**Discover** Applied Sciences

Research

# Statistical Fault Analysis of TinyJambu

**Iftekhar Salam**[1] · **Janaka Alawatugoda**[2,3] · **Hasindu Madushan**[4]

Received: 17 January 2024 / Accepted: 29 January 2024
Published online: 06 February 2024


**Abstract**
The resource-constrained IoT devices have limited resources such as processing power, memory, and battery capacity. Therefore it is challenging to adopt traditional cryptographic algorithms on them. In order to find a solution, the National Institute of Standards and Technology (NIST) initiated the Lightweight Cryptography (LWC) competition to standardize cryptographic algorithms for resource-constrained devices. The primary aim of our work is to implement and analyse selected finalist algorithms from the NIST competition using modern cryptanalysis techniques, with a focus on statistical fault attacks. Traditional analysis methods, such as linear and differential analysis, were not prioritized as most finalist algorithms have established defences against these methods. We implemented six of the selected finalist algorithms from the competition: Ascon, Elephant, GIFT-COFB, ISAP, TinyJambu, and Xoodyak. We chose TinyJumbu for statistical fault analysis because of its attractiveness, compact block size, and provision of a more lightweight keyed permutation.

**Highlights**

1. We reviewed the constructions and modes of operations in the NIST LWC finalists.
2. We implemented six finalist algorithms from the LWC competition: Ascon, Elephant, GIFT-COFB, ISAP, TinyJambu, and Xoodyak.
3. We present our results on statistical fault analysis against the TinyJambu algorithm.

**Keywords** Statistical fault analysis · TinyJambu · LWC · Cryptographic Algorithms · Resource-constrained IoT

## 1 Introduction

During the Internet of Things (IoT) age, diminutive, affordable, and energy-efficient devices serve a crucial role across many applications. To illustrate, a cardiac implant embedded within a patient's body necessitates dimensions of compactness while upholding prolonged operation without the need for frequent battery replacements or recharges. These devices are commonly recognized as resource-constrained, underscoring the paramount importance of security. The crux of the matter lies in the limited resources inherent to these devices, potentially leading to performance challenges when

✉ Janaka Alawatugoda, jalawatugoda@ra.ac.ae | [1]School of Computing and Data Science, Xiamen University Malaysia, 43900 Sepang, Selangor, Malaysia. [2]Research & Innovation Centers Division, Rabdan Academy, P.O. Box 114646 Abu Dhabi, United Arab Emirates. [3]Institute for Integrated and Intelligent Systems, Griffith University, Nathan, Queensland 4111, Australia. [4]Department of Computer Engineering, University of Peradeniya, Peradeniya 20400, Central Province, Sri Lanka.

executing conventional cryptographic algorithms. Hence, in recent times, scholars have been dedicated to formulating lightweight cryptography and exploring a spectrum of efficient cryptographic technologies.

The lightweight cryptographic algorithms are efficient even in low-end devices such as smart access cards, car keys, or RFID tags. Efficiency comes in different forms. They encrypt/decrypt much faster than standard cryptography algorithms and use less memory and storage. Usually, lightweight features are achieved using fewer numbers, rounds, or simpler structures. However, most lightweight cryptography algorithms have different and modern techniques to maintain high security. Furthermore, some lightweight cryptography algorithms are fast in hardware, and some are fast in software. However, a thorough analysis of these algorithms is required since they are still not as mature as standard cryptographic algorithms.

### 1.1 The NIST lightweight cryptography competition

National Institute of Standards and Technology (NIST), a globally recognized agency for fostering innovation and industry competition, initiated a transformative path in cryptography through milestones like the Data Encryption Standard (DES). Their ongoing venture, the Lightweight Cryptography (LWC) project, aims to standardize efficient cryptographic algorithms for devices with limited resources. The LWC project's inception was marked by the Lightweight Cryptography Workshop in 2015, gathering insights on securing such devices. Following this, NIST hosted a second workshop in 2016 and progressed by issuing the NISTIR 8114 Report on Lightweight Cryptography in 2017, along with a draft whitepaper outlining profiles for the lightweight cryptography standardization process.

In 2018, NIST initiated the Lightweight Cryptography competition, receiving 57 algorithm submissions by the deadline. Initially, 56 algorithms were chosen for the first round, and 32 progressed to the second round in 2019. Following evaluations in the second round, ten finalist algorithms were selected in 2021: Ascon, Elephant, GIFT-COFB, Grain128-AEAD, ISAP, PHOTON-Beetle, Romulus, SPARKLE, TinyJambu, and Xoodyak. NIST employs both internal assessments and third-party analyses to determine the winners.

All algorithms must enable Authenticated Encryption with Associated Data (AEAD) capability. Submissions can encompass either a single algorithm or an algorithm family, with an optional inclusion of a hash function. Algorithm designers need to specify actual values for adjustable parameters. A security strength declaration must accompany each variation of the algorithm. Additionally, designers are required to furnish references to third-party analyses of the algorithms.

### 1.2 Our contribution

The primary aim of this study is to implement and analyse selected finalist algorithms from the NIST lightweight cryptography competition using modern cryptanalysis techniques. This analysis is crucial to assess their security against advanced attack methods. Implementing these algorithms also aids in identifying security weaknesses and performance challenges, particularly on resource-constrained devices.

This research contributes to the final evaluation of the NIST competition, potentially leading to the establishment of the world's first standardized lightweight cryptography algorithm. Leveraging third-party analyses, NIST's evaluation benefits from this study's findings. Six algorithms were chosen for implementation due to their diverse structures and modes of operation. The study focuses on statistical fault attacks, as it offers novel approaches with potential against lightweight cryptography algorithms. We did not prioritize traditional analysis techniques like linear and differential analysis because most finalist algorithms have defences against them. We conducted a statistical fault attack analysis against TinyJambu. Our choice to implement statistical fault analysis on TinyJambu was mainly due to its compact block size, attractiveness, and provision of a more lightweight keyed permutation.

### 1.3 Organization of the paper

The paper is organized as follows: Sect. 2 discusses the constructions and modes of operations utilized in the six chosen algorithms. Further, Sect. 2 provides a concise discussion about the LWC finalist algorithms for implementation. Section 3 outlines the overarching methodology for the statistical fault attack. Additionally, it details the specific approach adopted for conducting these attacks, building upon the general attack methodologies. Section 4 details the implementation of the attack methodology. Moreover, the experimental setup for the cryptanalysis is explained in section 4. Finally, Sect. 5 provides the conclusion of the paper.

## 2 Constructions of the NIST LWC finalists

In the beginning, we will delve into the various structures and techniques that have been employed in the six algorithms that have been selected. Subsequently, we will proceed with a brief and informative discussion about the finalist algorithms of the Lightweight Cryptography Competition (LWC) that have been singled out for implementation.

### 2.1 Constructions and primitives utilized for the LWC finalists

Before delving into the intricacies of lightweight cryptographic algorithms, it is crucial to have a clear understanding of their constructions and primitives. These fundamental building blocks form the foundation of these algorithms and are responsible for their efficient and secure operation. Therefore, gaining a comprehensive understanding of these components is essential before proceeding further. We discuss the constructions and primitives used in various lightweight cryptographic algorithms in the below sections.

#### 2.1.1 Sponge

The Sponge construction introduced by Bertoni et al. [1] has emerged as a prominent and widely employed framework within the NIST Lightweight Cryptography (LWC) competition's finalists. While initially designed for secure hash functions, the Sponge construction exhibits qualities conducive to authenticated encryption, especially in lightweight contexts. It follows an iterative approach akin to a random oracle but is susceptible to inner-state collisions. Characterized by simplicity and efficiency, the Sponge function incorporates a variable-length input and produces an endless output. Operating with a fixed-size state and two core processes-absorbing and squeezing-it divides the state into confidential and output-generation segments. In each iteration, the function absorbs a set input length, transforms the entire state, and then squeezes out the output. As asserted by Bertoni et al. [1], when utilized as a hash function, the Sponge construction demonstrates collision complexities of $2^{(c+3)/2}$ for inner collisions and $2^{(c+3)/2}$ for output collisions, with $c$ representing the inner state size. Furthermore, the authors claim security against pre-image and second pre-image attacks. Bertoni et al. also elaborates on adapting the Sponge construction for authenticated cryptography and stream ciphers, achieved by embedding the secret key within the inner state to safeguard it from direct exposure in the output, with increased key secrecy as the capacity grows.

#### 2.1.2 Duplex

Duplex [2], a cryptographic construction related to the Sponge construction, is designed as a more streamlined way of generating message digests (MACs) compared to Sponge-based designs. It notably serves as one of the earliest complete authenticated encryption constructions for permutation-based ciphers without necessitating a key scheduling algorithm. With applications extending to pseudo-random bit generation, it's utilized in well-known lightweight cryptographic algorithms like Ascon. The Duplex function accepts plaintext, data header (associated data), and a key to yield ciphertext and a MAC tag via a process known as SpongeWrap. In the decryption process (unwrapping), the input comprises the wrapping key, ciphertext, associated data, and tag, yielding the plaintext if the tag is valid or an error if not. Duplex maintains the state between encryption (or decryption) calls, termed "duplexing", a departure from the standard Sponge construction. Security against generic attacks is readily provable, offering flexibility in choosing the bit rate and padding function. However, downsides include the inability to execute duplex-based encryption in parallel due to its reliance on the previous state, and a lack of inherent support for nonces, resulting in identical output for the same message sequence.

#### 2.1.3 SPONGENT

SPONGENT [3], built upon the Sponge construction, functions as a cryptographic primitive in Elephant and various other NIST Lightweight Cryptography (LWC) algorithms. Featuring thirteen variants such as SPONGENT-128/256/128, SPONGENT-160/320/160, SPONGENT-224/448/224, and SPONGENT-88/80/8, its permutation draws connections from PRESENT [4]. The permutation involves an ICounter updated via a linear feedback shift register (LFSR), a 4-bit S-box substitution

layer, and a permutation layer. The authors assert that SPONGENT's design curbs the linear hull effect, PRESENT's primary vulnerability. Claiming an upper bound of $2^{-28}$ for differential characteristic probability across six rounds, it resists differential cryptanalysis and linear attacks. SPONGENT offers robust security against collision and pre-image attacks when employed as a hash function, with an additional advantage of being efficiently implementable in hardware with minimal gate area, even as small as 738 GE for the SPONGENT-88/80/8 variant.

#### 2.1.4 KECCAK

KECCAK [5] forms a family of Sponge-based cryptographic algorithms widely employed in lightweight cryptographic designs like ISAP and Elephant. Offering a simple and versatile structure, KECCAK encompasses numerous variations. Its KECCAK-f permutation is optimized for hardware utilizing Single Instruction, Multiple Data (SIMD) and CPU pipelining, taking a bit-oriented and parallelizable approach. Comprising seven variants with state sizes ranging from 25 to 1600 bits and diverse round counts contingent on state size, the permutation integrates multiple mappings that furnish differential propagation, correlation, and other security-enhancing traits. This construction withstands various cryptanalysis approaches, such as higher-order differentials, impossible differentials, differential-linear attacks, boomerang attacks, and integral cryptanalysis, attributed to its intrinsic properties. The asymmetry within different rounds additionally thwarts slide attacks.

### 2.2 Review of selected NIST LWC finalist algorithms

Ten finalist algorithms were selected in 2021 for the NIST LWC project. These include Ascon, Elephant, GIFT-COFB, Grain128-AEAD, ISAP, PHOTON-Beetle, Romulus, SPARKLE, TinyJambu, and Xoodyak. Let us review the six selected NIST LWC finalist algorithms: Ascon, Elephant, GIFT-COFB, ISAP, TinyJambu, and Xoodyak.

#### 2.2.1 Ascon

Ascon [6, 7], a block cipher encryption with two main variants-Ascon-128 and Ascon-128a-stands as one of the final algorithms in the CAESAR competition. A new variant, Ascon-80pq, has been developed to resist quantum key-search attacks. The submission also introduces hash functions, Ascon-HASH and Ascon-HASHA. Rooted in a 320-bit permutation structure, Ascon aims for efficient performance in both hardware and software. With fixed parameters, Ascon-128 and Ascon-128a employ a 128-bit key, nonce, and tag, differing in their data block sizes of 64 and 128 bits, respectively. Operating in a duplex mode, specifically MonkeyDuplex, both encryption and decryption culminate in generating the Message Authentication Code (MAC).

Security claims include 128-bit security across all Ascon variations, even with accidental nonce reuse. Key recovery complexity is estimated at $2^{96}$ for Ascon-128a and $2^{128}$ for Ascon-128. Ascon-80pq bolsters resistance against Grover's algorithm-based key search with its larger key size. Ascon's hash functions exhibit security against collision, pre-image, length extension attacks, and second pre-image attacks. Timing attacks are thwarted by bit-sliced S-boxes. Notable performance features include operations based on 64-bit words and bit-wise operations, leading to a throughput of $4.9 - 7.3$ Gbps on less than 10 kGE hardware. Also, the lack of inverse operations enhances its efficiency.

The specification delves into design rationale, highlighting the advantages of Ascon's Sponge-based structure. It offers guidance on variant selection, outlines known attacks, and provides analysis of differential, linear, and algebraic properties. However, a gap remains in testing practical side-channel attacks. An implementation section covers performance metrics but does not extend to low-end devices like 8-bit microcontrollers.

#### 2.2.2 Elephant

The second algorithm in the NIST Lightweight Cryptography (LWC) finalist list is Elephant [8], functioning as a block cipher reliant on permutations and employing the encrypt-then-MAC approach for authentication. Notably, its inverse-free design allows for parallelization in both software and hardware. Elephant encompasses three variations-Dumbo, Jumbo, and Delirium. While Dumbo and Jumbo are grounded in Spongent [3] hashing, Delirium employs Keccak [5] as its core primitive. All three variants utilize Linear Feedback Shift Registers (LFSRs) for masking, with respective block sizes of 160-bits, 176-bits, and 200-bits. The main masking technique involves XORing an LFSR and a permutation (Spongent

or Keccak) with the plaintext, producing ciphertext in encryption and decrypting it with the mask to retrieve plaintext. Specifications encompass permutations and tag sizes (64 bits for Dumbo and Jumbo, 128 bits for Delirium).

The Elephant's analysis encompasses formal multi-user security of its authenticated encryption mode, deriving upper bounds for adversary advantages. The authors assert security levels of 112-bit, 127-bit, and 127-bit for Dumbo, Jumbo, and Delirium, respectively. While the analysis section covers theoretical differential, linear, and integral cryptanalysis on Spongent and Keccak permutations, third-party cryptanalysis details are lacking, particularly concerning side-channel attacks. Although a list of third-party analyses for Spongent and Keccak is provided, more third-party security assessments are necessary for Elephant's overall validation.

#### 2.2.3 GIFT-COFB

GIFT-COFB [9] is a lightweight cryptographic algorithm rooted in the GIFT block cipher. It employs the Combined Feedback (COFB) mode for authenticated encryption, obviating costly inverse operations for decryption. With recommended 128-bit block and tag parameters, GIFT-COFB relies on GIFT-128 [10] as its cryptographic primitive. GIFT-128, a 40-round substitution-permutation network cipher, operates with a 128-bit key. It features an initialization phase, cell substitution, bit permutation, and round key addition, necessitates key scheduling and round constants, and offers a Look-Up Table (LUT) variant for runtime enhancement. Providing clear algorithm steps and test vectors, GIFT-COFB also outlines hardware and software implementations. The hardware version requires 3927 gate equivalents (GE) and 40 cycles to encrypt a plaintext block, with power consumption at 156.3 $\mu$W. While not parallelizable by design, GIFT-COFB employs efficient bit-slice software implementations. The security analysis establishes GIFT-COFB's upper bounds against adversaries seeking to break its authenticated encryption, claiming 64-bit security against IND-CPA and 58-bit security against INT-CTXT. Third-party analyses on GIFT-128 are also highlighted.

#### 2.2.4 ISAP

ISAP [11] constitutes a family of permutation-based authenticated ciphers comprising four variants: ISAP-A-128A, ISAP-K-128A, ISAP-A-128, and ISAP-K-128. ISAP-A-128A and ISAP-A-128 utilize Ascon-p as their primitive, while ISAP-K-128A and ISAP-K-128 employ Keccak-p[400]. Employing encrypt-then-MAC mode, ISAP incorporates a re-keying mechanism for session key generation and features 128-bit key sizes. ISAP-A-128A and ISAP-A-128 boast 320-bit states, whereas ISAP-K-128A and ISAP-K-128 feature 400-bit states. Notably, ISAP's implementation avoids inverse operations.

Security claims encompass 128-bit security across all ISAP variants, with a focus on resisting passive side-channel attacks. The re-keying function mitigates differential power attacks (DPA) and fault attacks, while its Sponge-based structure bolsters resistance against simple power analysis (SPA) attacks. Caution against nonce reuse with the same plaintext is emphasized. Both Ascon-p and Keccak cryptographic primitives within ISAP are backed by proven security.

Designed for lightweight software and hardware implementation, ISAP-K-128A's hardware implementation requires just a 12 kGE gate area in the TSMC65nm cell library. ISAP-A-128A's software implementation exhibits efficiency with 450 cycles per byte on an AVR ATmega328p microcontroller. Both implementations demonstrate robust security against certain fault attacks, including Differential Fault Analysis (DFA), Single Fault Analysis (SFA), and Single Instruction Fault Analysis (SIFA). Furthermore, countermeasures against tag comparison are integrated to enhance security.

#### 2.2.5 TinyJAMBU

TinyJAMBU [12] constitutes an AEAD encryption scheme based on JAMBU, a prominent contender in the CAESAR competition, offering smaller block sizes and a more lightweight keyed permutation. With its main variation, TinyJAMBU-128, utilizing a 128-bit key and state, the keyed permutation features a nonlinear feedback register (NFRS), particularly notable for the parallel execution of 32 updates on a 32-bit CPU. Encryption and decryption phases involve initialization, associated data processing, plaintext/ciphertext processing, and finalization/verification, culminating in a 64-bit tag in the encryption finalization. TinyJAMBU-192 and TinyJAMBU-256 variations feature 192-bit and 256-bit keys respectively.

Security-wise, TinyJAMBU-128 offers 112-bit security, while TinyJAMBU-192 and TinyJAMBU-256 provide 168-bit and 224-bit security, respectively. The scheme thwarts key recovery even when the nonce is misused, ensuring a maximum forgery advantage of $2^{-15}$. Additionally, it boasts robust protection against differential forgery attacks, with the probability of a successful differential key recovery attack as low as $2^{-83}$.

Hardware implementation analysis of TinyJAMBU-128 demonstrates its gate area requirement as 3223 GE for 8 rounds per clock cycle, achieving a throughput of 135 Mbps. Software performance, measured on an ARM Cortex-M4F microcontroller, reveals a code size of 872 bytes and a consumption of 394 cycles per byte when encrypting 16 bytes of data.

#### 2.2.6 XOODYAK

XOODYAK [13] emerges as a versatile cryptographic primitive catering to AEAD encryption, hashing, and pseudo-random bit generation. Centred on the XOODO permutation, it derives inspiration from the Keccak-p permutation and boasts a 384-bit internal state. XOODYAK operates in hash mode and keyed mode, which are activated when initialized with a key. Employing the Cyclist mode of operation with the Cyclist object maintaining the state, the specification reports encryption and decryption cycles per byte as 91.2 and 91.3, respectively, based on ARM Cortex-M0 microcontroller results, accompanied by a code size of 3494 bytes. Performance analyses extend to ASIC and FPGA domains, with ASIC implementation necessitating a minimum gate area of 8101 GE at a 200 MHz frequency and FPGA deployment conducted on a Xilinx Artix-7 board. Despite comprehensive performance details, the XOODYAK specification lacks practical cryptoanalysis information.

## 3 Statistical fault analysis of TinyJambu

Statistical fault attacks were employed to analyse the TinyJambu algorithm in this work. The attacks were executed on self-implemented versions of the algorithm, utilizing C++/C programming to carry out the attacks and simulate them on a personal computer (PC). Python was used for statistical analysis within the statistical fault attack. This section outlines the overarching methodology for the statistical fault attack. Additionally, it delves into the specific approach adopted for conducting these attacks, building upon the general attack methodologies.

Statistical fault attacks were first introduced in Fuhr et al. [14]. One of the main advantages of this attack is that It only requires a set of ciphertexts. In other words, the knowledge of the plaintext input to the encryption function is not required, making it a powerful attack against many types of cryptography algorithms. The only requirement is that all the faulty ciphertexts should be collected using the same key. Usually, random plaintexts are used during the initialization phase of the attack. The first-ever statistical fault attack was launched against the AES algorithm. AES has a substitution and permutation-based structure. The key recovery attack targets the 8th and the 9th round of the AES permutation. Another similar attack, which was able to recover the state of the permutation, targeted the 6th and the 7th rounds.

The main idea for the attack depends on the fact that the output is uniformly distributed since the algorithms try to mimic the random function. The attacker can recover the state or the key bytes by finding the statistical distribution of the faulty bytes and the correlation between the faulty bytes and the output. Let us first consider the fault attack in general.

### 3.1 Fault attacks

Fault attacks involve deliberately injecting faults into a cryptographic system to exploit vulnerabilities and recover the internal state or key of the cipher. Various methods are employed for injecting these faults, including power glitches, timing glitches, laser, light, and electromagnetic (EM) pulses. Power glitches involve abrupt voltage changes, while timing glitches manipulate the clock pulse to affect register values. Laser-based methods focus a laser beam on the chip's location, necessitating chip de-packaging and advanced hardware knowledge. EM pulses target analog blocks via pulse or harmonic injection.

Following fault injection, the analysis phase aims to recover the key or state. This analysis falls into three main categories: difference-based, collision-based, and statistics-based. Difference-based methods exploit the cipher's confusion and diffusion properties, often relying on the difference between outputs with and without faults. Differential Fault Attack (DFA) and Algebraic Fault Attack (AFA) are examples, with DFA suitable for SPN-based ciphers and AFA focusing on algebraic equations. Linear and integral attacks are advanced versions of linear cryptanalysis. Collision-based attacks seek identical outputs for the same input with and without faults. The attacker gathers outputs for specific inputs, introduces faults, and identifies input pairs resulting in identical outputs to recover the cipher's internal state.

### 3.2 Statistical fault attacks

Statistics-based fault attacks are cryptanalysis techniques that leverage statistical methods to recover secret values by introducing faults into a system. This approach offers a notable advantage in terms of low time complexity compared to other methods. Initially proposed by Fuhr et al. [14], the technique was first applied to analyse AES. In this method, the attacker creates an inverse relationship for the round operation that includes key values. A set of output values is then collected for a range of random plaintexts and a secret key with injected faults in the state. These output values exhibit a distribution based on the applied fault injection method. The attack's success depends on achieving a well-biased fault distribution. The distribution is used to make educated guesses about the key. The guessing process is conducted byte by byte, making use of statistical methods for each key byte. The attack is especially efficient since there are only 255 possible byte values to consider. The challenge primarily lies in fault injection. The attack employs three main statistical methods-maximum likelihood, mean Hamming weight, and square Euclidean imbalance-to deduce the correct key byte using fault ciphertext or tags.

1. Maximum likelihood method If the exact distribution of the faulty bytes is known maximum likelihood method can be used to find the correct key byte hypothesis.
2. Mean Hamming weight method In the mean Hamming weight method, the attack does not have to know the exact distribution of the faulty bytes, but the output should be well biased. The attack needs to know whether the distribution is biased towards one or zero.
3. Square Euclidean Imbalance (SEI) method This attack is useful when the attacker only knows the distribution is biased. No knowledge of the shape of the distribution is required. The main idea of this attack is to measure the distance between the actual distribution and the uniform distribution.

In this project, the TInyJambu algorithm was analysed using a statistical fault attack. The structure of the TinyJambu algorithm will be explained below.

### 3.3 Structure of the TinyJambu algorithm

TinyJambu has a keyed permutation based on a non-linear feedback shift register (NFSR). This non-linear feedback shift register updates the 128-bit internal state of the cipher in each round. This update occurs one bit per round. But in a practical implementation, a few bytes (usually 32) will be updated at the same round, decreasing the runtime of the algorithms. However, the output of the permutation will be the same for both methods. The update function is given below:

$$\text{feedback} = s_0^{n-2} \oplus s_{47}^{n-2} \oplus (\sim (s_{70}^{n-2} \& s_{85}^{n-2})) \oplus s_{91}^{n-2} \oplus s_{127}^{n-1} \oplus K_{64}$$

For the attack, the inverse of this relationship should be used to calculate the values of the state bytes after injecting faults.

TinyJambu has four main phases for encrypting: initialization, associated data processing, encryption, and finalization. In the initialization, the state will be loaded and permutated with the nonce. It uses 640 rounds of non-linear feedback shift register updates. When processing the associated, the state is updated using the permutation with 640 rounds, and then the state is updated using the XOR operation. This update occurs block by block. When processing a partial block of associated data, there are a few extra steps to follow to ensure security.

### 3.4 Analysis of TinyJambu against the attack

This attack targets the finalization phase of the algorithm. In the finalization, the tag (or the MAC) will be generated based on the current state. It requires five faults to be injected into five bytes. These faults are injected into the 0th, 6th, 9th, 11th, and 12th bytes. These faults are injected just before the second last round of the permutation. Figure 1 indicates the affected bytes and their connection to the output tag byte. This connection can be represented as the equation.

$$s_0^{n-2} \oplus s_{47}^{n-2} \oplus (\sim (s_{70}^{n-2} \& s_{85}^{n-2})) \oplus s_{91}^{n-2} = s_{127}^{n-1} \oplus K_{64}$$

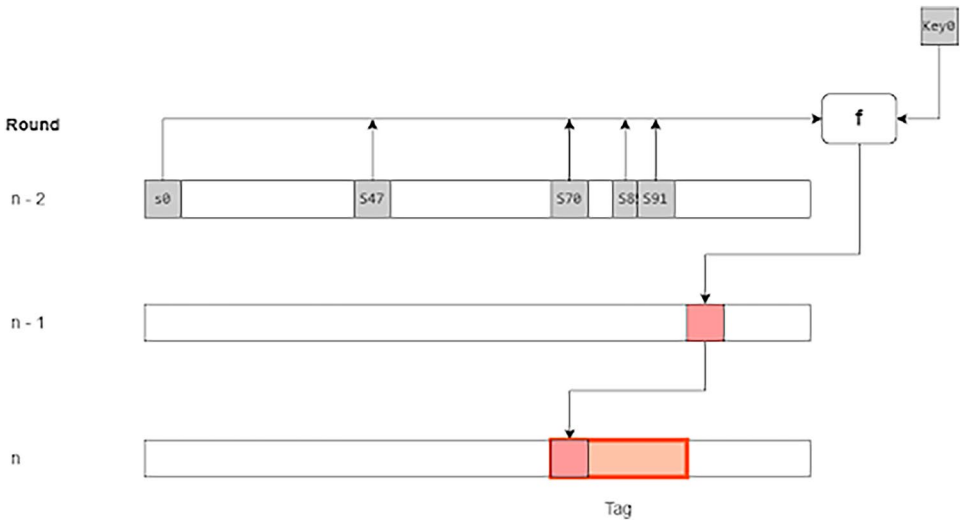


**Fig. 1** Positions of the fault injections

Some of the bytes of the state that contribute to the calculation of the tag value at the last round are not completely affected by the faults. For example, the byte that starts from the 47th bit of the state ($s_{47}$) has a bit that is not affected by any fault. Another way has to be used to find the distribution of these bytes. The rest of the bits that are not affected by the faults can be modelled as uniform random bits. In other words, the probability of any of these bits being one (or zero) is 0.5. Distributions of each bit can be found separately. Then, the probability of each possible value for the complete byte can be calculated using the AND operation. The following example shows how to calculate the probability of the faulty byte starting at the 70th bit.

The probability values of each bit being one is 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.5, 0.5. Here, the fault affects the first six bits, and the last two bytes are uniformly distributed. The probability of the complete byte being the value being 157(10011101) is $0.25 \times (1 - 0.25) \times (1 - 0.25) \times 0.25 \times 0.25 \times 0.25 \times 0.5 \times 0.5 = 0.00054$. After calculating the probabilities of all the values, the distribution of the expression $s_0^{n-2} \oplus s_{47}^{n-2} \oplus (\sim (s_{70}^{n-2} \& s_{85}^{n-2})) \oplus s_{91}^{n-2} \oplus s_{127}^{n-1}$ can be found. When the fault injection is simulated on software, the distribution of the faults is perfectly known. Therefore, the exact distribution of the expression is known. Then the value of the expression is calculated for a set of random plaintexts. Then the maximum likelihood values are calculated for each key byte hypothesis. Then the correct key byte can be selected. The same procedure is followed with the mean Hamming weights method.

## 4 Experimental setup and implementation

In this section, we provide a comprehensive overview of the implementation details, including how various processes are handled at the code level. Additionally, we discuss the experimental setup and results for the statistical fault analysis in detail.

### 4.1 Implementation of the LWC finalist algorithms

In this project, the six algorithms that were chosen, namely Ascon, Elephant, GIFT-COFB, ISAP, TinyJambu, and Xoodyak, were implemented using the C programming language.

Those implementations are specially developed for 8-bit microcontrollers. This means that most of the variables are 8-bit, which makes it easier to handle. The 8-bit microcontrollers have 8-bit registers. If the variables that are operated on are larger than 8-bit, they will use more than one register to store, increasing the run time and memory usage. Resource-constrained devices usually have low memory and storage. Therefore, it is better to deal with 8-bit variables since the program should be space-efficient. Also, operation execution of 8-bit CPUs is done 8-bit wise. If the operands are longer than 8 bits, the run time will be increased. In addition, the execution time when moving around variables will be increased. One example is the index variable in the for loop. Most of the time, for loops will

be used to repeat the rounds, the processing block of data, etc. Usually, the number of repetitions is small. In that case, it is better to use the uint8_t type instead of the regular int variables.

Implementations for all the algorithms have a common interface to use encryption and decryption. This makes it easier to test the functionality and, most importantly, makes it easier to measure the performance of the algorithms. Listing 1 shows the C code for the interface of the algorithm.

```
void encrypt(uint8_t *cipher_text, uint8_t *tag, uint8_t *plain_text,
uint32_t plain_text_len, uint8_t *key, uint8_t *associated_data,
uint32_t adlen, uint8_t *nonce);

uint8_t decrypt(uint8_t *plain_text, uint8_t *tag, uint8_t *cipher_text,
uint32_t cipher_text_len, uint8_t *key, uint8_t *associated_data,
uint32_t adlen, uint8_t *nonce);
```

**Listing 1** Interface for encryption and decryption

The interface for the encrypt function accepts the plaintext, key, associated data, and nonce as the input, while the decrypt function has ciphertext instead of the plaintext and the tag as the input. Encrypt function produces the ciphertext and the tag as the output. The output will also be given as the input argument to the function since the encrypt or decrypt function does not need to allocate dynamic memory, which is unacceptable in low-end microcontrollers. Furthermore, the implementations try to reuse the code as much as possible without duplication. Also, the algorithms and data structures are used to make the implementations more efficient.

### 4.2 Implemnentation of statistical fault attack

The implementation for statistical fault attack is developed using C++ programming language. As discussed in the previous section, the implementation for the TinyJambu algorithm is in C language. The interface for the algorithm is connected to the C++ attack model using a header file.

One of the major concerns when implementing fault attacks in software is to inject fault. To do that, the C codes for the algorithms should be modified. It uses a random number generator to forcefully change the value of the state to simulate the fault injection. This attack uses the following fault model. The probability of the state value being one is 0.25, and the probability of the state value being zero is 0.75. This means the distribution of the fault bytes is biased towards zero by a great margin. However, having such a perfect distribution for the fault bytes may not be possible in real life. Listing 2 shows example codes for the fault injection model.

```
uint8_t generate_faulty_byte()
{
    uint8_t faulty_byte = 0;
    for (int i = 0; i < 8; i++)
    {
        double rand_val = (double)rand() / (double)RAND_MAX;
        faulty_byte |= (rand_val >= 0.75 ? 0x01 : 0x00) << i;
    }
    return faulty_byte;
}
```

**Listing 2** Example code for the fault injection model

In the statistical fault attacks, it is necessary to reverse the final round (which is equal to all the other rounds in the TinyJambu algorithm). In the practical implementation of the TinyJambu, the state is updated to 8-bit. In other words, the eight rounds of the algorithms are executed at once. It makes it faster on an 8-bit microcontroller. However, this makes the logic for the inverse of the round function complicated. The basic logic for implementing the round function depends on the shift and OR operations. This implies that the inverse round function codes should also depend on these operations. The main step is to recover the previous state before the non-linear feedback shift register update. The feedback from the previous round is saved at the first byte of the state. Then, this feedback can be used with a few other bytes from the current state to recover the first byte of the previous state. Listing 3 shows the C code.

```
uint32_t update_state_reverse(uint32_t state[4],
uint8_t key[16], uint8_t round)
{
    uint32_t feedback = state[3];
    state[3] = state[2];
    state[2] = state[1];
    state[1] = state[0];
    uint32_t s47 = (state[1] >> 15) | (state[2] << 17);
    uint32_t s70 = (state[2] >> 6) | (state[3] << 26);
    uint32_t s85 = (state[2] >> 21) | (state[3] << 11);
    uint32_t s91 = (state[2] >> 27) | (state[3] << 5);

    state[0] = feedback ^ s47 ^ (~(s70 & s85)) ^ s91 ^
    ((uint32_t*)key)[round & 3];
    return state[0];
}
```

**Listing 3** Statistical fault attack; reversing the final round of the TinyJambu algorithm

Next, this inverse function can be used to find the value of the faulty byte for a given key and a ciphertext. Thereafter, it uses a random byte sequence generator to generate a set of random plaintexts. The collect_faulty_tag_function returns such a set of random ciphertexts with the fault-injected cipher. This function uses one of the modern functions that can be used in C++, generate. It is a part of the algorithms package in C++. It takes a reference to a function or a lambda expression as an input. This function/lambda should generate some value as the output. The generate function will repeatedly call this function/lambda and put the outputs to a list or similar data structure. This procedure was used to generate a set of ciphertexts for a set of plaintexts. The distribution of the combination of the faulty bytes could be developed. As explained in the section 3, this distribution could be used to guess the correct key bytes. To do that, it was required to implement the functions for calculating the likelihood value, mean Hamming weight, and the Square Euclidean Imbalance values for the distribution.

To analyse the distribution of the combination of the faulty bytes, it is plotted using the Python Matplotlib library. The C program outputs the probability values for each byte value to import the data to the Python program. Then, these values were used as input for the Python program, which plotted the distribution graph. The X-axis represents the byte values, and the y-axis represents the probability value of the corresponding byte.

### 4.3 Analysis of statistical fault attacks on TinyJambu

In this section, we discuss the results obtained from the implementation of statistical fault attacks on TinyJambu. The attack presented in section 3.2 was launched against the TinyJambu algorithm. The first step was to find the distribution for the expression $s_0^{n-2} \oplus s_{47}^{n-2} \oplus (\sim (s_{70}^{n-2} \& s_{85}^{n-2})) \oplus s_{91}^{n-2} \oplus s_{127}^{n-1}$. Figure 2 shows this distribution. However, the attack could not recover the key or part of the key as expected. In other words, the probability of success of the attack is low. The second step was to recover the 8th byte of the key. However, after running the attack many times, it was clear that it was unsuccessful. The correct key byte was not returned by maximum likelihood, mean hamming weight, or square Euclidean imbalance.

The distribution found for the expression $s_0^{n-2} \oplus s_{47}^{n-2} \oplus (\sim (s_{70}^{n-2} \& s_{85}^{n-2})) \oplus s_{91}^{n-2} \oplus s_{127}^{n-1}$ is clearly biased towards 255. One requirement of the statistical fault attack is a biased distribution for the faulty byte or the combination of bytes. Next, the likelihood values were plotted for some random input plaintexts. Figure 3 shows this plot. Each bar represents the likelihood for the corresponding key-byte hypothesis. According to the likelihood values, the correct key byte guess should be the value corresponding to the tallest bar. However, in many cases, it was not correct.

Next, the mean Hamming weight method is used to guess the correct key byte. Figure 4 indicates the plot for each key byte guess's mean hamming weight values. Since the key byte guess is biased toward the maximum value, the maximum mean hamming weight should give the most probable key byte guess. However, as in the maximum likelihood method, the results were not accurate. Furthermore, many values have mean Hamming weights close to the maximum values.

## 5 Conclusions and future works

In cryptography, ensuring the security of algorithms goes beyond theoretical strengths, as even a robust algorithm can become vulnerable due to implementation errors. Therefore, the algorithm should be very carefully handled. Issues such as the Reveal of Unverified Plaintext can cause massive vulnerabilities in the cryptosystems, especially

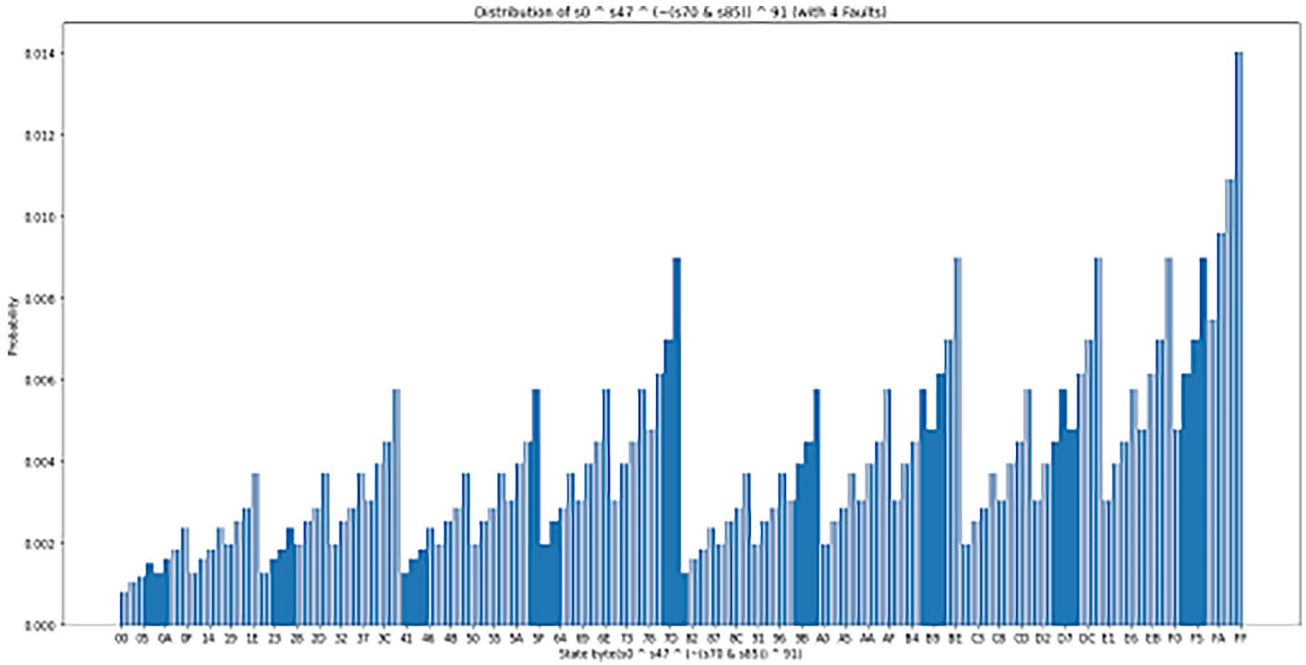


**Fig. 2** The distribution of the expression $s_0^{n-2} \oplus s_{47}^{n-2} \oplus (\sim (s_{70}^{n-2} \& s_{85}^{n-2})) \oplus s_{91}^{n-2} \oplus s_{127}^{n-1}$

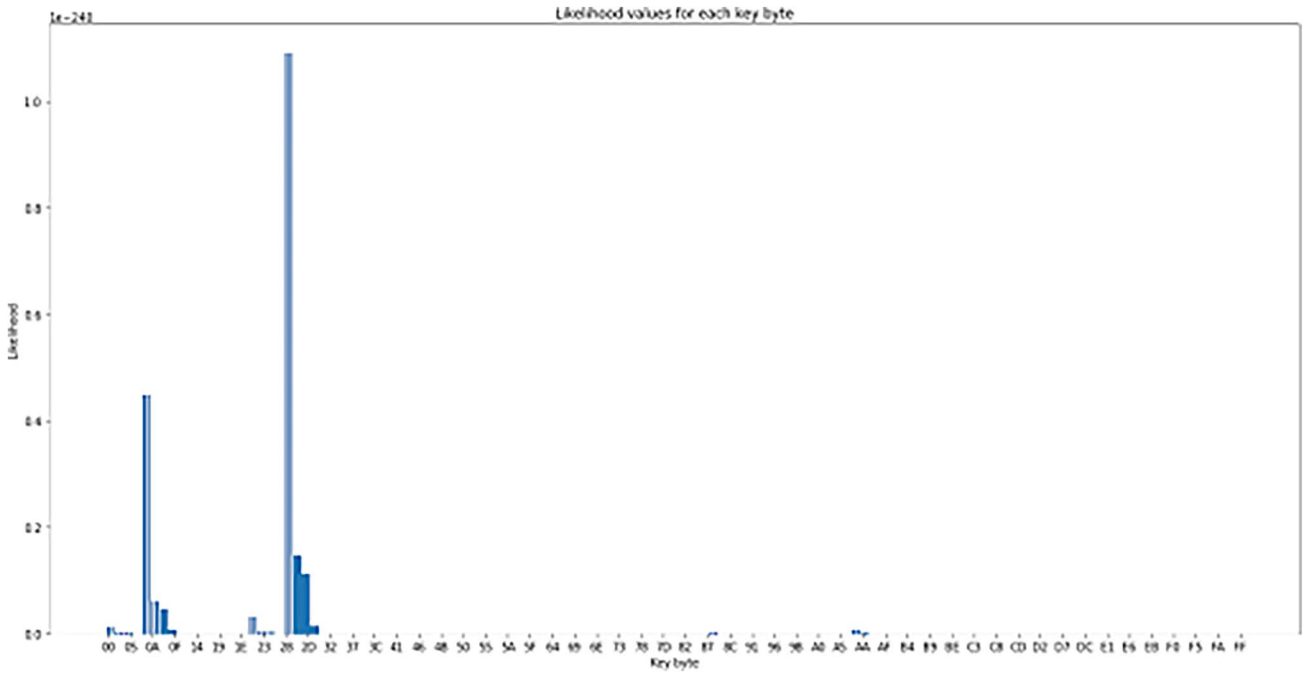


**Fig. 3** The likelihood value for each possible key byte

in the IoT environment. Addressing these challenges requires a proactive approach. Technologies and solutions designed to identify and predict threats in advance play a crucial role in mitigating issues associated with implementation flaws. These measures not only enhance the overall security posture but also contribute to minimizing the impact of potential vulnerabilities, thereby fortifying cryptosystems against a range of threats [15–17].

Nowadays, cryptanalysts are dedicated to preventing statistical relations between elements in lightweight cryptographic algorithms. A notable example is the TinyJambu algorithm, where a structured design appears to counteract

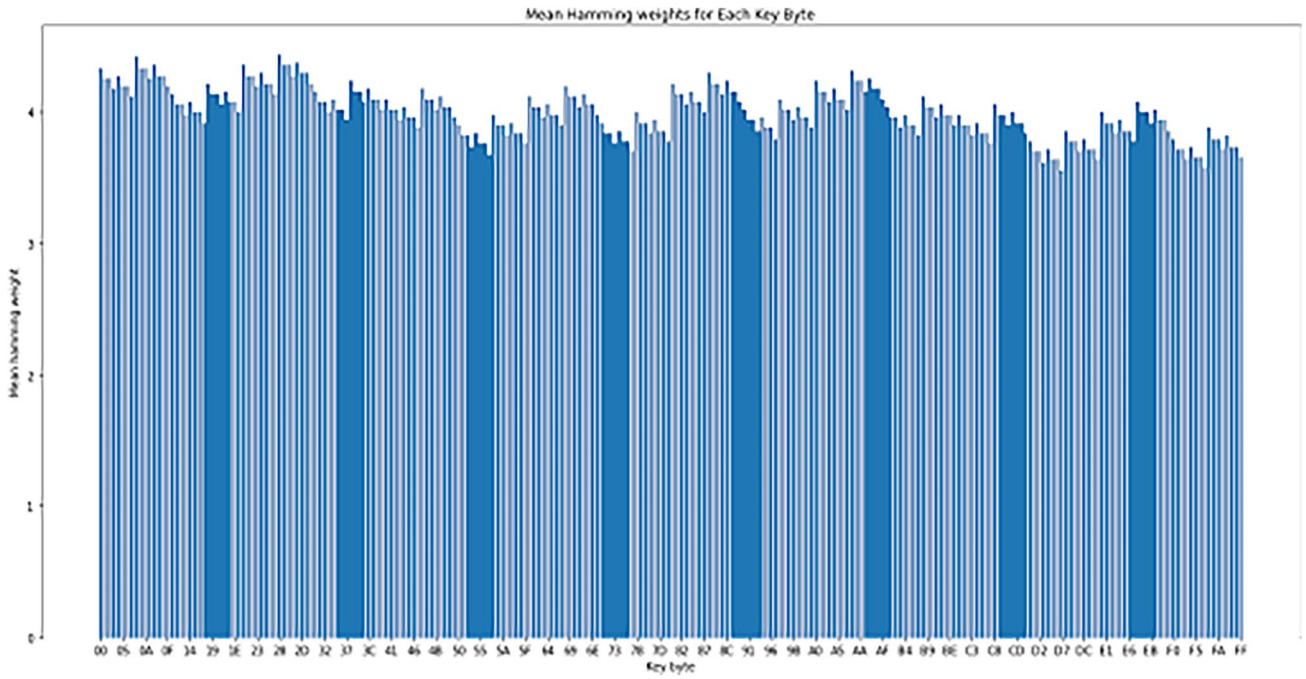


**Fig. 4** The mean Hamming weight value for each possible key byte

statistical attacks, rendering a statistical fault attack unsuccessful. This highlights the importance of considering and countering various attack vectors in cryptographic algorithm design.

Looking ahead, the focus of future research lies in a comprehensive analysis of other LWC competition finalist algorithms, specifically targeting statistical fault attacks. The algorithms earmarked for analysis include Grain128-AEAD, PHOTON-Beetle, Romulus, and SPARKLE. By subjecting these algorithms to rigorous scrutiny, we will aim to glean insights into their resilience against statistical fault attacks, contributing to the ongoing evolution of secure cryptographic practices. This highlights the importance of considering and countering various attack vectors in cryptographic algorithm design.

**Author contributions** All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Hasindu Madushan, Iftekhar Salam and Janaka Alawatugoda. The first draft of the manuscript was written by Hasindu Madushan, and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

**Funding** Janaka Alawatugoda is supported by Rabdan Academy Research Funding. Iftekhar Salam is supported by the Ministry of Higher Education Malaysia through the Fundamental Research Grant Scheme (FRGS), project no. FRGS/1/2021/ICT07/XMU/02/1, as well as the Xiamen University Malaysia Research Fund under Grant XMUMRF/2022-C9/IECE/0032.

**Data availibility** Data sets generated during the current study are available from the corresponding author on reasonable request.

**Code availability** The source codes for the implementation of the six selected finalist candidates can be accessed from: https://github.com/cepdnaclk/e16-4yp-implementation-of-lightweight-cryptographic-algorithms/tree/main/code/lwc-implementations. The source codes for the statistical fault analysis can be accessed from: https://github.com/cepdnaclk/e16-4yp-implementation-of-lightweight-cryptographic-algorithms/tree/main/code/lwc-cryptanalysis/sfa_tinyjambu/src.

## Declarations

**Ethics approval and consent to participate** N/A

**Consent for publication** N/A

**Competing interests** The authors declare that there is no competing interests.

## References

1. Bertoni G, Daemen J, Peeters M, Van Assche G. Sponge functions. In: ECRYPT hash workshop. 2007;2007.
2. Bertoni G, Daemen J, Peeters M, Van Assche G. Duplexing the sponge: single-pass authenticated encryption and other applications. In: Miri A, Vaudenay S, editors. Selected Areas in Cryptography. Springer, Berlin Heidelberg: Berlin, Heidelberg; 2012. p. 320–37.
3. Bogdanov A, Knezevic M, Leander G, Toz D, Varici K, Verbauwhede I. SPONGENT: the design space of lightweight cryptographic hashing. IEEE Transact Comput. 2013;62(10):2041–53. https://doi.org/10.1109/TC.2012.196.
4. Bogdanov A, Knudsen LR, Leander G, Paar C, Poschmann A, Robshaw MJB, et al. PRESENT: an ultra-lightweight block cipher. In: Paillier P, Verbauwhede I, editors., et al., Cryptographic hardware and embedded systems - CHES 2007. Springer, Berlin Heidelberg: Berlin, Heidelberg; 2007. p. 450–66.
5. Bertoni G, Daemen J, Peeters M, Van Assche G. Keccak. In: Johansson T, Nguyen PQ, editors. Advances in cryptology - EUROCRYPT 2013. Springer, Berlin Heidelberg: Berlin, Heidelberg; 2013. p. 313–4.
6. Dobraunig C, Eichlseder M, Mendel F, Schläffer M. Ascon v1.2. Submission to NIST. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf.
7. Dobraunig C, Eichlseder M, Mendel F, Schläffer M. Ascon v1. 2: lightweight authenticated encryption and hashing. J Cryptol. 2021;34:1–42. https://doi.org/10.1007/s00145-021-09398-9.
8. Tim, Ku B, Leuven, Imec-Cosic, Chen YL, Mennink B. Elephant v2. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf.
9. Banik S, , Chakraborti A, Iwata T, Minematsu K, Nandi M, et al. GIFT-COFB v1.1. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/gift-cofb-spec-final.pdf.
10. Banik S, Pandey SK, Peyrin T, Sasaki Y, Sim SM, Todo Y. GIFT: a small present. In: Fischer W, Homma N, editors. Cryptographic hardware and embedded systems - CHES 2017. Cham: Springer International Publishing; 2017. p. 321–45.
11. Dobraunig C, Eichlseder M, Mangard S, Mendel F, Mennink B, Primas R, et al. Isap v2.0. IACR Trans Symmetric Cryptol. 2020;2020:390–416. https://doi.org/10.13154/tosc.v2020.iS1.390-416.
12. Wu H, Huang T. TinyJAMBU: a family of lightweight authenticated encryption algorithms (version 2). https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/tinyjambu-spec-final.pdf.
13. Daemen J, Hoffert S, Mella S, Peeters M, Assche GV, Keer RV. Xoodyak, a lightweight cryptographic scheme. https://repository.ubn.ru.nl/bitstream/handle/2066/221073/221073.pdf?sequence=1.
14. Fuhr T, Jaulmes E, Lomné V, Thillard A. Fault attacks on AES with faulty ciphertexts only. In: 2013 workshop on fault diagnosis and tolerance in cryptography. 2013. 108–118.
15. Agrawal A, Seh AH, Baz A, Alhakami H, Alhakami W, Baz M, et al. Software security estimation using the hybrid fuzzy ANP-TOPSIS approach: design tactics perspective. Symmetry. 2020. https://doi.org/10.3390/sym12040598.
16. Kumar R, Baz A, Alhakami H, Alhakami W, Agrawal A, Khan RA. A hybrid fuzzy rule-based multi-criteria framework for sustainable-security assessment of web application. Ain Shams Eng J. 2021;12(2):2227–40. https://doi.org/10.1016/j.asej.2021.01.003.
17. Kumar R, Khan S, Khan R. Durable security in software development: needs and importance. CSI Commun. 2015;2015(10):34–6.