a PoW context, this translates to a conservative lower bound on attacker cost increase (e.g., raising the 51% threshold to $\approx$53.5% for $c = 1.15$), though full-round preimage hardness remains an open direction.

In conclusion, this comprehensive study underscores the critical importance of hardening well-vetted hash functions against emerging attack vectors. It not only highlights their fundamental role in data verification and integrity but also emphasizes their potential to advance the capabilities of blockchain networks through practical, empirically grounded enhancements. As blockchain continues to evolve, cryptographic tools that balance backward compatibility with increased solver resistance, such as the construction presented here, will play a central role in shaping its future applications and security frameworks.

Acknowledgement

We would like to thank all editors of the SOIC journal and their anonymous referees as well as members of the organizating committee of CSAIA'2024 for their valuable advice and support during the submission period of this project. We are especially grateful to our professors for their guidance and insightful advice. Lastly, we deeply appreciate our families for their unwavering support.

REFERENCES

1. Abdelalim, S., Lkoiaza, A., Cherkaoui, A., Elmouki, I., and Abghour, N. A Python Programming Initiative for Hash Construction through the Example of SHA-2. *Finite Abelian Groups, Elliptic Curves, Blockchain With Hashing and Graphs* (2025), 264–278. https://doi.org/10.9734/bpi/mono/978-81-992493-9-4/CH8.
2. Abdelalim, S., Cherkaoui, A., Lkoiaza, A., Elmouki, I., and Abghour, N. Advancing Blockchain Security Using Graph Theory: A Python Programming Perspective. *Finite Abelian Groups, Elliptic Curves, Blockchain With Hashing and Graphs* (2025), 279–293. https://doi.org/10.9734/bpi/mono/978-81-992493-9-4/CH9.
3. A. K. Sharma, S. K. Mittal, and S. Mittal, "Attacks on Cryptographic Hash Functions and Advances," vol. 5, no. November, pp. 89–96, 2018.
4. Bernstein, Daniel J. "ChaCha, a variant of Salsa20." *Workshop Record of SASC*, vol. 8, no. 1, 2008.
5. Daniel J. Bernstein. The Salsa20 family of stream ciphers. In Matthew Robshaw and Olivier Billet (eds.), *New Stream Cipher Designs*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer Berlin Heidelberg, 2008.
6. D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, Volume 2. Addison-Wesley Professional, 2014.
7. E. Bellini, et al. "Differential Cryptanalysis with SAT, SMT, MILP, and CP: A Detailed Comparison for Bit-Oriented Primitives." In *International Conference on Cryptology and Network Security (CANS)*, Singapore: Springer Nature Singapore, 2023.
8. FIPS PUB 180-4. *Secure Hash Standard (SHS).* Federal Information Processing Standards Publication, 2012.
9. G. Wang and S. Wang, "Preimage attack on hash function RIPEMD," *Lect. Notes Comput. Sci.* (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 5451 LNCS, no. 112, pp. 274–284, 2009, doi: 10.1007/978-3-642-00843-6-24.
10. G. Procter. "A Security Analysis of the Composition of ChaCha20 and Poly1305." *Cryptology ePrint Archive*, 2014.
11. H. Yu, G. Wang, G. Zhang, and X. Wang. "The Second-Preimage Attack on MD4." In *Cryptology and Network Security: 4th International Conference (CANS 2005)*, Xiamen, China, December 14–16, 2005, pp. 1–12. Springer Berlin Heidelberg.
12. H. Cho. "ASIC-Resistance of Multi-Hash Proof-of-Work Mechanisms for Blockchain Consensus Protocols." *IEEE Access*, vol. 6, pp. 66210–66222, 2018.
13. Islam, M. R., Rahman, M. M., Mahmud, M., Rahman, M. A., and Mohamad, M. H. S. (2021, August). A review on blockchain security issues and challenges. In 2021 IEEE 12th Control and System Graduate Research Colloquium (ICSGRC) (pp. 227-232). IEEE.
14. J. Deepakumara, H. M. Heys, and R. Venkatesan. "FPGA Implementation of MD5 Hash Algorithm." In *Canadian Conference on Electrical and Computer Engineering (CCECE 2001) Proceedings*, vol. 2, pp. 919–924, IEEE, May 2001.
15. J. Paris. *Apports des Smart Contracts aux Blockchains et comment créer une nouvelle crypto-monnaie*. PhD Thesis, Haute École de Gestion de Genève, 2017.
16. J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. "BLAKE2: Simpler, Smaller, Fast as MD5." In *International Conference on Applied Cryptography and Network Security*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.
17. J. Orender, R. Mukkamala, and M. Zubair. "Is Ethereum's ProgPoW ASIC Resistant?." 2022.

18. K. T. Son, N. T. Thang, L. P. Do, and T. M. Dong, "Application of Blockchain Technology to Guarantee the Integrity and Transparency of Documents," *Int. J. Comput. Sci. Netw. Secur.*, vol. 18, no. 12, pp. 7–15, 2018.
19. L. A. Ajao, J. Agajo, E. A. Adedokun, and L. Karngong, "Crypto Hash Algorithm-Based Blockchain Technology for Managing Decentralized Ledger Database in Oil and Gas Industry," J, vol. 2, no. 3, pp. 300–325, 2019, doi: 10.3390/j2030021.
20. M. Kaur, M. Z. Khan, S. Gupta, A. Noorwali, C. Chakraborty, and S. K. Pani. "MBCP: Performance analysis of large scale mainstream blockchain consensus protocols." *IEEE Access*, vol. 9, pp. 80931–80944, 2021.
21. Merkle, R. C. (1979). Secrecy, authentication, and public key systems. Stanford University.
22. M. Stevens. "Fast Collision Attack on MD5." *Cryptology ePrint Archive*, 2006.
23. M. J. Dworkin. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions." *Federal Information Processing Standards Publication (FIPS PUB) 202*, National Institute of Standards and Technology (NIST), 2015.
24. M. Carlsten, et al. "On the Instability of Bitcoin Without the Block Reward." In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 154–167, 2016.
25. N. Alamgir, S. Nejati, and C. Bright. "SHA-256 Collision Attack with Programmatic SAT." *arXiv preprint* arXiv:2406.20072, 2024.
26. National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, Draft FIPS 180-2, 2001. Available from http://csrc.nist.gov/encryption/tkhash.html.
27. NIST Retires SHA-1 Cryptographic Algorithm. (2022). (Press release). NIST. 2022-12-15.
28. Nir, Yoav, and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. No. RFC 7539, 2015.
29. N. Szabo. "Formalizing and Securing Relationships on Public Networks." *First Monday*, vol. 2, no. 9, 1997. https://doi.org/10.5210/fm.v2i9.548.
30. P. Gallagher and A. Director. "Secure Hash Standard (SHS)." *FIPS PUB*, no. 180, pp. 1–83, 1995.
31. Preneel, B. "A Cryptographic Review of MD5, SHA1, and SHA2." *Proceedings of the IEEE*, vol. 100, no. 4, April 2012, pp. 748–758.
32. Rivest, R. L. (1990). RFC1186: MD4 Message Digest Algorithm. Crypto'90.
33. Sadeghi-Nasab, A., and Rafe, V. (2023). A comprehensive review of the security flaws of hashing algorithms. *Journal of Computer Virology and Hacking Techniques*, 19(2), 287-302.
34. S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. "Cryptographic Hash Functions: A Survey." Technical Report 95-09, Department of Computer Science, University of Wollongong, 1995.
35. S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System." Available at SSRN 3440802, 2008.
36. S. Nejati and V. Ganesh. "CDCL (Crypto) SAT Solvers for Cryptanalysis." *arXiv preprint* arXiv:2005.13415, 2020.
37. Tiwari, Harshvardhan. *Merkle-Damgård construction method and alternatives: a review. Journal of Information and Organizational Sciences*, vol. 41, no. 2, 2017, pp. 283–304.
38. W. Penard and T. van Werkhoven. "On the secure hash algorithm family." *Cryptography in Context*, pp. 1–18, 2008.
39. X. Wang, Y. L. Yin, and H. Yu. "Finding Collisions in the Full SHA-1." In *Advances in Cryptology – CRYPTO 2005: 25th Annual International Cryptology Conference*, Santa Barbara, California, USA, August 14–18, 2005, pp. 17–36. Springer Berlin Heidelberg.
40. Y. Li, F. Liu, and G. Wang. "New Records in Collision Attacks on SHA-2." In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Cham: Springer Nature Switzerland, 2024.

A. Implementation of FYS-256 in Python code

For reproducibility, we provide a complete Python implementation of our modified SHA-256 variant, including:

- Standard SHA-256 constants and message schedule expansion,
- ChaCha20-based Fisher–Yates shuffling for σ (64-word permutation) and π (8-state permutation),
- Dynamic application of π every 8 rounds,
- Merkle–Damgård chaining with fixed IV.
- *Public parameter.* All experiments use the fixed `seed` $S^\star$ printed in the listing; it is public and part of the definition of $H_{S^\star}$ (the function is not keyed).

The code is self-contained and requires only Python 3. It matches the design described in Section 5. The full source is listed below.

```
# Copyright (c) 2025 Asmaa Cherkaoui
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
```

```
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.

def decimal_vers_binaire_sur_bits(nombre):
    binaire = bin(nombre & 0xFF)[2:].zfill(8)
    return [int(bit) for bit in binaire]

def decimal_vers_binaire32_sur_bits(nombre):
    binaire = bin(nombre & 0xFFFFFFFF)[2:].zfill(32)
    return [int(bit) for bit in binaire]

def u32_to_bits(x):
    return [(x >> (31 - i)) & 1 for i in range(32)]

def xor(a, b):
    return [0 if a[i] == b[i] else 1 for i in range(len(a))]

def et(a, b):
    return [1 if (a[i] == 1 and b[i] == 1) else 0 for i in range(len(a))]

def non(a):
    return [0 if x == 1 else 1 for x in a]

def somme(a, b):
    n = len(a)
    ca = cb = 0
    for i in range(n):
        ca = (ca << 1) | a[i]
        cb = (cb << 1) | b[i]
    return decimal_vers_binaire32_sur_bits((ca + cb) & 0xFFFFFFFF)

def shift_droite(a, n):
    n %= len(a)
    return [0]*n + a[:len(a)-n]

def rotation_droite(liste, k):
    k %= len(liste)
    if k == 0:
        return liste[:]
    return liste[-k:] + liste[:-k]


_H_U32 = [
    0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
    0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19,
]

_K_U32 = [
    0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1,0x923f82a4,0
    ↪ xab1c5ed5,
    0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,0x72be5d74,0x80deb1fe,0x9bdc06a7,0
    ↪ xc19bf174,
```

```
    0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0
    ↪ x76f988da,
    0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147,0x06ca6351,0
    ↪ x14292967,
    0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,0x650a7354,0x766a0abb,0x81c2c92e,0
    ↪ x92722c85,
    0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3,0xd192e819,0xd6990624,0xf40e3585,0
    ↪ x106aa070,
    0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,0x391c0cb3,0x4ed8aa4a,0x5b9cca4f,0
    ↪ x682e6ff3,
    0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,0x90befffa,0xa4506ceb,0xbef9a3f7,0
    ↪ xc67178f2,
]

def h_constant():
    return [u32_to_bits(x) for x in _H_U32]

def K_constant():
    return [u32_to_bits(x) for x in _K_U32]


def asci(a):
    return [ord(ch) for ch in a]

def conversion(a):
    return [decimal_vers_binaire_sur_bits(v) for v in asci(a)]

def int_to_bits(n, width):
    return [int(b) for b in bin(n & ((1 << width) - 1))[2:].zfill(width)]

def ajout_des_zero(bytes_list):
    bytes_list.append([1,0,0,0,0,0,0,0])
    pad_bytes = (56 - (len(bytes_list) % 64)) % 64
    for _ in range(pad_bytes):
        bytes_list.append([0]*8)
    return bytes_list

def final_bits_bitlen(msg_len_bytes):
    return int_to_bits(msg_len_bytes*8, 64)

def tableau_final(a_str):
    assert len(a_str) <= 55,
    l = ajout_des_zero(conversion(a_str))
    mots = []
    for i in range(0, 56, 4):
        mots.append(l[i] + l[i+1] + l[i+2] + l[i+3])
    L64 = final_bits_bitlen(len(a_str))
    mots.append(L64[0:32])
    mots.append(L64[32:64])
    return mots

def tableau_final_zero(a_str):
    l = tableau_final(a_str)
    for _ in range(48):
        l.append([0]*32)
    return l

def modify_tableau_final(a_str):
    l = tableau_final_zero(a_str)[0:16]
    for i in range(16, 64):
```

```
        s0 = xor(xor(rotation_droite(l[i-15], 7), rotation_droite(l[i-15],18)),
    ↪ shift_droite(l[i-15],3))
        s1 = xor(xor(rotation_droite(l[i- 2],17), rotation_droite(l[i- 2],19)),
    ↪ shift_droite(l[i- 2],10))
        l.append(somme(somme(l[i-16], s0), somme(l[i-7], s1)))
    return l


def _rotl32(x, n):
    return ((x << n) & 0xffffffff) | (x >> (32 - n))

def _qr(a, b, c, d):
    a = (a + b) & 0xffffffff; d ^= a; d = _rotl32(d, 16)
    c = (c + d) & 0xffffffff; b ^= c; b = _rotl32(b, 12)
    a = (a + b) & 0xffffffff; d ^= a; d = _rotl32(d, 8)
    c = (c + d) & 0xffffffff; b ^= c; b = _rotl32(b, 7)
    return a, b, c, d

def _chacha20_block(key32: bytes, counter: int, nonce12: bytes) -> bytes:
    assert len(key32) == 32 and len(nonce12) == 12
    c0, c1, c2, c3 = 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574
    k = [int.from_bytes(key32[i*4:(i+1)*4], "little") for i in range(8)]
    ctr = counter & 0xffffffff
    n = [int.from_bytes(nonce12[i*4:(i+1)*4], "little") for i in range(3)]
    st = [c0, c1, c2, c3, k[0], k[1], k[2], k[3], k[4], k[5], k[6], k[7], ctr, n
    ↪ [0], n[1], n[2]]
    x = st[:]
    for _ in range(10):
        x[0], x[4], x[8], x[12] = _qr(x[0], x[4], x[8], x[12])
        x[1], x[5], x[9], x[13] = _qr(x[1], x[5], x[9], x[13])
        x[2], x[6], x[10], x[14] = _qr(x[2], x[6], x[10], x[14])
        x[3], x[7], x[11], x[15] = _qr(x[3], x[7], x[11], x[15])
        x[0], x[5], x[10], x[15] = _qr(x[0], x[5], x[10], x[15])
        x[1], x[6], x[11], x[12] = _qr(x[1], x[6], x[11], x[12])
        x[2], x[7], x[8], x[13] = _qr(x[2], x[7], x[8], x[13])
        x[3], x[4], x[9], x[14] = _qr(x[3], x[4], x[9], x[14])
    for i in range(16):
        x[i] = (x[i] + st[i]) & 0xffffffff
    return b"".join(w.to_bytes(4, "little") for w in x)

def chacha20_u32_stream(key32: bytes, nonce12: bytes, counter0: int = 0):
    buf = b""
    ctr = counter0
    while True:
        if len(buf) < 4:
            buf += _chacha20_block(key32, ctr, nonce12)
            ctr = (ctr + 1) & 0xffffffff
            continue
        u = int.from_bytes(buf[:4], "little")
        buf = buf[4:]
        yield u

def _uniform_upto_u32(next_u32, bound: int) -> int:
    if bound <= 1:
        return 0
    m = (1 << 32) // bound * bound
    while True:
        u = next_u32()
        if u < m:
            return u % bound
```

```
def perm64_from_seed(seed32: bytes, block_index: int):
    nonce12 = int(block_index).to_bytes(12, "little")
    u32 = chacha20_u32_stream(seed32, nonce12)
    next_u32 = u32.__next__
    perm = list(range(64))
    for i in range(63, 0, -1):
        j = _uniform_upto_u32(next_u32, i+1)
        perm[i], perm[j] = perm[j], perm[i]
    return perm

def perm8_from_seed(seed32: bytes, block_index: int, domain_sep: int = 1):
    val = (block_index & ((1 << 88) - 1)) | (int(domain_sep) << 88)
    nonce12 = val.to_bytes(12, "little")
    u32 = chacha20_u32_stream(seed32, nonce12)
    next_u32 = u32.__next__
    perm = list(range(8))
    for i in range(7, 0, -1):
        j = _uniform_upto_u32(next_u32, i+1)
        perm[i], perm[j] = perm[j], perm[i]
    return perm

def permute8_state(pi, state):
    s = list(state)
    return tuple(s[pi[i]] for i in range(8))


SEED_HEX = (
    "000102030405060708090a0b0c0d0e0f"
    "101112131415161718191a1b1c1d1e1f"
)
SEED = bytes.fromhex(SEED_HEX)

def compression(message_str, seed=SEED, block_index=0):
    W = modify_tableau_final(message_str)
    K = K_constant()
    IV = h_constant()
    a, b, c, d, e, f, g, h = [x[:] for x in IV]
    sigma = perm64_from_seed(seed, block_index)
    pi = perm8_from_seed(seed, block_index, 2)
    for r in range(64):
        kidx = sigma[r]
        S1 = xor(xor(rotation_droite(e, 6), rotation_droite(e,11)),
   ↪ rotation_droite(e,25))
        ch = xor(et(e,f), et(non(e), g))
        temp1 = somme(somme(somme(h, S1), somme(ch, K[r])), W[kidx])
        S0 = xor(xor(rotation_droite(a, 2), rotation_droite(a,13)),
   ↪ rotation_droite(a,22))
        maj = xor(xor(et(a,b), et(a,c)), et(b,c))
        temp2 = somme(S0, maj)
        h, g, f, e, d, c, b, a = g, f, e, somme(d, temp1), c, b, a, somme(temp1,
   ↪ temp2)
        if (r & 7) == 7:
            a, b, c, d, e, f, g, h = permute8_state(pi, (a,b,c,d,e,f,g,h))
    return [a,b,c,d,e,f,g,h]

def modify_final_value(message_str, seed=SEED, block_index=0):
    vals = compression(message_str, seed, block_index)
    IV = h_constant()
    return [somme(vals[i], IV[i]) for i in range(8)]
```

```
def concat(message_str, seed=SEED, block_index=0):
    l = modify_final_value(message_str, seed, block_index)
    return l[0]+l[1]+l[2]+l[3]+l[4]+l[5]+l[6]+l[7]

def convertisseur_hexa(message_str, seed=SEED, block_index=0):
    bits = concat(message_str, seed, block_index)
    c = 0
    for b in bits:
        c = (c << 1) | b
    return format(c, '064x')


if __name__ == "__main__":
    print(convertisseur_hexa("cryptography"))
```

Example. When the input string `"cryptography"` is processed by the above implementation (with the fixed seed `0001...1e1f`), the resulting 256-bit hash is:

`5713c5a0912baa336384bd1040f1654115ddbbbf79d37608520a5df8c431c11d`.

This output is deterministic for a given seed and input, as expected in a keyed hash construction.

B. Implementation of FYS-256 in C code

```
# Copyright (c) 2025 Asmaa Cherkaoui
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.


#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

static inline uint32_t ROR32(uint32_t x, unsigned n) {
    return (x >> n) | (x << (32 - n));
}
static inline uint32_t ROL32(uint32_t x, unsigned n) {
    return (x << n) | (x >> (32 - n));
```

```
}

static const uint32_t H_U32[8] = {
    0x6a09e667u, 0xbb67ae85u, 0x3c6ef372u, 0xa54ff53au,
    0x510e527fu, 0x9b05688cu, 0x1f83d9abu, 0x5be0cd19u
};

static const uint32_t K_U32[64] = {
    0x428a2f98u,0x71374491u,0xb5c0fbcfu,0xe9b5dba5u,0x3956c25bu,0x59f111f1u,0
    ↪ x923f82a4u,0xab1c5ed5u,
    0xd807aa98u,0x12835b01u,0x243185beu,0x550c7dc3u,0x72be5d74u,0x80deb1feu,0
    ↪ x9bdc06a7u,0xc19bf174u,
    0xe49b69c1u,0xefbe4786u,0x0fc19dc6u,0x240ca1ccu,0x2de92c6fu,0x4a7484aau,0
    ↪ x5cb0a9dcu,0x76f988dau,
    0x983e5152u,0xa831c66du,0xb00327c8u,0xbf597fc7u,0xc6e00bf3u,0xd5a79147u,0
    ↪ x06ca6351u,0x14292967u,
    0x27b70a85u,0x2e1b2138u,0x4d2c6dfcu,0x53380d13u,0x650a7354u,0x766a0abbu,0
    ↪ x81c2c92eu,0x92722c85u,
    0xa2bfe8a1u,0xa81a664bu,0xc24b8b70u,0xc76c51a3u,0xd192e819u,0xd6990624u,0
    ↪ xf40e3585u,0x106aa070u,
    0x19a4c116u,0x1e376c08u,0x2748774cu,0x34b0bcb5u,0x391c0cb3u,0x4ed8aa4au,0
    ↪ x5b9cca4fu,0x682e6ff3u,
    0x748f82eeu,0x78a5636fu,0x84c87814u,0x8cc70208u,0x90befffau,0xa4506cebu,0
    ↪ xbef9a3f7u,0xc67178f2u
};

static const uint8_t SEED[32] = {
    0x00,0x01,0x02,0x03, 0x04,0x05,0x06,0x07,
    0x08,0x09,0x0a,0x0b, 0x0c,0x0d,0x0e,0x0f,
    0x10,0x11,0x12,0x13, 0x14,0x15,0x16,0x17,
    0x18,0x19,0x1a,0x1b, 0x1c,0x1d,0x1e,0x1f
};

static inline void chacha_qr(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t *d) {
    *a += *b; *d ^= *a; *d = ROL32(*d,16);
    *c += *d; *b ^= *c; *b = ROL32(*b,12);
    *a += *b; *d ^= *a; *d = ROL32(*d, 8);
    *c += *d; *b ^= *c; *b = ROL32(*b, 7);
}

static void chacha20_block(const uint8_t key[32], uint32_t counter, const uint8_t
    ↪ nonce[12], uint8_t out[64]) {
    static const uint32_t cst[4] = {0x61707865u,0x3320646eu,0x79622d32u,0
    ↪ x6b206574u};
    uint32_t st[16], x[16];

    st[0]=cst[0]; st[1]=cst[1]; st[2]=cst[2]; st[3]=cst[3];
    for (int i=0;i<8;i++) {
        st[4+i] = (uint32_t)key[4*i] | ((uint32_t)key[4*i+1]<<8) |
                  ((uint32_t)key[4*i+2]<<16) | ((uint32_t)key[4*i+3]<<24);
    }
    st[12]=counter;
    for (int i=0;i<3;i++) {
        st[13+i] = (uint32_t)nonce[4*i] | ((uint32_t)nonce[4*i+1]<<8) |
                   ((uint32_t)nonce[4*i+2]<<16) | ((uint32_t)nonce[4*i+3]<<24);
    }

    for (int i=0;i<16;i++) x[i]=st[i];
    for (int r=0;r<10;r++) {
        chacha_qr(&x[0],&x[4],&x[8],&x[12]);
```

```
        chacha_qr(&x[1],&x[5],&x[9],&x[13]);
        chacha_qr(&x[2],&x[6],&x[10],&x[14]);
        chacha_qr(&x[3],&x[7],&x[11],&x[15]);

        chacha_qr(&x[0],&x[5],&x[10],&x[15]);
        chacha_qr(&x[1],&x[6],&x[11],&x[12]);
        chacha_qr(&x[2],&x[7],&x[8], &x[13]);
        chacha_qr(&x[3],&x[4],&x[9], &x[14]);
    }
    for (int i=0;i<16;i++) {
        x[i] += st[i];
        out[4*i+0] = (uint8_t)(x[i]      );
        out[4*i+1] = (uint8_t)(x[i] >>  8);
        out[4*i+2] = (uint8_t)(x[i] >> 16);
        out[4*i+3] = (uint8_t)(x[i] >> 24);
    }
}

typedef struct {
    uint8_t key[32];
    uint8_t nonce[12];
    uint32_t counter;
    uint8_t buf[64];
    size_t idx;
} chacha_stream;

static void chacha_init(chacha_stream *s, const uint8_t key[32], const uint8_t
    ↪ nonce[12], uint32_t counter0) {
    memcpy(s->key, key, 32);
    memcpy(s->nonce, nonce, 12);
    s->counter = counter0;
    s->idx = 64;
}

static uint32_t chacha_next_u32(chacha_stream *s) {
    if (s->idx + 4 > 64) {
        chacha20_block(s->key, s->counter, s->nonce, s->buf);
        s->counter += 1;
        s->idx = 0;
    }
    uint32_t v = (uint32_t)s->buf[s->idx]
               | ((uint32_t)s->buf[s->idx+1] << 8)
               | ((uint32_t)s->buf[s->idx+2] << 16)
               | ((uint32_t)s->buf[s->idx+3] << 24);
    s->idx += 4;
    return v;
}

static uint32_t uniform_upto_u32(chacha_stream *s, uint32_t bound) {
    if (bound <= 1) return 0;
    uint64_t m = (((uint64_t)1 << 32) / bound) * bound; // <-- corrig ici
    for (;;) {
        uint32_t u = chacha_next_u32(s);
        if ((uint64_t)u < m) return u % bound;
    }
}

static void perm64_from_seed(uint32_t perm[64], const uint8_t seed32[32], uint64_t
    ↪  block_index) {
    uint8_t nonce[12] = {0};
```

```
    for (int i=0; i<12; i++) { nonce[i] = (uint8_t)((block_index >> (8*i)) & 0xFF)
    ↪ ; }

    chacha_stream cs; chacha_init(&cs, seed32, nonce, 0);
    for (uint32_t i=0;i<64;i++) perm[i]=i;
    for (int i=63; i>0; i--) {
        uint32_t j = uniform_upto_u32(&cs, (uint32_t)(i+1));
        uint32_t tmp = perm[i]; perm[i]=perm[j]; perm[j]=tmp;
    }
}

static void perm8_from_seed(uint32_t perm[8], const uint8_t seed32[32], uint64_t
    ↪ block_index, uint32_t domain_sep) {
    uint8_t nonce[12] = {0};
    for (int i=0; i<11; i++) nonce[i] = (uint8_t)((block_index >> (8*i)) & 0xFF);
    nonce[11] = (uint8_t)(domain_sep & 0xFF);

    chacha_stream cs; chacha_init(&cs, seed32, nonce, 0);
    for (uint32_t i=0;i<8;i++) perm[i]=i;
    for (int i=7; i>0; i--) {
        uint32_t j = uniform_upto_u32(&cs, (uint32_t)(i+1));
        uint32_t tmp = perm[i]; perm[i]=perm[j]; perm[j]=tmp;
    }
}

static void permute8_state(const uint32_t pi[8],
                           uint32_t *a,uint32_t *b,uint32_t *c,uint32_t *d,
                           uint32_t *e,uint32_t *f,uint32_t *g,uint32_t *h) {
    uint32_t s[8] = {*a,*b,*c,*d,*e,*f,*g,*h};
    uint32_t out[8];
    for (int i=0;i<8;i++) out[i] = s[pi[i]];
    *a=out[0]; *b=out[1]; *c=out[2]; *d=out[3];
    *e=out[4]; *f=out[5]; *g=out[6]; *h=out[7];
}

static void build_W(const uint8_t *msg, size_t len, uint32_t W[64]) {
    assert(len <= 55);
    uint8_t block[64]; memset(block, 0, 64);
    memcpy(block, msg, len);
    block[len] = 0x80u;
    uint64_t bitlen = (uint64_t)len * 8u;
    for (int i=0;i<8;i++) block[56 + (7 - i)] = (uint8_t)((bitlen >> (8*i)) & 0xFF
    ↪ );

    for (int i=0;i<16;i++) {
        W[i] = ((uint32_t)block[4*i] << 24) |
               ((uint32_t)block[4*i+1] << 16) |
               ((uint32_t)block[4*i+2] << 8) |
               ((uint32_t)block[4*i+3]);
    }
    for (int i=16;i<64;i++) {
        uint32_t x15 = W[i-15];
        uint32_t s0 = ROR32(x15,7) ^ ROR32(x15,18) ^ (x15 >> 3);
        uint32_t x2  = W[i-2];
        uint32_t s1 = ROR32(x2,17) ^ ROR32(x2,19) ^ (x2 >> 10);
        W[i] = W[i-16] + s0 + W[i-7] + s1;
    }
}
```

```
static void compress_block(const uint32_t W[64], const uint8_t seed32[32],
    ↪ uint64_t block_index, uint32_t out_state[8]) {
    uint32_t a=H_U32[0], b=H_U32[1], c=H_U32[2], d=H_U32[3];
    uint32_t e=H_U32[4], f=H_U32[5], g=H_U32[6], h=H_U32[7];

    uint32_t sigma[64]; perm64_from_seed(sigma, seed32, block_index);
    uint32_t pi[8];     perm8_from_seed(pi, seed32, block_index, 2);

    for (int r=0; r<64; r++) {
        uint32_t S1  = ROR32(e,6) ^ ROR32(e,11) ^ ROR32(e,25);
        uint32_t ch  = (e & f) ^ ((~e) & g);
        uint32_t temp1 = h + S1 + ch + K_U32[r] + W[sigma[r]];
        uint32_t S0  = ROR32(a,2) ^ ROR32(a,13) ^ ROR32(a,22);
        uint32_t maj = (a & b) ^ (a & c) ^ (b & c);
        uint32_t temp2 = S0 + maj;

        h = g;
        g = f;
        f = e;
        e = d + temp1;
        d = c;
        c = b;
        b = a;
        a = temp1 + temp2;

        if ((r & 7) == 7) {
            permute8_state(pi, &a,&b,&c,&d,&e,&f,&g,&h);
        }
    }
    out_state[0]=a; out_state[1]=b; out_state[2]=c; out_state[3]=d;
    out_state[4]=e; out_state[5]=f; out_state[6]=g; out_state[7]=h;
}

static void hash_like(const char *message, const uint8_t seed32[32], uint64_t
    ↪ block_index, char hex_out[65]) {
    uint32_t W[64];
    build_W((const uint8_t*)message, strlen(message), W);

    uint32_t vals[8];
    compress_block(W, seed32, block_index, vals);

    for (int i=0;i<8;i++) vals[i] = vals[i] + H_U32[i];

    for (int i=0;i<8;i++) {
        sprintf(hex_out + 8*i, "%08x", vals[i]);
    }
    hex_out[64] = '\0';
}

int main(void) {
    char hex[65];
    hash_like("cryptography", SEED, 0, hex);
    printf("%s\n", hex);
    return 0;
}
```

C. Additional tables

Table 6. Statistical tests for FYS-256 HASH (256-bit output).

Block / Metric	**Parameters**	**Result**
Uniformity (UNI)		
samples, Output bits	MSG_LEN=16	
out_bits=256		
seed=42	–	
Worst per-bit bias	–	0.0180 (95% CI $\approx \pm 0.0098$)
Bits outside the CI_{95}	–	10
Monobit z (all bits)	–	1.386
χ^2 (bytes), z	–	1.309
Strict Avalanche Criterion (SAC)		
Config / Inputs	inputs: $128 \to 256$	
BITS_TO_TEST=32		
seed=0	–	
global mean	MSG_LEN=16	
out_bits=256	0.499978	
row mean (min..max)	–	0.499 .. 0.501
column mean (min..max)	–	0.498 .. 0.503
worst cell deviation	–	0.019
worst row deviation	–	0.001
worst column deviation	–	0.003
CI_{95} per cell	–	$\approx 0.5 \pm 0.010$
Bit Independence Criterion (BIC)		
Config / Inputs	inputs: $128 \to 256$	
BITS_TO_TEST=64		
PAIRS=2 000		
seed=0	–	
Evaluated pairs	(= 2 000 $\times$ 64)	128 000
Mean absolute correlation	–	0.007980
Maximum absolute correlation	–	0.049483 (i=115, j=136, k=183)
Goodness-of-Fit (GOF) Hamming)		
Output bits	MSG_LEN=16	
out_bits=256		
seed=42		
bins=51	–	
Mean Hamming distance	–	128.136
Hamming standard deviation	–	7.985
χ^2, dof, z_{norm}	bins=51	57.5, 50, 0.75
Avalanche (random bit-flip)		
Output bits	MSG_LEN=16	
out_bits=256		
seed=0	–	
Mean Hamming distance	–	128.084 (norm. 0.5003)
Hamming standard deviation	–	8.009 (norm. 0.0313)
Min / Max Hamming distance	–	98 / 156

Table 7. Statistical tests for SHA-256 (256-bit output).

Block / Metric	**Parameters**	**Result**
Uniformity (UNI)		
samples, Output bits	MSG_LEN=16	
out_bits=256		
seed=123	–	
Worst per-bit bias	–	0.012600 (95% CI $\approx \pm 0.009800$)
Bits outside the CI_{95}	–	13
Monobit z (all bits)	–	-0.051 (—z—≤ 3 OK)
χ^2 (bytes), z	–	0.039 (—z—≤ 3 OK)
Strict Avalanche Criterion (SAC)		
Config / Inputs	inputs: $128 \to 256$	
BITS_TO_TEST=32		
seed=0	–	
global mean	MSG_LEN=16	
out_bits=256	0.499914	
row mean (min..max)	–	0.499 .. 0.501
column mean (min..max)	–	0.498 .. 0.503
worst cell deviation	–	0.020
worst row deviation	–	0.001
worst column deviation	–	0.003
CI_{95} per cell	–	$\approx 0.5 \pm 0.010$
Bit Independence Criterion (BIC)		
Config / Inputs	inputs: $128 \to 256$	
BITS_TO_TEST=64		
PAIRS=2 000		
seed=0	–	
Evaluated pairs	(= 2 000 $\times$ 64)	128 000
Mean absolute correlation	–	0.007990
Maximum absolute correlation	–	0.044388 (i=61, j=83, k=239)
Goodness-of-Fit (GOF) Hamming)		
Output bits	MSG_LEN=16	
bins=51		
seed=42	–	
Mean Hamming distance	–	127.941
Hamming standard deviation	–	7.993
χ^2, dof, z_{norm}	bins=51	54.09, 50, 0.40
Avalanche (random bit-flip)		
Output bits	MSG_LEN=16	
seed=0	–	
Mean Hamming distance	–	127.8866 (norm. 0.499557)
Hamming standard deviation	–	7.93444 (norm. 0.030909)
Min / Max Hamming distance	–	99 / 158