Panther: A Sponge Based Lightweight Authenticated Encryption Scheme

K. V. L. Bhargavi(✉), Chungath Srinivasan(✉), and K. V. Lakshmy

TIFAC-CORE in Cyber Security, Amrita School of Engineering,
Amrita Vishwa Vidyapeetham, Coimbatore, India
{c_srinivasan,kv_lakshmy}@cb.amrita.edu

Abstract. In the modern era, lots of resource-constrained devices have exploded, creating security issues that conventional cryptographic primitives cannot solve. These devices are connected to an unsecured network such as internet. These lightweight devices not only have limited resources, but also lead to the demand for new lightweight cryptographic primitives with low cost, high performance, low cost of deployment, and effective security outcomes. After reviewing various encryption schemes, designs, and security details, this paper provides a secure cipher Panther, which performs both encryption and authentication using the best components. The design of the Panther is based on a sponge structure using Topelitz matrix and NLFSR (Non-Linear Feedback Shift Register) as the main linear and non-linear components, respectively. Security analysis shows that it is not affected by advanced cryptographic analysis proposed in recent cryptographic literature.

Keywords: Authenticated Encryption (AE) · Sponge construction · Authenticated Encryption with Associated Data (AEAD) · Lightweight cipher

1 Introduction

The rapid explosion of technological development has created many new devices like RFID, IoT, sensor networks and smart cards which makes everything smarter. In addition, these devices operate in a variety of environments. The device connects to the Internet, disrupting communication with the attacker's target and creating various security loop holes. The data should be encrypted and/or authenticated. There are three challenges with these new devices. 1) Secure encryption, authentication only or encryption and authentication 2) Cost of execution, limitation of space and resources 3) Performance latency, power consumption, available capacity and memory.

Lightweight Cryptography: Lightweight ciphers [18] helps us to achieve proper security goals like confidentiality, integrity and authenticity in lightweight environments. Over the past 50 years, symmetric key encryption has come a long way. During 1970's DES followed by RC4 competitions were held. Then, in 2005,

A. Adhikari et al. (Eds.): INDOCRYPT 2021, LNCS 13143, pp. 49–70, 2021.
https://doi.org/10.1007/978-3-030-92518-5_3

the eSTREAM competition was held. At the end of 2008, the SHA3 hash competition and in 2013, the CAESAR encryption competition were held as shown in Fig. 1. There are many symmetric key cryptographic primitives existing in the literature but they cannot meet the security requirements of resource-constrained devices. Traditional encryption technologies like AES, SHA2, SHA3, and RSA are suitable for server and desktop environments, but require a lot of processing and memory. CAESAR and NIST competitions focused on finding new methods to meet security requirements in lightweight environments like ACORN, ASCON [9], Elephant [10], WAGE [1]. The main constraints to consider are power, gate equivalence and cost. Since, the resource constrained devices are operated by batteries for power and the size of hardware available is less. Lightweight cryptography needs to find a balance of trade-off among them while designing cryptographic primitives for the respective application.

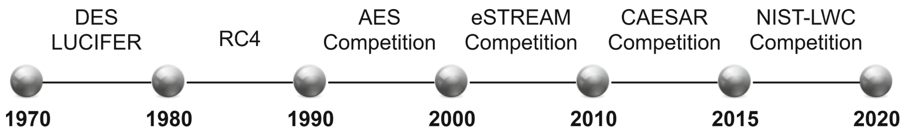


Fig. 1. Development in cryptographic primitives

Authenticated Encryption: Plain encryption differs from Authenticated Encryption (AE). AE provides both confidentiality and authentication, whereas plain encryption just provides confidentiality. The integrity and privacy of confidential messages exchanged across an insecure channel should be guaranteed. The demand for AE arose from the realisation that integrating distinct authentication and confidentiality of the block cipher operating modes in a secure manner will be hard and prone to errors. AE [16] provides confidentiality through encryption, integrity and assurance of message origin will be provided by authentication. This technique can be used to design a lightweight cryptographic algorithm which provides essential security in resource constrained environment. There are few situations where we need to encrypt and authenticate one part of data and the other part of data requires only authentication which is known as associated data, like the header in the packet need not to be encrypted but need to be authenticated and content to be encrypted and authenticated, then we use the technique AEAD. Since, a single approach will provide all the features of security, we are opting for AEAD technique.

The algorithm generates ciphertext and an authentication tag from key, plaintext, and associated data given as input during encryption. When decrypting the ciphertext, the algorithm takes the key, authentication tag and related data, which then returns plaintext or an error, if the tag doesn't match with the ciphertext.

AEAD: Both AE and AEAD are encryption algorithms which provide data confidentiality and also ensures authenticity. The destination address of a network packet is contained in the packet header, which must be publicly accessible in order to send the packet. As a result, the header should be authenticated rather than being encrypted. Associated data must be authenticated before being transmitted. Then we will use authenticated encryption with associated data technique where associated data is not encrypted and the message is encrypted and a tag used to authenticate both associated data and plaintext.

Sponge Based AEAD: The sponge function [15] has a finite internal state that accepts a variable length bit stream as input and outputs a specified length bit stream. It is capable of creating a variety of cryptographic techniques like MAC, masking functions, hash algorithms, pseudo random generator, Authenticated Encryption, password hashes. For generating authenticated tag a sponge function is used as defined in Hash-One [17], but does not provides encryption. Sponge based AEAD can be easily adapted to meet the requirements and do not require key scheduling. In the CAESAR competition on AE, sponge construction is used in 10 of the 57 submissions, and in the NIST LWC competition, sponge construction is utilised in 20 of the 56 submissions. These results indicate that sponge-based constructions will be beneficial in a range of next-generation cryptographic primitives, not just in the proposed SHA-3 hashing standard.

The sponge function employs a permutation or transformation function that works with bits of width b. A finite state with size b bits is partitioned into 2 sections in the sponge construction. The capacity c is the inner part, and the bit rate r is the outer part, where $b = r + c$ bits. Initialization, absorption, and squeezing phases are used to process the given input, which is divided into r bit blocks. The state will be populated in initialization phase using the intial vector (IV) and key as input. During absorption phase, processing of associated data and plaintext will happen. The input data is divided into r bits and then send to the processing. The ciphertext will be obtained while processing the plaintext. In the squeezing phase, the authenticated tag is obtained from the internal state.

1.1 Our Contribution

There is a need for new AE techniques because the conventional techniques are prone to attacks as they are unrealistic in constrained environments. The main constraints in lightweight environments are physical implementation area, power consumption, resources, performance, proper security. An analysis of numerous authenticated encryption schemes, their designs, and security aspects was conducted as part of a literature review. We propose an approach to design a secure cipher using sponge based lightweight AEAD technique named Panther. The inputs for the algorithm are variable length associated data, plaintext and a key. We send these inputs to the sponge and extract the ciphertext of length equal to the plaintext and an authentication tag of given length. We have used panther to transmit real-time traffic in an insecure network like internet. We have also analyzed panther against few cryptographic attacks to prove the strength of the cipher. The results showed that panther is immune to various cryptanalytic attacks which are proposed against other ciphers in the recent cryptographic literature.

Outline of the Paper: We formally present a thorough description of our proposed encryption, Panther, in Sect. 2. In Sect. 3, we describe the benefits of several components used in the proposed cipher. Security analysis of Panther is detailed in Sect. 4 followed by conclusion and future scope in Sect. 5.

2 Proposed Cipher

As indicated in Fig. 2, Sponge is a one-for-all cryptography primitive model created by Bertoni et al. It works in an iterative manner. It comprises of a 328-bit state divided into 82 blocks of four bits each. The sponge's state size splits into two parts r and c. The state's bit rate s_r is 64 bits, collected from the last four blocks (16 bits) in each register P, Q, R, S, while the remaining blocks are used to fill the sponge's capacity s_c.

$$s_r = P15||P16||P17||P18||Q16||Q17||Q18||Q19||R17||R18||R19||R20||S18||S19||S20||S21$$
$$s_c = state - s_r$$

2.1 Notations Used in Panther

All algorithms described in the paper works on the internal state of 328 bits. The 328 bits is splitted into P, Q, R, S registers. The notations mentioned throughout the paper are listed in Table 1.

2.2 Encryption and Decryption Methods in Panther

The sponge-based AEAD technique is one of the most successful way to develop an AEAD algorithm. We utilised sponge construction to create ciphertext and a variable-length authentication tag. The cipher's internal state is made up of four NLFSR's of lengths 19, 20, 21, and 22 over the field F_{2^4} and the field polynomial $x^4 + x^3 + 1$. The sponge's capacity is 264 bits, while the rate is 64 bits. The length of each key and initial vector (IV) is 128 bits.

We use Panther to acquire ciphertext and tag in the encryption algorithm, as illustrated in Fig. 2. The key, IV, plaintext (PT) and associated data (AD) are inputs to the encryption function E, which returns ciphertext (CT) and a tag for authentication, written as $E(key, IV, AD, PT, hashlen) = (CT, tag)$.

The internal state is iterated 92 times during the initialization and finalization phases using state update function, where as the processing of AD, PT, and CT states are updated four times, as illustrated in Algorithm 1.

Figure 3 depicts the decryption and verification process, which is described as $D(key, IV, AD, CT, tag) \in \{PT, Error\}$.

The decryption process in Algorithm 2 is having three phases: The initialization phase is followed by absorption phase, in which we will perform ciphertext processing. The input to this phase is ciphertext blocks which will be xored with rate bits of sponge and produces plaintext as output. The ciphertext block xored with rate bits of sponge will be forwarded as rate part of the state for further state updation function. The last phase is finalization phase, in which the tag is

Table 1. Notations

Notation	Description
Key	Secret key of size 128 bits
IV	Initialization vector of size 128 bits
PT	Plaintext of arbitrary length
CT	Ciphertext of arbitrary length
AD	Associated data of arbitrary length
Error	Error, verification of tag is failed
State	The internal state of sponge of size 328 bits
T_p	Toeplitz matrix
S_b	S-Box
f_i	Feedback polynomial for register i
g_i	Interconnection polynomial for register i
rc	Round constant
F	State update function
$\overline{x}$	Complement of x
s_r, s_c	Bit rate of size 64 bits, capacity of sponge of size 264 bits
P, Q, R, S	Size of internal state is 328 bits which splits into P, Q, R, S
$\oplus$	XOR
$\otimes$	Field multiplication over F_{2^4}
$0^*\|\|1$	Zeros followed by 1
$>>$	Right Shift

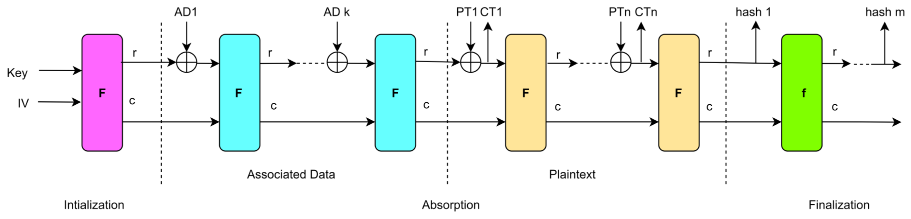


Fig. 2. Sponge based AEAD encryption

Algorithm 1. Encryption

```
1: function ENCRYPTION($key, IV, AD, PT, hashlen$)
2:     $state = 0$
3:     $state = \text{Initialize}(key, IV)$                          ▷ Initialization Phase
4:     $state = \text{AdProcessing}(state, AD)$    ▷ Absorption Phase    ▷ AD Processing
5:     $state, CT = \text{PlaintextProcessing}(state, PT)$          ▷ Plaintext Processing
6:     $tag = \text{Finalization}(state, hashlen)$                   ▷ Finalization Phase
       return $CT, tag$
```

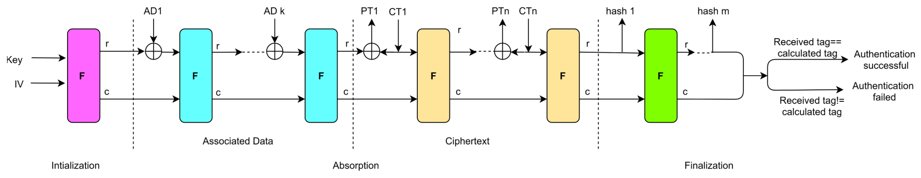


Fig. 3. Sponge based AEAD decryption

verified. If the result of encryption tag is same as the tag obtained during the decryption process, the tag is verified, and the plaintext obtained matches the original message. Otherwise, an error is returned.

Algorithm 2. Decryption

```
1: function DECRYPTION($key, IV, AD, CT, tag$)
2:     $state$ = Initialize($key, IV$)                          ▷ Initialization Phase
3:     $state$ = AdProcessing($state, AD$)     ▷ Absorption Phase     ▷ AD Processing
4:     $state, PT$ = CiphertextProcessing($state, CT$)          ▷ Ciphertext Processing
5:     $decTag$ = Finalization(state, hashlen)                  ▷ Finalization Phase
6:     if $tag == decTag$ then
         return $PT$
7:     else
         return Error
```

2.3 Panther AEAD

To generate ciphertext and tag, we use sponge-based authenticated encryption. The decryption method is the inverse of the encryption method. The data is processed in the absorption phase and does not need to be encrypted. The three phases of authenticated encryption are initialization phase, absorption phase, and finalization phase.

2.4 Initialization Phase

The inputs are key and IV, both of which are 128-bit long. The key is loaded first, then IV. The remaining bits are made up of 64 bits of key complemented bits, seven 1's, and a 0 at the end. Using state update function F, state is updated for 92 times. The initialization process is explained via the Algorithm 3.

2.5 Absorption Phase

The absorption phase partitions both associated data and plaintext into r bit chunks. The input bits are XORed with state bits and interleaved with state update function F. The associated data is organised into k 64-bit blocks. Each

Algorithm 3. Initialization Phase

```
1: function INITIALIZE($key, IV$)
2:     for $i = 0$ to 127 do
3:         $state[i] = key[i]$
4:         $i++$
5:     for $i = 0$ to 127 do
6:         $state[i + 128] = IV[i]$
7:         $i++$
8:     for $i = 0$ to 63 do
9:         $state[i + 256] = \overline{key[i]}$
10:        $i++$
11:    for $i = 0$ to 7 do
12:        $state[i + 320] = 1$
13:        $i{+}{+}$
14:    $state[327] = 0$
15:    $state = \mathrm{F}(state, 92)$                ▷ State update function
       return state
```

of these k blocks is updated four times with function F. Plaintext is divided into n blocks of 64 bits each after processing k blocks. We will obtain ciphertext while processing the plaintext. Once all of the input blocks have been processed, the state is updated 92 times before switching to the squeezing phase. The absorption mechanism is explained in Algorithm 4.

2.6 Ciphertext Processing

In decryption process we will perform ciphertext processing as shown in the Algorithm 5. The received ciphertext is divided into blocks of 64 bits and given to this phase as input. The state bits are XORed with ciphertext. Here, in this step we will retrieve the PT.

2.7 Finalization Phase

Here, the rate part s_r of the internal state of sponge is given as output blocks which is interlaced with four applications of function F. The user's hash length input determines the number of output blocks. The authentication tag is the result of this phase. The Algorithm 6 describes this phase.

2.8 State Update Function

The cipher's internal state is made up of four NLFSR of length 19, 20, 21, and 22 over F_{2^4}. The state is updated 92 times during the initialization and finalization stages, and four times during the intermediate rounds. The feedback polynomial value, the interconnection polynomial value, and the round constants are all calculated here for each NLFSR feedback. The results of the four NLFSR are

Algorithm 4. Absorption Phase

```
1: function AdProcessing($state, AD$)
2:     $adlen$ = length($AD$)
3:     if $adlen\%64! = 0$ then
4:         $pad = 64 - (adlen\%64) - 1$
5:         $AD = AD||1||0^{pad}$
6:     $AD_1, AD_2...., AD_k = AD$                ▷ Dividing into $k$ blocks
7:     for $i = 1$ to $k$ do
8:         for $j = 0$ to 63 do
9:             $s_r[j] = s_r[j] \oplus AD[i][j]$          ▷ Array of rate bits
10:        $F(state, 4)$                          ▷ State update function
    return state
11: function PlaintextProcessing($state, PT$)
12:    $ptlen = length(PT)$
13:    if $ptlen\%64! = 0$ then
14:        $pad = 64 - (ptlen\%r) - 1$
15:        $PT = PT||1||0^{pad}$
16:    $PT_1, PT_2, \cdots, PT_n = PT$            ▷ Dividing into $n$ blocks.
17:    $ct = 0$
18:    for $i = 1$ to $n$ do
19:        for $j = 0$ to 63 do
20:            $s_r[j] = s_r[j] \oplus PT[i][j]$
21:            $CT[ct] = s_r[j]$
22:            $ct + +$
23:        if $i < n$ then
24:            F($state$, 4)                      ▷ State update function
    return $CT, state$
```

Algorithm 5. Ciphertext processing

```
1: function Ciphertext Processing($state, CT$)
2:     $CT_1, CT_2...., CT_K = CT$     ▷ Dividing into $k$ blocks of each size $r$
3:     $n = 0$
4:     for $i = 1$ to $k$ do
5:         for $j = 0$ to 63 do
6:             $PT[n] = s_r[j] \oplus CT[i][j]$
7:             $s_r[j] = CT[i][j]$
8:         if $i < k$ then
9:             F($state$, 4)                      ▷ State update function
    return $state, PT$
```

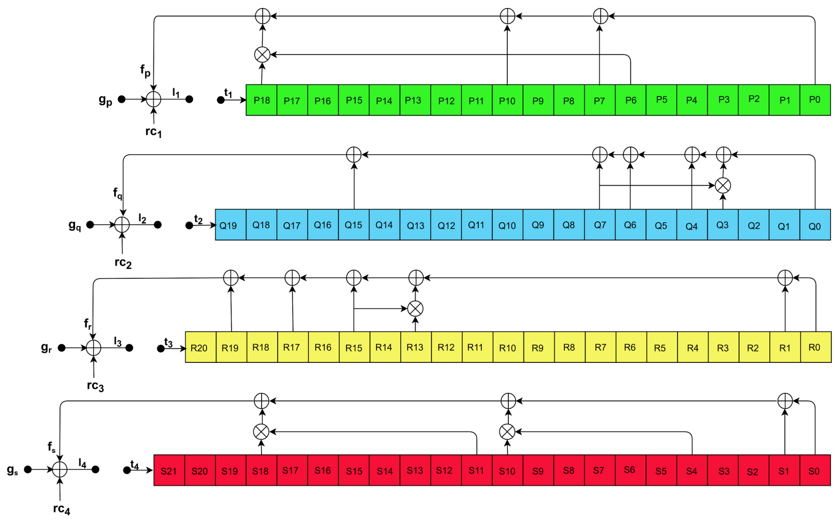
then sent to a filter function, which performs linear and nonlinear operations. Then the registers will be shifted by one block. The $P18, Q19, R20, S21$ blocks are updated with the new result. The polynomial $x^4 + x^3 + 1$ is used to perform field multiplication (Figs. 4 and 5).

Algorithm 6. Finalization Phase

```
1: function FINALIZATION($state, hashlen$)
2:     F($state, 92$)
3:     $t = 0$
4:     if $hashlen\%64 == 0$ then
5:         for $i = 0$ to $(hashlen/64) - 1$ do
6:             for $j = 0$ to 63 do
7:                 $tag[t] = s_r[j]$
8:                 $t = t + 1$
9:             F($state, 4$)
10:    else
11:        for $i = 0$ to $(hashlen/64) - 1$ do
12:            for $j = 0$ to 63 do
13:                $tag[t] = s_r[j]$
14:                $t = t + 1$
15:            F($state, 4$)
16:        for $j = 0$ to $(hashlen\%64) - 1$ do
17:            $tag[t] = s_r[j]$
18:            $t = t + 1$
       return $tag$
```

Algorithm 7. State update function

```
1: function F($state, n$)
2:     for $i = 1$ to $n$ do
3:         $f_p = P_0 \oplus P_7 \oplus P_{10} \oplus P_6 \otimes P_{18}$    ▷ Step 1: Calculate Feedback Polynomial
4:         $f_q = Q_0 \oplus Q_4 \oplus Q_6 \oplus Q_7 \oplus Q_{15} \oplus Q_3 \otimes Q_7$
5:         $f_r = R_0 \oplus R_1 \oplus R_{15} \oplus R_{17} \oplus R_{19} \oplus R_{13} \otimes R_{15}$
6:         $f_s = S_0 \oplus S_1 \oplus S_4 \otimes S_{10} \oplus S_{11} \otimes S_{18}$
7:         $g_p = Q_9 \oplus R_{10} \oplus S_{12}$    ▷ Step2: Calculate Interconnection Polynomial
8:         $g_q = P_4 \oplus R_2 \oplus S_5$
9:         $g_r = P_{12} \oplus Q_{11} \oplus S_{16}$
10:        $g_s = P_{16} \oplus Q_{17} \oplus R_2$
11:        $rc_1, rc_2, rc_3, rc_4 = (0111)_2, (1001)_2, (1011)_2, (1101)_2$    ▷ Step3: Round Constant selection
12:        $l_1 = f_p \oplus g_p \oplus rc_1$    ▷ Step4: Creating inputs for filter function
13:        $l_2 = f_q \oplus g_q \oplus rc_2$
14:        $l_3 = f_r \oplus g_r \oplus rc_3$
15:        $l_4 = f_s \oplus g_s \oplus rc_4$
16:        $[d_1, d_2, d_3, d_4]^T = T_p * [l_1, l_2, l_3, l_4]^T$    ▷ Step5: toeplitz multiplication($T_p$ is toeplitz matrix)
17:        $[t_1, t_2, t_3, t_4]^T = T_p * [S_b[d_1], S_b[d_2], S_b[d_3], S_b[d_4]]^T$    ▷ Step6: S-box($S_b$) followed by toeplitz multiplication
18:        $P >> 1, Q >> 1, R >> 1, S >> 1$    ▷ Step7: Shift the Registers
19:        $P18, Q19, R20, S21 = t_1, t_2, t_3, t_4$
20:        $n + +$
       return $state$
```



f_p
g_p
l_1
rc_1
t_1
P18 P17 P16 P15 P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0
f_q
g_q
l_2
rc_2
t_2
Q19 Q18 Q17 Q16 Q15 Q14 Q13 Q12 Q11 Q10 Q9 Q8 Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0
f_r
g_r
l_3
rc_3
t_3
R20 R19 R18 R17 R16 R15 R14 R13 R12 R11 R10 R9 R8 R7 R6 R5 R4 R3 R2 R1 R0
f_s
g_s
l_4
rc_4
t_4
S21 S20 S19 S18 S17 S16 S15 S14 S13 S12 S11 S10 S9 S8 S7 S6 S5 S4 S3 S2 S1 S0

Fig. 4. f_i function

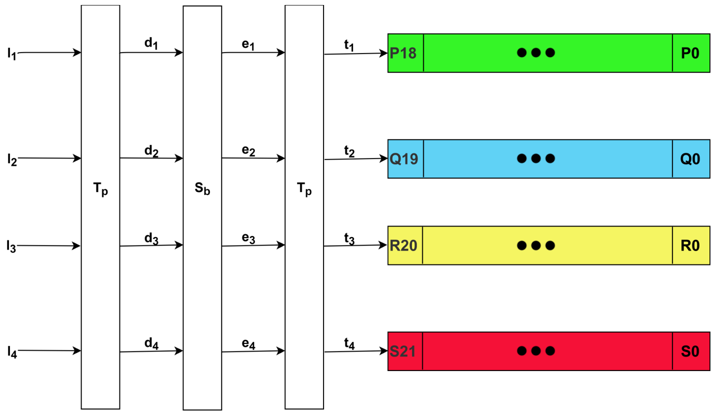


Fig. 5. $l_i \rightarrow t_i$ function

2.9 Toeplitz Matrix

Diffusion layers are constructed using MDS (Maximum Distance Separable) matrices. It is a square matrix with non-singular submatrices. The toeplitz matrices can be used to reduce the complexity of multiplication by using shift and xor operations. The field polynomial should be chosen so that fewer gates are required. As a linear layer, we're utilising the toeplitz matrix. Toeplitz matrices have a constant in descending diagonal from left to right. A typical 4×4 toeplitz matrix looks like

$$T = \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ t_{-1} & t_0 & t_1 & t_2 \\ t_{-2} & t_{-1} & t_0 & t_1 \\ t_{-3} & t_{-2} & t_{-1} & t_0 \end{bmatrix}$$

We don't need to keep all of the matrix element, just the first row and first column will suffice. As a result, we can reduce storage complexity. Diffusion and confusion attributes are included in every state change. A matrix is said to be "circulant" if each row is a left circulant shift of the preceeding row. Toeplitz Matrices that circulate are known as circulant matrices. We are using a 4×4 Toeplitz matrix which is both MDS and circulant.

$$T_p = \begin{bmatrix} 1 & 1 & t & t^{-1} \\ t^{-2} & 1 & 1 & t \\ 1 & t^{-2} & 1 & 1 \\ t^{-1} & 1 & t^{-2} & 1 \end{bmatrix}$$

Consider the primitive element t, which is a root of $t^4 + t^3 + 1$, then the matrix $T_p(t)$ has an XOR count $10 + 4 \times 3 \times 4 = 58$.

2.10 Substitution Box

S-boxes are the fundamental building blocks of practically all modern stream ciphers, and they play a critical role in maintaining security. In a cipher, they're one of the most important nonlinear components. To make the cipher resistant to all types of cryptanalytic attacks, they must be carefully developed or chosen. We have selected a 4×4 S-box from well studied and tested Present [8] cipher.

d_i	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S_b[d_i]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

3 Design Rationale

Our proposed scheme's goal is to achieve the best possible balance of size, speed in terms of software, hardware and security. Because of its well-known hardware efficiency, we used an NLFSR in the state update of sponge. To withstand standard cryptographic attacks and to ensure that every state bit impacts the entire state, the state is updated in a nonlinear fashion using toeplitz followed by S-box followed by toeplitz matrix. The majority of our design is based on well-researched and standard primitives.

Because the message given into the internal state of the sponge, we can get nearly free authentication security. The huge number of iterations in the initialization is primarily intended to ensure that the secret key is better safeguarded when IV is repeated.

The problem is that because we employed nonlinear feedback registers in our sponge construction for state updation, it is difficult to track differential propagation in the state, especially if we wish to provide robust security of 128-bit. Our strategy is to address this issue so that authentication security may be simply assessed by concatenating four different sized linear feedback shift registers. This mechanism used to guarantee, if once exists a difference in the state, the S-box and toeplitz function and tap positions will help to eliminate the differential propagation. We have chosen the suitable tap points for all NLFSR's to guarantee high security.

3.1 Choice of the Mode for Authenticated Encryption

Our authenticated encryption mode design idea is based on the sponge methodology [5]. When compared to other available building approaches, such as various block cipher or stream cipher modes the sponge-based design offers several advantages. The sponge construction has been thoroughly researched, examined, and verified as secure for a variety of applications. Furthermore, the sponge construction is utilised in Keccak, the SHA-3 winner. Other capabilities such as hash, MAC, and cipher can be added as needed. The design is elegant and simple, with a large internal state and no need of key scheduling. We can compute plaintext and ciphertext blocks in parallel without having to wait for the complete message or message length input. Encryption and decryption uses the same permutation which reduces low implementation overhead. We chose the rate bits in a non-sequential manner to make the keystream more unpredictable. Our design includes a 92-round keyed initialization and finalisation phase, which is more robust than prior sponge-based authenticated encryption designs. Though an attacker is able to recover the internal state of sponge while data processing, this does not imply that key will be retrieved completely during trivial forgeries. Bertoni [6] proved that the sponge's security is $O(2^{c/2})$. The security in our cipher is $O(2^{132})$.

3.2 Positioning of the Key and IV

The primary objective of loading the initial values into the state is to keep the different instances separate. In our design, the key is loaded to the first 128 bits of the state, followed by IV for the following 128 bits, and the remaining bits are loaded with the complement of 64 bits from beginning then seven 1's and a zero. Even though both the key and the IV are zeros, the positioning of key, IV prevents the whole state undergoing zero.

3.3 Choice of Rate Positions

The internal state consists of a rate component and a capacity part, with the rate part allowing the attacker to insert messages into the state. The state's rate locations are determined by the security and efficiency of the hardware implementation. From a security standpoint, the chosen rate locations allow the feedback polynomial to disperse the input bits as quickly as feasible after absorbing the message into the state, resulting in quicker confusion and diffusion. Using the shifting property, the length of the process of updating the rate positions is reduced.

3.4 Substitution Layer Selection

A 4×4 S-Box is used in the substitution layer, chosen from the well known PRESENT cipher and was created to enhance diffusion in lightweight environments. The direct outcome for hardware efficiency will be attained by employing a 4×4 S-box which is generally significantly smaller than an 8×8 S-box. Because this S-box has been thoroughly researched and tested, it will be resistant to both differential and linear attacks, as well as well-suited to hardware implementation efficiency. The nonlinear function S-box adds difference noise to the feedback to reduce forgery attempts success rate.

The characteristics of this S-box are as follows: Nonlinearity = 4, differential uniformity = 4. The S-box has a balanced output, and there are no fixed points in the S-box.

3.5 Choice of Linear Layer

A circulant matrix is a matrix in which each row represents one cyclic shift from the preceding row. Because the entire matrix may be produced from the initial row, these matrices are useful in hardware design. MDS matrices has been utilised as the diffusion layer because of the highest diffusion power. For example, the diffusion layer of AES uses MDS matrix. The toeplitz matrix, which is both an MDS and a circulant matrix was chosen. We picked the Toeplitz matrix because it has a lower XOR count, which means less hardware is needed, and we don't have to store all of the elements in the matrix as it is circulant, which decreases storage complexity. For storing the matrix, we just require a single-dimensional array with $(2n - 1)$ items. The diffusion characteristics of Toeplitz, which is an MDS matrix, are well known. Under the irreducible polynomial $x^4 + x^3 + 1$, the minimal value of the XOR count of a 4×4 MDS matrix over F_{2^4} is $10 + 4 \times 3 \times 4$.

4 Security Analysis

4.1 Linear and Differential Cryptanalysis

Linear Cryptanalysis: It attempts to exploit high possibility of linear expressions with ciphertext bits, plaintext bits, and key bits occurring. This is a known plaintext attack, which implies that attacker will be aware of few pairs of plaintext with their corresponding ciphertexts. On the other side, the attacker has no idea what plaintexts and ciphertexts are exposed. It is plausible to suppose the attacker is aware of some random pairs of plaintexts and ciphertexts in many applications and circumstances. The aim of this attack is to figure out expressions with high or low likelihood of occurrence. When a cipher tends or does not hold a linear equation, it has weak randomisation capabilities.

We need to calculate the linear probability bias that is the deviation of linear expression probability from 1/2. If a linear expression holds with probability p_L for some randomly picked plaintexts with associated ciphertexts then probability bias is $p_L - 1/2$. If the value of the probability bias is high then more linear cryptanalysis may be used with some known plaintexts. We take into account the features of the single non linear component: the S-box to create linear expressions. It is feasible to construct linear approximations among pairs of input and output bits in the S-box after the non-linearity features of the S-box have been identified.

The Table 2 depicts all linear approximations of the S-box used in the cipher.

Table 2. LAT table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	+8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	−2	−2	0	0	−2	+6	+2	+2	0	0	+2	+2	0	0
2	0	0	−2	−2	0	0	−2	−2	0	0	+2	+2	0	0	−6	+2
3	0	0	0	0	0	0	0	0	+2	−6	−2	−2	+2	+2	−2	−2
4	0	+2	0	−2	−2	−4	−2	0	0	−2	0	+2	+2	−4	+2	0
5	0	−2	−2	0	−2	0	+4	+2	−2	0	−4	+2	0	−2	−2	0
6	0	+2	−2	+4	+2	0	0	+2	0	−2	+2	+4	−2	0	0	−2
7	0	−2	0	+2	+2	−4	+2	0	−2	0	+2	0	+4	+2	0	+2
8	0	0	0	0	0	0	0	0	−2	+2	+2	−2	+2	−2	−2	−6
9	0	0	−2	−2	0	0	−2	−2	−4	0	−2	+2	0	+4	+2	−2
A	0	+4	−2	+2	−4	0	+2	−2	+2	+2	0	0	+2	+2	0	0
B	0	+4	0	−4	+4	0	+4	0	0	0	0	0	0	0	0	0
C	0	−2	+4	−2	−2	0	+2	0	+2	0	+2	+4	0	+2	0	−2
D	0	+2	+2	0	−2	+4	0	+2	−4	−2	+2	0	+2	0	0	+2
E	0	+2	+2	0	−2	−4	0	+2	−2	0	0	−2	−4	+2	−2	0
F	0	−2	−4	−2	−2	0	+2	0	0	−2	+4	−2	−2	0	+2	0

It is now quite straightforward to create a secure round function. An S-box layer plus a diffusion layer built on MDS code gives a high security margin against differential and linear attacks straight away, even when the number of rounds is minimal.

Differential Cryptanalysis: This is a chosen plaintext attack where the attacker selects inputs and outputs to infer key. It exploits high likelihood of few plaintext differences goes into the cipher's final round. Using this highly likely differential characteristic and by utilising information flowing into the last round of the encryption, we may extract bits from the last layer of sub keys.

It makes use of information from XOR of 2 inputs(input difference) and XOR of 2 outputs (output difference). The attacker selects an input difference x, and has several tuples (x_1, x_2, y_1, y_2), with x_1, x_2 as inputs and y_1, y_2 as outputs. Attacker guesses the key value of the previous round for each pair of y_1 and y_2 and decrypts the XOR at the last but one round. Checks whether the result matches the most likely outcome and keeps track of the number of matches in a frequency table for each key. The correct key will have a high frequency.

The distribution table for differences in S-boxes measures how many pairings with a certain difference of input leads to a certain difference of output. It's a crucial step in finding high-probability transitions and building the differential characteristics, which will be employed later in the attack. The difference distribution table (DDT) of the S-box is provided in Table 3. All iterative differences are copied to the diagonal of the DDT. The S-box has a differential uniformity of 4 and a differential branch number of 3.

Table 3. DDT table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	4	0	0	0	4	0	4	0	0	0	4	0	0
2	0	0	0	2	0	4	2	0	0	0	2	0	2	2	2	0
3	0	2	0	2	2	0	4	2	0	0	2	2	0	0	0	0
4	0	0	0	0	0	4	2	2	0	2	2	0	2	0	2	0
5	0	2	0	0	2	0	0	0	0	2	2	2	4	2	0	0
6	0	0	2	0	0	0	2	0	2	0	0	4	2	0	0	4
7	0	4	2	0	0	0	2	0	2	0	0	0	2	0	0	4
8	0	0	0	2	0	0	0	2	0	2	0	4	0	2	0	4
9	0	0	2	0	4	0	2	0	2	0	0	0	2	0	4	0
A	0	0	2	2	0	4	0	0	2	0	2	0	0	2	2	0
B	0	2	0	0	2	0	0	0	4	2	2	2	0	2	0	0
C	0	0	2	0	0	4	0	2	2	2	2	0	0	0	2	0
D	0	2	4	2	2	0	0	2	0	0	2	2	0	0	0	0
E	0	0	2	2	0	0	2	2	2	2	0	0	2	2	0	0
F	0	4	0	0	4	0	0	0	0	0	0	0	0	0	4	4

Differential Attack Using ML Technique: Following in the traditions of Gohr's [11] work on deep learning-based round reduction cryptanalysis(DL), Baksi [3] discusses a deep learning-based solution for differential attacks on non-Markov ciphers by simplifying the differentiating problem into a classification strategy, they used a range of models with varied widths and numbers of neurons, such as Convolutional Neural Networks (CNN) and MultiLayer Perceptron

(MLP). We utilized Tensorflow, Keras, and the Adam algorithm as optimizer function. We trained the model offline by introducing a large number of input differences (t) into the cipher and storing all of the output differences as training data. The accuracy of ML model training reports is ($\alpha = 0.6) > 1/t$. Hence, we can move on to the next phase. During the online phase, we trained the ML model by treating it like an oracle and putting it to the test with a variety of random input differences. The accuracy ($\alpha = 0.4) < 1/t$ is revealed by the results. We can therefore ensure that the proposed cipher acts as an random oracle and has good random characteristics.

4.2 Slide Re-synchronization Attack

The attack objective is to discover the related keys and initial values. With a probability of 2^{-2}, there appears a related (key', IV') pair for each pair (key, IV). The slide attack [7] when applied to the stream ciphers initialization is known as the slide re-synchronization attack. Attacker takes advantage of this fact and then try to find the related pair

$$(key',\ IV') = ((k'_0, k'_1, \cdots, k'_{127}), (IV'_0, IV'_1, \cdots, IV'_{256}, k'_{127}))$$

with a probability of 2^{-2}, this yields the 1-bit shifted keystream sequence. Their attack strategy is based on two observations:

1. States of the key initialization process are similar to each other.
2. Key set up and keystream generation are similar.

The relationship between (key, IV) and (key', IV') is $key = (k_0, k_1, \cdots, k_{127}) \implies key' = (k_1, k_2 ...., k_{127}, b)$, where $b \in \{0, 1\}$, $IV = (IV_0, IV_1, \cdots, IV_{127}) \implies IV' = (IV_1, IV_2 ...., IV_{127}, 1)$. They showed that stream ciphers which have similar states in the initialization process and using similar mechanisms for key setup and keystream generation are vulnerable to slide resynchronization attacks. While it does not yet produce an effective key recovery attack, it indicates an initialization vulnerability that may be overcome with a small amount of work. Our cipher resists this attack as we are using other NLFSR's data while updating the state. Some of the initialization bits are a mix of complement of key, so it won't undergo slide-resynchronization attack.

4.3 Time Memory Trade-Off Attack

This is one of the general attacks on any cipher. It has mainly two phases

1. Preprocessing Phase
2. Realtime Phase

In preprocessing phase, attacker try to understand the structure of the cipher design and then creates a summary of all the findings in some tables. Attacker invests more time in this phase to gather as much information as possible.

In real time phase, attacker uses the pre-computed tables in the earlier phase to obtain keys in a quicker way. Hellman [13] is famous for the well-known time/memory trade-off exploit. It employs any parameter combination that satisfies $TM^2 = N^2$, $P = N, D = 1$. The best T and M options are determined by the relative costs of these computing resources. Hellman obtains the precise trade-off $T = N^{2/3}$ and $M = N^{2/3}$ for block ciphers by selecting $T = M$. Babbage [2] and Golic [12] separately described the simplest time memory trade-off attack which will be referred as BG attack. They reduced preprocessing time to half ($P = M$) and also the attack time to half ($T = D$).

To prevent tradeoff attacks (TMTO attacks) with l bit key and v bit IV, the internal state size must be at least twice as large as the key size. Then the basic TMTO attack would have complexity at least O(2^l), which is equivalent to the exhaustive key search.

Assume pre-computation time to be 2^p. The attacker observes 2^d frames and mounts an online attack with time 2^t and memory 2^m to recover the secret key K of one frame. Then the TMTO attack would satisfy the following constraints.

$$\begin{aligned} p &= l + v - d \\ t &\geqslant d \\ t + m &= l + v \end{aligned} \tag{1}$$

In the pre-computation phase, we generate keystream sequences, i.e. frames, for 2^m random (Key, IV) pairs and store them in a list in memory. Next in the online phase, we observe 2^d frames using which we are trying to recover the corresponding secret key K. From the birthday paradox, it follows that one of these frames will be broken when

$$m = d = (l + v)/2 \tag{2}$$

From the above Eqs. (1) and (2), it means that the size of IV should be approximately l bits; otherwise the TMTO attack will be faster than the brute-force attack. In our design, $l = v = 128$. Assume $d = 64$, i.e. we could observe 2^{64} frames. Then from (1), we have $p = 192$, $t \geqslant 128$ and $m \leqslant 128$. This implies that the complexity of TMTO attack on our design is at least same as that of brute-force attack.

4.4 Fault Attack

Fault attacks [14] is one of the powerful attacks on any cipher and have shown to be efficient against various stream ciphers. It's uncertain whether or not these attacks are truly possible. Adversary is allowed to flip bits in any of the shift register is one scenario in a fault attack. However, the attacker will not have complete control over the amount of defects or their precise position. A much more powerful assumption is that the opponent can flip precisely one bit, but at a position that he is unable to control. He also has the ability to reset the encryption and introduce a new flaw. If the attacker can repeatedly reset the encryption, each time generating a new defect in a known place that he can estimate from the output difference. Given the more realistic premise that the adversary is unable to control the amount of faults inserted, determining the induced difference from the output differences appears to be more challenging. In the NLFSR, it is feasible to introduce flaws. These flaws will propagate non-linearly in the NLFSR, making their evolution more difficult to forecast. As a result, it appears that inserting faults into the NLFSR is more challenging than LFSR.

4.5 Correlation Attack

The proposed architecture passes all structural tests [19], includes key/keystream correlation test, which evaluates correlation of key and the associated keystream by fixing an IV. The IV/keystream correlation test is the second test, which looks at the correlation of IV and related keystream by fixing a key. The next one is the frame correlation test, which examines the correlation of keystreams with several IV's. The results showed the cipher is resilient against correlation attack.

Correlation attack takes advantage of any correlation between the keystream and the LFSR output in the cipher. The keystream may be thought of as a noisy or distorted version of the LFSR output. The issue of determining the LFSR's internal state is therefore reduced to a decoding problem, with keystream representing the received codeword and LFSR internal state representing the original message. However, unlike LFSR-based ciphers, the recommended state of the ciphers changes in a non-linear form, making it impossible to determine how an attacker should combine these equations to retrieve the condition.

4.6 Cube Attack

We investigated the proposed cipher by selecting 100 cubes and generated 1000 keystream bits corresponding to each cube. For computing the equations we have set the IV vectors to zero at positions other than the cube. Random 128 bit vector is generated and used as key K. Then for each cube and i^{th} keystream bit $F_i(K, IV)$, the polynomial $\sum_{IV \in C} F_i(K, IV)$ is verified for linearity by checking the relation (3) for sufficient number of random key pairs (X, Y).

$$\sum_{IV \in C} F_i(X+Y, IV) = \sum_{IV \in C} F_i(X, IV) + \sum_{IV \in C} F_i(Y, IV) + \sum_{IV \in C} F_i(0, IV) \quad (3)$$

Note that the nonlinearity of the component functions of S-box is 4 and its algebraic immunity is 2. Also for the proposed cipher after 92 rounds of the initialization phase, the degree of the output polynomial is estimated to be very high. As a result, it would be hard for an attacker to gather low-degree relations among the secret key bits.

4.7 Diffusion of Key and IV over the Keystream

This test verifies that every single bit of a key and an IV on the keystream has been diffused. To meet the diffusion condition, every single bit of key and IV must have an equal probability of affecting the keystream. Small differences in key or IV must produce huge change in the keystream. This test begins with the selection of randomly chosen key and IV and produces a length of keystream (L). New keystreams are produced by altering key and IV bits. Create a matrix $(k + v) \times L$ by applying XOR on original keystream and bit changed keystreams. This method is done for N times and resulting matrices are summed up. To evaluate diffusion property for the matrix entries we apply Chi-Square Goodness of Fit test. When N is high, the elements in the matrix follows normal distribution of mean $N/2$, variance $N/4$, resulting in a secure cipher.

We have picked [0–498], [499–507], [508–516], [517–525] and [526–1024] as the category boundaries using these estimated probabilities. To pass the Chi-square goodness of fit test, which has four degrees of freedom and a significance threshold of $\alpha = 0.01$ the resultant chi-square value should be $\chi^2 \leqslant 13.277$. Here, the observed and predicted values are compared, and our design passed with a average chi-square value of $\chi^2 = 4.4$ after analyzing various sample sizes.

4.8 Banik's Key-Recovery Attack

The existence of roughly 2^{30} IV's for each key in Sprout was originally noted out in this work [4], so that the LFSR state becomes entirely 0state following the Key-IV mixing phase. The LFSR does not get feedback from the output bit during the keystream phase, and so remains in the zero state throughout the evolution of cipher weaken the cipher's algebraic structure. Their work was used to report the following: In practical time, key-IV pairings were discovered that produced keystream bits with a period of 80. In the multiple IV mode, a key recovery attack was recorded. The attacker searches keystream for a fixed secret key and several secret IVs, then waits until one of the IVs is queried, causing the LFSR to fall into the all zero state after the key-IV mixing. Because the cipher's algebraic structure was weakened, simple equations on the key bits could be derived, which could be solved to discover the secret key.

To overcome this, we used a constant 1 in the last bits of the state in Panther, so that when the cipher eventually enters keystream generation mode, the NLFSR state is never all zero because the last bits are 1's, and it never falls into the all zero trap.

4.9 Result of NIST Tests

For testing pseudo random sequence generators we can use the Statistical Test Suite SP800-22 of NIST. It consists of 16 tests that check if the bits produced by each encryption method are random. 1000 files are used to perform this test, each of which has a 10^6 bit sequence and corresponds to a distinct key and IV. In the parentheses beside each test, the input parameters used for the test are

Table 4. NIST results

S. No.	Statistical tests	p-value	Proportion
1	Frequency	0 851383	1.0000
2	Block frequency ($m = 100000$)	0.657933	0.9905
3	Forward cumulative sums	0.062821	0.9815
4	Backward cumulative sums	0.678686	0.9900
5	Runs	0.145326	1.0000
6	Longest runs of ones ($M = 1000$)	0.657933	0.9900
7	Non-overlapping template ($m = 9, B = 000000001$)	0.494392	0.9905
8	Universal ($L = 9, Q = 5120$)	0.419021	1.0000
9	Overlapping template ($m = 9$)	0.171867	0.9860
10	Spectral DFT	0.202268	0.9968
11	Approximate entropy ($m = 10$)	0.759756	0.9900
12	Rank	0.699313	0.9853
13	Random excursions ($x = \pm 1$)	0.568055	0.9968
14	Random excursions variant ($x = -1$)	0.534146	0.9880
15	Linear complexity ($M = 500$)	0.319084	0.9853
16	Serial ($m = 16$)	0.514124	1.0000

listed in Table 4. If we take a sample data of 1000 binary sequence size, the minimal passing rate for each statistical test is about equal to 0.980567%. We may conclude from Table 4 that the suggested cipher passed all NIST randomness tests since the p-value uniformity of every test is larger than or equal to 0.01.

4.10 Diffusion of Plaintext over Message Authentication Code

This test verifies that each plaintext bit on the hash has been diffused. Each bit of plaintext should have an equal probability of influencing hash creation to fulfil the diffusion property. Minor changes in the plaintext should cause the hash to alter in an unpredictable way. First, a constant key and IV are chosen in the Diffusion test. We chose the key and IV to be zero, and a 256-bit plaintext is generated. New hashes are then produced by altering each bit of the plaintext. The original plaintext hash values are then XORed with these hash values. A matrix of size $msg \times L$ was created using these vectors. This process is iterated N times and the resultant matrices will be calculated by adding in real numbers. To evaluate the diffusion of matrix entries apply the Chi-Square Goodness of Fit test.

When N is high, the elements in the matrix follows normal distribution with a mean of $N/2$, variance of $N/4$, resulting in a secure cipher. We have taken $N = 1024,\ \ L = 256,\ \ msg = 256$ and the limits for each category with these approximate probabilities are chosen as, [0–498], [499–507], [508–16], [517–525] and [526–1024]. To pass the Chi-square goodness of fit test, which has four degrees of freedom and a significance threshold of $\alpha = 0.01$ the resultant chi-square value should be $\chi^2 \leqslant 13.277$. Here, the observed and predicted values are compared, and our design passed with a average chi-square value of $\chi^2 = 3.83$ after analyzing various sample sizes with random key and IV.

5 Conclusion and Future Work

Because of picking the best components with the least hardware load and adequate security, the suggested sponge based approach can be employed in lightweight environments. To update the state, the design employs NLFSR and a toeplitz matrix. The proposed design has good pseudo randomness and diffusion qualities, which is a criterion for a good design according to the literature. The suggested technique can be used in resource-constrained devices to achieve adequate security goals at a cheap cost. Panther was used to send real-time traffic via an unsecured network. We tested the strength of Panther using a variety of cryptography attacks. The future scope is to test the suggested scheme's resilience and strength employing more cryptanalytic attacks in a realistic manner.

References

1. Aagaard, M., AlTawy, R., Gong, G., Mandal, K., Rohit, R., Zidaric, N.: WAGE: an authenticated cipher. Submission to NIST Lightweight Cryptography Standardization Project (announced as round 2 candidate on August 30, 2019) (2019)
2. Babbage, S.: Improved "exhaustive search" attacks on stream ciphers. In: 1995 European Convention on Security and Detection, pp. 161–166. IET (1995)
3. Baksi, A., Breier, J., Chen, Y., Dong, X.: Machine learning assisted differential distinguishers for lightweight ciphers. In: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 176–181. IEEE (2021)
4. Banik, S.: Some results on Sprout. In: Biryukov, A., Goyal, V. (eds.) INDOCRYPT 2015. LNCS, vol. 9462, pp. 124–139. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26617-6_7
5. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the indifferentiability of the sponge construction. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78967-3_11
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Duplexing the sponge: single-pass authenticated encryption and other applications. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 320–337. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28496-0_19
7. Biryukov, A., Wagner, D.: Slide attacks. In: Knudsen, L. (ed.) FSE 1999. LNCS, vol. 1636, pp. 245–259. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48519-8_18
8. Bogdanov, A., et al.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74735-2_31

9. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: ASCON v1.2. Submission to the CAESAR Competition (2016)
10. Dobraunig, C., Mennink, B.: Elephant v1 (2019)
11. Gohr, A.: Improving attacks on round-reduced Speck32/64 using deep learning. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11693, pp. 150–179. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26951-7_6
12. Golić, J.D.: Cryptanalysis of alleged A5 stream cipher. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 239–255. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-69053-0_17
13. Hellman, M.: A cryptanalytic time-memory trade-off. IEEE Trans. Inf. Theory **26**(4), 401–406 (1980)
14. Hoch, J.J., Shamir, A.: Fault analysis of stream ciphers. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 240–253. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28632-5_18
15. Krishnan, L.R., Sindhu, M., Srinivasan, C.: Analysis of sponge function based authenticated encryption schemes. In: 2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS), pp. 1–5. IEEE (2017)
16. Maimut, D., Reyhanitabar, R.: Authenticated encryption: toward next-generation algorithms. IEEE Secur. Priv. **12**(2), 70–72 (2014)
17. Mukundan, P.M., Manayankath, S., Srinivasan, C., Sethumadhavan, M.: Hash-one: a lightweight cryptographic hash function. IET Inf. Secur. **10**(5), 225–231 (2016)
18. Rohit, R.: Design and cryptanalysis of lightweight symmetric key primitives. University of Waterloo (2020)
19. Turan, M.S., Doganaksoy, A., Calik, C.: Detailed statistical analysis of synchronous stream ciphers. In: ECRYPT Workshop on the State of the Art of Stream Ciphers (SASC 2006) (2006)