# Sponge-based Lightweight Hash Function

Abdullah Alshamdayn
*Louisiana Tech University*
Ruston, Louisiana 71270
aal132@latech.edu

Manki Min
*Louisiana Tech University*
Ruston, Louisiana 71270
mankimin@latech.edu

***Abstract*—The rapid growth of resource-constrained devices, combined with advances in telecommunications, has significantly expanded the number of connected systems, enabling the development of affordable, energy-efficient, portable, and high-performance sensors for diverse applications. However, this progress introduces security and privacy concerns related to the reliability of hardware, software, and communication infrastructure. Cryptographic hash algorithms ensure the preservation of data integrity, even as adversaries' computational capabilities continue to advance. Lightweight cryptographic hash functions, a resource-optimized variant of conventional primitives, effectively address the challenges of securing communication in systems like RFID tags and wireless sensors. Their applicability extends to other domains with similar characteristics, such as embedded medical devices, consumer IoT, industrial IoT, and pervasive sensing systems. Moreover, this work introduces a 256-bit lightweight hash function for constrained devices. The proposed design employs the sponge construction as its mode of operation. Additionally, comprehensive software implementation results are reported across multiple metrics. The evaluation demonstrates that the proposed algorithm achieves favorable trade-offs between performance and security, making it well-suited for a wide range of devices with limited computational and storage resources.**

***Index Terms*—Lightweight cryptography, Lightweight hash function, Sponge construction, Resource-constrained**

## I. Introductio

The use of resource-constrained devices is rapidly increasing. Securing these small sensor nodes, which communicate with gateway devices, presents a challenge. These devices are often deployed in critical settings, where unauthorized disclosure of information or access can lead to severe consequences. Ensuring data integrity is essential, as it verifies that the sensor data has not been altered during transmission from the nodes to the edge, fog, or cloud server. Cryptographic hash functions enable the identification of unauthorized modification of the transmitted sensor data. This enables the verification of sensor accuracy on the gateway or applications, significantly enhancing the overall security.

The majority of constrained devices, such as Radio-Frequency Identification (RFID) tags and wireless sensors, have limited resources, including restricted memory, i.e., Read-Only Memory (ROM) and Random Access Memory (RAM), low computing power, and a small physical footprint. These devices are designed for real-time applications that require fast and precise responses while maintaining strong security within the limited resources available. This dual requirement poses a challenge. Memory footprint, execution time, and energy consumption must be balanced against the desired security level. Classical cryptographic algorithms, while secure, are often too costly and unsuitable for efficient implementation in highly constrained environments; they are not viable options. The use of these classical cryptographic standards on constrained devices may result in inadequate performance. The challenges associated with traditional cryptography are resolved by lightweight cryptography, a specialized area that incorporates low memory footprint, minimal energy consumption, and real-time responsiveness on devices with limited resources.

Lightweight cryptography (LWC) is employed to provide security services, ensuring confidentiality and integrity, in situations that require efficient cryptographic approaches in limited devices. This includes industrial, medi-

cal, and infrastructure deployments, as well as other domains with similar constraints. Another aspect of lightweight cryptography is its applicability not only to resource-constrained devices such as RFID tags and sensors but also to other devices that possess ample resources and interact with them directly to perform tasks using secrets.

This paper introduces a 256-bit lightweight hash for constrained platforms. The design utilizes a sponge construction, maintaining a compact memory footprint while achieving practical speed. The objective is to develop a resource-balanced primitive that preserves diffusion under both typical and adversarial inputs.

The structure of this paper is as follows: Section II provides a background on lightweight hash functions. Section III presents the proposed work. Finally, Section V concludes the work and outlines future work.

## II. Background

Resource-constrained devices face strict resource limitations. They have limited memory for both storage and execution, and communication in these systems is often intensive due to the large number of interconnected sensor nodes and the complex communication protocols that extend across gateways, edge devices, fog nodes, and cloud servers. Hash functions must be implemented efficiently across heterogeneous platforms to provide integrity checks without degrading the system's responsiveness.

A cryptographic hash function $\mathcal{H}$ is a mathematical algorithm that takes a variable-length bitstring $m$ as input to produce a fixed-length hash output $y$. They are utilized in a wide range of applications, including message authentication codes, pseudorandom number generators, digital signatures, key generation functions, and even post-quantum cryptographic schemes. The hash function $\mathcal{H}$ must possess the following properties:

1) Compression
   The function $\mathcal{H}$ takes an input $m$ of finite length and maps it to a hash digest $\mathcal{H}(m)$ of length $n$:
   $$\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^n.$$
2) Efficiency
   Computing the hash value $\mathcal{H}(m)$ must be efficient. In lightweight settings, this requires low RAM/ROM usage, as well as a low number of clock cycles per byte, on constrained devices.

Deploying resource-constrained devices in critical applications requires a reduced memory footprint and high computational performance. In such environments, many classical cryptographic primitives are impractical because their implementations demand substantial processing time and memory, especially for hash functions, where execution-time memory is crucial. The efficiency of a lightweight cryptographic system depends on the hash function's throughput (cycles per byte) and RAM/ROM usage while securely producing a fixed-size digest from variable-length inputs. In this setting, the hash must maximize efficiency without incurring excessive computational or storage overhead, achieving an appropriate trade-off between security and resource consumption.

However, these limitations highlight the importance of device security and motivate lightweight cryptographic (LWC) techniques. LWC is an emerging field that addresses the rapid growth of constrained devices by incorporating cryptographic primitives into limited-resource platforms without compromising security. Lightweight designs strike a balance between cost, energy consumption, efficiency, and security by optimizing implementation overhead while maintaining practical cryptographic strength.

When designing a cryptographic hash function, both security and performance aspects must be carefully considered. An effective algorithm should achieve a balanced trade-off between cost, efficiency, and security.

## III. Proposed Work

The proposed 256-bit hash adopts the sponge construction [1] as its mode of operation. Under the sponge model, security is determined by the capacity $c$ and the robustness of the underlying permutation (roughly $2^{c/2}$ for collisions and $2^c$ for preimages). With a 16-round permutation function, this work uses a 352-bit state, where the rate ($r$) is 96 bits and the capacity ($c$) is 256 bits. The hash algorithm is formulated as:

$$\text{Hash}(M) = \text{Squeeze}(\text{Permutation}^{16}(\text{Absorb}(M)))$$

The sponge operates in two steps: the absorbing phase (input processing) and the squeezing phase (output generation). The permutation function $f$ is the core component of the Sponge-based hash function. This scheme utilizes a

Substitution Box (S-box) and Add-Rotate-XOR operations to ensure diffusion (spreading input across the internal state) and confusion (hiding any statistical relationships). The proposed permutation function is presented in Algorithm 1.

**Algorithm 1:** hash-256 Permutation

**Input:** State $S$, Round Constants $RC$
**Output:** Permuted State $S$

**Function** permutation($S, RC$);
**for** $r \leftarrow$ *1* **to** *16* **do**
  **for** $i \leftarrow$ *0* **to** *10* **do**
    $S[i] \leftarrow (S[i] + \text{RotL}(S[(i-1) \bmod 11], 7) + RC[r]) \oplus (RC[r] \ll (i \bmod 16))$;
  **for** $i \leftarrow$ *0* **to** *10* **do**
    $S[i] \leftarrow \bigoplus_{k=0}^{7} SBOX\,[(S[i] \gg (4k)) \,\&\, 0xF] \ll (4k)$;
  **for** $i \leftarrow$ *0* **to** *10* **do**
    $S[i] \leftarrow \text{RotL}(S[i] \oplus S[j], 5) + \text{mix}(\text{RotL}(S[j] + S[k], 9)) + RC[r]$

The steps of the hash function are as follows:

*1) Padding*

This is an important step to ensure the input message fits into the sponge construction fixed rate ($r$) size. This process involves calculating the number of additional bytes required to extend the message to the next multiple of the rate ($r$) block size. If the message length is already a multiple of the rate ($r$) block size, a full padding block is appended to ensure proper processing. An extra block is added if not a multiple of the rate ($r$) size.

*2) Absorbing Phase*

In this phase, the state $S$ is initialized to zero, then after the padded message is split into blocks $B_1, B_2, ..., B_n$, each block $B_i$ is XORed into the first 96 bits of the state. After absorbing each block, the permutation function mixes the input message thoroughly with the internal state. This process continues until all the blocks are processed.

*3) Permutation Function f*

This is the core component of the sponge construction. It consists of 16 rounds, each of which includes three main steps: add-rotate-XOR (ARX) operations, substitution box (S-box), and an additional diffusion layer. Each operation or layer serves a certain cryptographic purpose as follows:

3-1) Addition, Rotation, and XOR (ARX) Layer

These three cryptographic operations are chosen to achieve three cryptographic properties, including nonlinearity through addition, diffusion through rotation, and confusion through XOR mixing. For each word $S[i]$ in the state, for $i = 0 \ldots 10$,

$$S[i] = \underbrace{S[i] + \text{RotL}\big(S[(i-1) \bmod 11],\, 7\big) + RC_r}_{\text{Additive layer}} \oplus \underbrace{(RC_r \ll (i \bmod 16))}_{\text{XOR layer}}.$$

The previous state word $S[(i-1) \bmod 11]$ is rotated left by seven bits. The rotated value is then added to the current word $S[i]$ together with the round constant $RC_r$. Finally, the resulting sum is XORed with the round constant left-shifted by $i$ mod 16.

3-2) The Substitution Box (S-Box) Layer

The S-box introduces nonlinearity to resist linear cryptanalysis, where the adversaries attempt to approximate the function's behavior using linear equations. The scheme's 4-bit S-box is applied to each 4-bit segment (nibble) of the 32-bit words that make up the internal state. The substitution Box design is as follows:

| **x** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **S(x)** | 7 | C | B | 1 | 9 | E | 4 | 2 | 0 | 6 | A | F | 8 | 5 | 3 | D |

For each nibble position $k \in \{0, \ldots, 7\}$ representing the eight 4-bit nibbles comprising a 32-bit word, the algorithm extracts the nibble, substitutes it through the S-box to introduce non-linearity, and finally concatenates the eight substituted nibbles to reconstruct the updated 32-bit word.

3-3) Additional Mixing Layer

The algorithm processes the internal state using a sliding window of three consecutive 32-bit words. For each index $i$, it selects the triplet $S[i]$, $S[j]$, and $S[k]$ (with $j = (i+1) \bmod 11$ and $k = (i+2) \bmod 11$). This rotation ensures that every word in the state is mixed with its immediate neighbors, increasing diffusion. The selected words are then combined through a series of XOR, addition, and rotation operations, followed by mixing with a round constant. This strengthens the overall transformation by

thoroughly diffusing small input changes across the state.

## IV. Results and Discussion

This section describes the software implementation of the proposed hash function and the evaluation methodology. The study employs established metrics and tests suitable for constrained devices. Open-source tools in [6], [7] are used to facilitate benchmarking. The goal of the proposed work is to minimize memory usage without compromising security. The evaluation methodology and results are presented below.

### A. Implementation Cost and Efficiency Metrics

- ROM: It refers to the code size of the hash implementation. This includes any static tables or constants (for example, S-boxes and round constants).
- RAM: It refers to the peak memory during hashing. This includes the internal state, stack frames, temporary variables, and small buffers used for absorption/squeezing or padding.
- Throughput: It refers to the rate at which the software implementation processes data.

### B. Diffusion and Statistical Tests

- Avalanche Test: It evaluates how well the hash function diffuses small changes in the input across its output bits. A strong avalanche effect means flipping a single input bit changes around 50% of output bits. This test helps assess unpredictability and the uniform spread of output changes.
- Combination Test: It evaluates the hash's output when given structured inputs consisting of repeated blocks, byte combinations, or adversarially designed patterns to examine distribution quality and reveal possible bias or clustering. It is to check that the hash produces well-distributed outputs in real-world scenarios where inputs include structures such as network packets and memory blocks.
- Zeroes Test: It is a stress test to evaluate how the hash function behaves when given inputs that are entirely or mostly composed of zero bytes. This edge case is important because weak hash functions often fail to sufficiently diffuse low-entropy inputs.
- Differential Test: It measures the sensitivity of the hash function to small changes in the input, and measures the Hamming distance over the digest. A strong design causes unpredictable, large changes in the output for small input differences.

### C. Results

TABLE I
Resource Usage and Throughput Metrics

| Hash | RAM (B) | ROM (B) | Throughput (B/s) |
|---|---|---|---|
| Proposed | 488 | 4160 | 9324093 |

TABLE II
Avalanche Tests Results

| Input Size (bits) | Samples | Worst Bias (%) |
|---|---|---|
| 24 | 300,000 | 0.623333 |
| 32 | 300,000 | 0.712000 |
| 40 | 300,000 | 0.674000 |
| 48 | 300,000 | 0.769333 |
| 56 | 300,000 | 0.604000 |
| 64 | 300,000 | 0.569333 |
| 72 | 300,000 | 0.679333 |
| 80 | 300,000 | 0.794000 |
| 96 | 300,000 | 0.621333 |
| 112 | 300,000 | 0.782000 |
| 128 | 300,000 | 0.715333 |
| 160 | 300,000 | 0.672000 |
| 512 | 300,000 | 0.812000 |
| 1024 | 300,000 | 0.672667 |

TABLE III
Combination Test Results

| Test Type | Collisions (Act./Exp.) | Worst Bias (%) |
|---|---|---|
| Lowbits | 702/668.6 | 0.029 |
| Highbits | 672/668.6 | 0.046 |
| Hi–Lo | 17383/17322.9 | 0.031 |
| 0x80000000 | 8084/8186.7 | 0.027 |
| 0x00000001 | 8183/8186.7 | 0.047 |
| 0x8000000000000000 | 8229/8186.7 | 0.034 |
| 0x0000000000000001 | 8229/8186.7 | 0.044 |
| 16-bytes [0–1] | 8207/8186.7 | 0.021 |
| 16-bytes [0–last] | 8209/8186.7 | 0.048 |
| 32-bytes [0–1] | 8075/8186.7 | 0.057 |
| 32-bytes [0–last] | 8082/8186.7 | 0.020 |
| 64-bytes [0–1] | 8172/8186.7 | 0.051 |
| 64-bytes [0–last] | 8132/8186.7 | 0.036 |
| 128-bytes [0–1] | 8283/8186.7 | 0.039 |
| 128-bytes [0–last] | 8023/8186.7 | 0.035 |

### D. Discussion

In this work, we evaluated the algebraic behavior at two levels. First, we analyzed the 4-bit substitution box and computed the algebraic degrees $[3, 3, 3, 2]$ for its four output bits, which demonstrates the nonlinearity of the box locally. Second, we evaluated the full 352-bit permutation, where the substitution box is combined with addition, rotation, XOR, and diffusion, and we found that the algebraic degree reaches its maximum after two rounds.

TABLE IV
ZEROES TEST RESULTS FOR 204,800 KEYS

| Metric | Expected | Actual |
|---|---|---|
| 32-bit collisions | 4.9 | 4 |
| High bits (23) worst | 2479 | 2534 |
| Low bits (26) worst | 312 | 330 |
| Worst bias | - | 0.287% |

TABLE V
DIFFERENTIAL TEST RESULTS

| Differentials Test | Tests Run | Collisions (Act./Exp.) |
|---|---|---|
| Up to 5 bits | $8.30 \times 10^9$ | 3 / 1.93 (ignored) |
| Up to 4 bits | $1.10 \times 10^{10}$ | 0 / 2.57 |
| Up to 3 bits | $2.80 \times 10^9$ | 0 / 0.65 |

Additionally, minimizing RAM and ROM footprint is essential on embedded platforms. The proposed hash function, evaluated alongside Xoodyak [4] and PHOTON-Beetle [5] on the same machine, uses 488 bytes of RAM, about 18% less than Xoodyak (600 bytes) and 47% less than PHOTON-Beetle (936 bytes), and 4,160 bytes of ROM, about 17% less than PHOTON-Beetle (5,050 bytes) and 43% less than Xoodyak (7,296 bytes). It achieves 9.32 MB/s of hashing throughput while maintaining this small memory footprint; Xoodyak and PHOTON-Beetle provide higher throughput but at the cost of larger RAM and ROM usage. Thus, the proposed hash offers a balanced speed–resource trade-off for embedded and other resource-constrained systems.

For the diffusion and statistical tests, the proposed hash shows strong performance. The Avalanche Test reports worst-case per-bit bias between 0.569% and 0.812% across input sizes of 24–1024 bits, indicating that single-bit input changes propagate close to the ideal 50% output change and that input differences are thoroughly mixed, preventing clustering and promoting an approximately uniform spread of changes across the digest. The Combination Test confirms robust behavior under structured, pattern-heavy inputs, with outputs remaining well distributed, and the Zeroes Test shows that zero-dominated data does not degrade the hash, with no detectable bias in the output distribution. Moreover, across billions of test inputs, no collisions were observed in the Differential Test campaigns for 128-bit and 256-bit input lengths, indicating that small input differences (1–5 bit flips) are spread broadly across the digest without collision in these runs. Overall, these results indicate strong diffusion and near-independent output-bit behavior, reducing output clustering and enhancing resistance to collisions.

## V. CONCLUSION

In this paper, we present a lightweight 256-bit hash for resource-constrained platforms. The design employs a sponge construction and utilizes a 16-round permutation over a 352-bit state. Our evaluation indicates that our software achieves practical speed with an implementation footprint that is significantly smaller than that of other existing algorithms. Diffusion and statistical tests show consistent bit mixing, stable behavior on structured and zero-dominated inputs, and strong sensitivity to small input changes.

## REFERENCES


[1] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "On the Indifferentiability of the Sponge Construction," in *Proc. EUROCRYPT 2008*, LNCS. Springer. DOI: 10.1007/978-3-540-78967-3_11.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Cryptographic Sponge Functions*. Keccak Team, ver. 0.1. [Online]. Available: https://keccak.team/files/CSF-0.1.pdf.

[3] Guo, Jian, Thomas Peyrin, and Axel Poschmann. "The PHOTON family of lightweight hash functions." Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31. Springer Berlin Heidelberg, 2011

[4] Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., & Van Keer, R. (2020). Xoodyak, a lightweight cryptographic scheme. IACR Transactions on Symmetric Cryptology, 2020(S1), 60-87. https://doi.org/10.13154/tosc.v2020.iS1.60-87

[5] Bao, Z.; Chakraborti, A.; Datta, N.; Guo, J.; Nandi, M.; Peyrin, T.; Yasuda, K. PHOTON-Beetle Authenticated Encryption and Hash Family. Submission to NIST LWC Project. 2021

[6] "SUPERCOP: System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives," [Online]. Available: https://bench.cr.yp.to/supercop.html

[7] "SMHasher: A test suite for hash functions," GitHub repository, [Online]. Available: https://github.com/rurban/smhasher