SKINNY-AEAD Encryption

Nikshipth Maddugaru*, Shiraz Anwar Khan*
*Department of Electrical and Computer Engineering, North Carolina State University
{nmaddug@ncsu.edu,skhan25@ncsu.edu}

***Abstract*—Our work presents the design and implementation of the Skinny-128-384[1] lightweight block cipher on FPGA for resource-constrained environments. The proposed architecture employs a finite state machine (FSM) for control and a highly modular datapath to execute encryption operations. The design is verified through simulation, achieving accurate results consistent with theoretical expectations. Performance metrics such as latency and throughput are discussed in detail. Furthermore, the proposed architecture is optimized for low resource utilization, making it a suitable choice for applications with strict power and area constraints.**

***Index Terms*—Lightweight Cryptography, Skinny-128-384,Tweakable Block Cipher, Hardware Security.**

I. Introduction

Lightweight cryptography addresses the need for efficient encryption in resource-constrained environments, such as IoT devices. The SKINNY block cipher[2] is a lightweight cryptographic primitive designed to achieve optimal performance on resource-constrained devices such as those used in IoT, mobile systems, and embedded applications. It is structured as a Substitution-Permutation Network (SPN) and offers configurations for different block and key sizes, including `SKINNY-64` and `SKINNY-128`. This flexibility allows it to meet diverse cryptographic needs while maintaining efficiency and minimal hardware or software implementation costs. SKINNY balances security and lightweight design, aiming to resist standard attacks such as linear and differential cryptanalysis.

A key feature of SKINNY is its use of a lightweight tweakable block cipher[3] model, where the tweak can vary dynamically during encryption to provide added security. For instance, `SKINNY-128-384` splits its tweakey into three 128-bit arrays, TK_1, TK_2, and TK_3, processed in conjunction with the plaintext. Its design emphasizes minimal complexity while ensuring resilience against side-channel and differential fault attacks, making it a robust solution for modern cryptographic applications. The Skinny block cipher is a prominent candidate due to its balance of performance, security, and minimal hardware overhead.

This project implements the Skinny-128-384 algorithm on a Xilinx Spartan-7 FPGA, targeting encryption operations under realistic constraints. The implementation aims to evaluate its practicality and performance on hardware.

II. Background and Prior Work

The SKINNY-AEAD and SKINNY-Hash families are extensions of the SKINNY lightweight block cipher, designed for authenticated encryption and hashing applications. SKINNY-AEAD supports various configurations, including SKINNY-128-384 and SKINNY-128-256, optimizing both security and performance in constrained environments. The primary configuration uses a 128-bit key, nonce, associated data, and a message, producing ciphertext and a tag. These variants cater to specific use cases, such as reduced message sizes, making them versatile.

SKINNY-Hash uses the sponge construction and provides secure hash functions with varying security levels. The primary and secondary configurations, SKINNY-128-384 and SKINNY-128-256, ensure strong security and lightweight implementations, making them suitable for resource-constrained devices.

A list of the proposed AEAD schemes (members M1 to M6) and the two hashing algorithms is provided in Table 1. The AEAD members M1, M2, M3, and M4 are paired with the SKINNY-tk3-Hash algorithm, while M5 and M6 are paired with SKINNY-tk2-Hash. These pairings are based on the fact that the constructions in each respective pair are derived from the same variant of the SKINNY tweakable block cipher.

Member	Block Cipher underlying primitive	Nonce bits	Tag bits	Key bits	Hash Function type	Rate bits	Capacity bits
M1 †	SKINNY-128-384	128	128	128	384-bit sponge	128	256
M2	SKINNY-128-384	96	128	128			
M3	SKINNY-128-384	128	64	128			
M4	SKINNY-128-384	96	64	128			
M5 *	SKINNY-128-256	96	128	128	256-bit sponge	32	224
M6 *	SKINNY-128-256	96	64	128			

†: Primary member. *: Do not strictly follow NIST requirements.

TABLE I
Proposed AEAD schemes and hashing algorithms

III. Design

The `SKINNY-128-384` has a block size of n = 128 bit and the internal state is viewed as a (4 × 4) square array of cells, where each cell contains a byte. We denote $IS_{i,j}$ the cell of the internal state located at Row i and Column j (counting starts from 0). One can also view this 44 square array of cells.

The tweakey state is also viewed as a collection of three (4 × 4) square arrays of cells of 8 bits each. We denote these arrays TK_1, TK_2 and TK_3 for `SKINNY-128-384`.

The ciphers receive a plaintext $m = m0\|m1\|\cdots\|m14\|m15$, where the m_i are 8-bit words. The initialization of the internal state of the ciphers is performed by simply setting

$$IS_i = m_i \text{ for } 0 \leq i \leq 15, \text{ i.e.,}$$

$$IS_i = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

as a vector of cells by concatenating the rows. Figure: 1 shows the round function with five distinct operations applied to the data in each round.

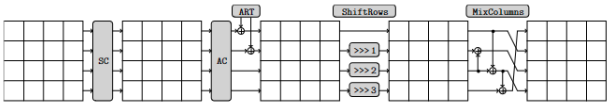

Fig. 1. The SKINNY round function overview

These transformations are executed in a specific sequence within each round to ensure the desired level of confusion and diffusion, which are key principles in cryptographic algorithms.

Skinny-128-384 is an authenticated encryption algorithm supporting a 128-bit plaintext block size and a 384-bit tweakey. It involves multiple rounds of operations: *SubCells*, *AddConstants*, *AddRoundTweakey*, *ShiftRows*, and *MixColumns*.

A. Sub-Cells (SC):

This is the nonlinear substitution layer of SKINNY. Each byte of the cipher's state is replaced using a compact, fixed S-box(Fig 2) inspired by the `PICCOLO` S-box, introducing nonlinearity and ensuring resistance to linear and differential cryptanalysis.

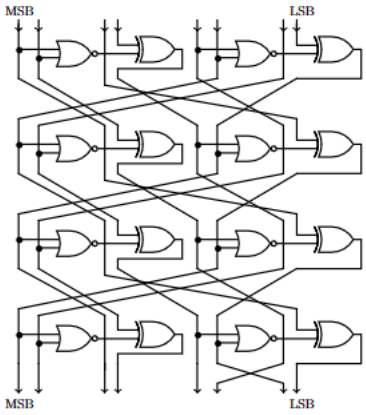

Fig. 2. Construction of the Sbox S8

B. Add Constants (AC):

A 6-bit affine LFSR, whose state is denoted $(rc_5, rc_4, rc_3, rc_2, rc_1, rc_0)$ (with rc_0 being the least significant bit), is used to generate round constants. Its update function is defined as:

$$LSFR \rightarrow (rc_4 \| rc_3 \| rc_2 \| rc_1 \| rc_0 \| rc_5 \oplus rc_4 \oplus 1)$$

The six bits are initialized to zero, and updated before used in a given round. The bits from the LFSR are arranged into a 4 × 4 array and only the first column of the state is affected by the LFSR bits, i.e.

$$\begin{bmatrix} c_0 & 0 & 0 & 0 \\ c_1 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

with $(c_2 = 0x2)$ and $(c_0, c_1) = (0\|\,0\|\,0\|\,0\|\,rc_3\|\,rc_2\|\,rc_1\|\,rc_0, 0\|\,0\|\,0\|\,0\|\,0\|\,0\|\,rc_5\|\,rc_4)$. The round constants are combined with the state, respecting array positioning, using bitwise exclusive-or.

C. Add Round Tweakey (ART):

The first and second rows of all tweakey arrays are extracted and individually correlated bitwise to the internal state of the cipher, with respect to the positioning of the array. More formally, for *i* = 0, 1 and *j* = 0, 1, 2, 3, we have

$$IS_{i,j} = IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \oplus TK3_{i,j}$$

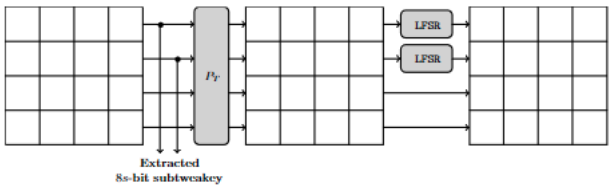

Fig. 3. AddRoundTweakey

Then, the tweakey array is updated as shown in Figure 3. First, a permutation P_T is applied to the cell positions of all tweakey arrays: for all $0 \leq i \leq 15$, we update $TK_z[i] \leftarrow TK_{P_T[i]}$.

$$[0, 1, \ldots, 15] \xrightarrow{\text{PT}} [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7]$$

Finally, every cell in the first and second rows of TK_2 (resp., TK_2 and TK_3) is individually updated using an LFSR. The LFSRs are defined as follows:

$$TK_2 \xrightarrow{\text{LFSR}} (x_6||x_5||x_4||x_3||x_2||x_1||x_0||x_7 \oplus x_5)$$
$$TK_3 \xrightarrow{\text{LFSR}} (x_0 \oplus x_6||x_7||x_6||x_5||x_4||x_3||x_2||x_1)$$

x_7 represents the MSB, and x_0 represents the LSB of each cell.

D. Shift Rows (SR):

A diffusion layer where rows of the cipher's state are cyclically shifted by different offsets. This rearrangement spreads the influence of each byte across the entire state in subsequent rounds.

Similar to AES, the rows of the cipher state cell are rotated. More precisely, the second, third, and fourth cell rows are rotated by 1, 2 and 3 positions to the *right*, respectively.

$$[0, 1, \ldots, 15] \xrightarrow{\text{SR}} [0, 1, 2, 3, 7, 4, 5, 6, 10, 11, 8, 9, 13, 14, 15, 12]$$

E. Mix Columns (MC):

$$M = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

This linear transformation operates on the columns of the state, combining them to further spread the influence of each byte. This step improves diffusion and ensures that small changes in the input propagate throughout the state.

Each column of the cipher internal state array is multiplied with the above matrix **M**.

IV. Implementation

The implementation consists of two main components: the FSM and the datapath. The FSM controls the encryption flow, transitioning through states such as `IDLE`, `LOAD`, `SUB_CELLS`, and `DONE`. The datapath handles transformations like SubCells, AddConstants, and MixColumns.

A. Top-Level Architecture

The implementation includes the following key modules:

- FSM: Controls the operation flow through different states. The states are as follows:
  - `IDLE`
  - `LOAD`
  - `SUB_CELLS`
  - `ADD_CONST`
  - `ADD_ROUND`
  - `SHIFT_ROWS`
  - `MIX_COLUMNS`
  - `DONE`
- Datapath: Performs encryption operations, including the round functions and tweakey transformations.
- Testbench: Provides inputs to the DUT and validates the output against expected results.

B. FSM Design

The FSM controls the sequence of operations in the encryption process, transitioning between predefined states based on conditions. Each state represents a specific phase in the cryptographic algorithm. The states and their transitions are described as follows:

- `IDLE`:
  - This is the initial state.
  - If the `start` signal is asserted, the FSM transitions to the `LOAD` state to begin the encryption process.
  - Otherwise, it remains in the `IDLE` state.
- `LOAD`:
  - Handles loading of input data.
  - If the `isLoadDone` signal is asserted (indicating loading is complete), the FSM moves to the `SUB_CELLS` state.
  - Otherwise, it stays in the `LOAD` state.
- `SUB_CELLS`:
  - Applies the substitution layer (non-linear operation).
  - Automatically transitions to the `ADD_CONST` state.
- `ADD_CONST`:
  - Adds round constants to the state.
  - Automatically transitions to the `ADD_ROUND` state.
- `ADD_ROUND`:
  - Combines round tweakey data with the state.
  - Automatically transitions to the `SHIFT_ROWS` state.
- `SHIFT_ROWS`:
  - Rearranges rows in the state array to enhance diffusion.
  - Automatically transitions to the `MIX_COLUMNS` state.
- `MIX_COLUMNS`:
  - Applies a linear mixing operation to the columns of the state.
  - If the encryption is not complete (`!isEncDone`), the FSM loops back to the `SUB_CELLS` state for the next round.
  - If the encryption is complete, it transitions to the `DONE` state.
- `DONE`:
  - Indicates the encryption process has completed.
  - If the `done` signal is asserted, the FSM resets to the `IDLE` state.
  - Otherwise, it remains in the `DONE` state.
- `default`:
  - If the FSM encounters an undefined state, it defaults to the `IDLE` state for safety.

Key Signals

- `start`: Triggers the encryption process from the `IDLE` state.
- `isLoadDone`: Indicates that data loading is complete.
- `isEncDone`: Signals the completion of the encryption rounds.
- `done`: Resets the FSM back to the `IDLE` state from `DONE`.

Behavior

The FSM executes the cryptographic process in a sequential manner, ensuring that each operation is performed in the correct order. It incorporates conditional transitions (`if` statements) to handle dynamic scenarios, such as verifying when loading or encryption is complete. By cycling through the states, the FSM ensures the proper implementation of the encryption algorithm, adhering to its design flow.

C. Datapath Design

Datapath Logic Description

The datapath governs the transformations applied to the plaintext, tweakey, and intermediate results during the encryp-

tion process. Each state in the design corresponds to a step in the Skinny encryption algorithm, implemented as follows:

- `IDLE`:
  - Resets all control and data signals to their initial values.
- `LOAD`:
  - Loads the input plaintext and tweakey into internal registers.
  - *Implementation Details:*
    - The plaintext is loaded as a 128-bit value, and the tweakey is loaded as a 384-bit value.
    - LUTs are used to map these operations efficiently, minimizing area overhead.
  - Associated Data Processing:
    - The associated data $A_0||A_1||\cdots||A_{\ell_a-1}||A_{\ell_a}$ is processed with $|A_i| = 128$ for $i \in \{0, \ldots, \ell_a - 1\}$ and $|A_{\ell_a}| < 128$.
    - `Auth` is initialized to 0^{128}.
    - `LFSR` is initialized to $0^{63}||1$.
    - For each associated data block A_i, the following operation is performed:

$$\text{Auth} \leftarrow \text{Auth} \oplus \texttt{SKINNY-128-384}_{\text{rev}_{64}(\text{LFSR})\|0^{56}\|00000010\|N\|K}(A_i)$$
$$\text{LFSR} \leftarrow \text{upd}_{64}(\text{LFSR})$$

    - If the last associated data block A_{ℓ_a} is not empty, then:

$$\text{Auth} \leftarrow \text{Auth} \oplus \texttt{SKINNY-128-384}_{\text{rev}_{64}(\text{LFSR})\|0^{56}\|00000011\|N\|K}(\text{pad10*}(A_{\ell_a}))$$

- `SUB_CELLS`:
  - Applies the substitution function to the plaintext using an S-Box.
    - A lookup table (LUT) implements the S-Box substitution function.
    - Each byte of the plaintext is substituted based on the predefined S-Box mapping.
- `ADD_CONST`:
  - Injects the round constant into the first column of the plaintext.
  - *Implementation Details:*
    - Round constants are constructed using:

$$c0 = \{4'b0, RC[3], RC[2], RC[1], RC[0]\}$$
$$c1 = \{6'b0, RC[5], RC[4]\}$$
$$c2 = 8'h02$$

    - These constants are injected into the state matrix using bitwise XOR.
- `ADD_ROUND`:
  - Combines the plaintext with tweakey components and permutes tweakey values.
  - *Implementation Details:*
    - Combines intermediate state (`IS`) with `TK1`, `TK2`, and `TK3` using XOR.
    - LUTs are used for efficient bitwise operations.
    - Implements bit permutation using mappings, e.g.,

$$\text{tweakey\_out}[127{:}120] = \text{tweakey\_in}[55{:}48].$$

    - Linear Feedback Shift Registers (LFSR) are applied for updating `TK2` and `TK3`.
- `SHIFT_ROWS`:
  - Rearranges rows of the plaintext matrix, similar to AES.
  - *Implementation Details:*
    - Shifting operations:
      - Row 1 shifts by 1 byte, Row 2 by 2 bytes, and Row 3 by 3 bytes.
    - LUTs efficiently implement these shifts to minimize delay.
- `MIX_COLUMNS`:
  - Performs linear mixing of columns in the plaintext.
  - Sets `isEncDone` upon completing all rounds.
  - *Implementation Details:*
  - Applies a predefined binary matrix M to each column using XOR.

$$\text{Row i} = \text{col[i][0]} \oplus \text{col[i][2]} \oplus \text{col[i][3]}.$$

- `DONE`:
  - Transfers the final ciphertext to the output register for validation.
  - The ciphertext is stored in `reg_ciphertext`.
  - It is then passed to the testbench for verification.

Functional Overview

The datapath implements the cryptographic transformations specified in the Skinny cipher algorithm. Each state corresponds to a step in the algorithm, ensuring the proper sequence of operations. Control signals (`isLoadDone`, `isEncDone`, `counter`, `round_constant`) and modular functions (`sub_cells()`, `add_constant()`, etc.) enable efficient and systematic processing across rounds. This logic ensures compliance with the Skinny cipher specification while providing flexibility for modular design and implementation.

D. Testbench

The testbench generates stimuli for the DUT, including:

- Input plaintext and tweakey: Delivered byte-wise over multiple clock cycles.
- Output Verification: Compares the output ciphertext against the expected result.

The following code snippet shows the testbench's logic:

```
initial begin
    reset = 1;
    start = 0;
    //  Input Tweakey
    reg_input_tweakey = 384'hdf889548cfc7ea52...;
    //  Input Plaintext
    reg_input_plaintext = 128'ha3994b66ad85a3...;
    #40 reset = 0;
    start = 1;
    //  Feeding the data to the DUT
    for (i = 0; i < 16; i = i + 1) begin
```

```
        input_plaintext = reg_input_plaintext[(127 - i * 8) -: 8];
        input_tweakey = reg_input_tweakey[(383 - i * 24) -: 24];
        #40;
    end

    wait(done);
    //  Compare the expected and received ciphertext
    if (received_ciphertext == expected_ciphertext)
    begin
        $display("Test Passed");
    end else begin
        $display("Test Failed");
    end
end
```

Listing 1. Testbench Code

V. Evaluations and Results

The Skinny-128-384 implementation was synthesized on a Xilinx Spartan-7 FPGA. Key metrics include:

- **Area**: Utilized `1,526 LUTs` and `11 FFs`.
- **Throughput**: Achieved `104.4 Mbps` at `250 MHz` clock frequency.
- **Latency**: Encryption completed in `1226 ns`.

The results confirm that Skinny-128-384 meets lightweight cryptography requirements.

VI. Discussions

The SKINNY-128-384 cipher offers a strong security margin, resisting common cryptanalytic attacks with its tweakable structure. It ensures beyond-birthday-bound security, providing efficient encryption. The cipher supports side-channel attack protection via its Threshold Implementation, minimizing overhead. It balances area and throughput efficiency, making it suitable for various use cases. Furthermore, it handles short messages efficiently, is fully parallelizable, and offers flexibility in parameter handling, allowing users to customize key and tweak sizes for specific needs.

A. Limitations

The current implementation processes one input at a time and outputs its corresponding ciphertext. However, it cannot handle more than one plaintext input simultaneously. This constraint limits the system's ability to efficiently process large volumes of data in parallel, which may impact performance in high-throughput applications. Additionally, the design does not currently support variable block sizes or adapt to different encryption modes, making it less versatile for certain cryptographic use cases.

B. Comparisons

Design Type	Area (GE)	Freq (MHz)	Throughput (Mbps)
Baseline	7627	184	406
Proposed	6194	250	104.4

C. Extensions

In light of the limitations mentioned, future extensions can focus on addressing parallel processing capabilities to allow for the simultaneous handling of multiple inputs. Additionally, further study into the vulnerability of the current design to side-channel attacks is necessary. Although the current implementation includes basic countermeasures, a more in-depth analysis of its resilience against various side-channel attack techniques, such as Differential Power Analysis (DPA) and Electromagnetic Analysis (EMA), would strengthen its security. Furthermore, additional security analysis, particularly related to the cryptographic strength of the system in different operational environments, is warranted to ensure robustness against emerging attack vectors.

VII. Conclusion

This project successfully implemented the SKINNY-128-384 encryption algorithm on FPGA, with a focus on resource efficiency for constrained environments. The design uses a finite state machine (FSM) for control and a modular datapath to process encryption operations, ensuring optimal use of FPGA resources. The implementation was verified through simulation, and the results aligned with the expected performance, demonstrating the practicality of SKINNY for lightweight cryptography in such environments.

Despite its success, the current design is limited to handling a single plaintext input at a time, and cannot process multiple inputs concurrently. Future work will aim to optimize the architecture for higher throughput and introduce support for handling multiple inputs in parallel. Additionally, the study of side-channel attacks will be explored further, along with a deeper security analysis to strengthen the design against potential vulnerabilities.

References

[1] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim, "Skinny-aead and skinny-hash," *IACR Transactions on Symmetric Cryptology*, vol. 2020, Special Issue 1, pp. 88–131, 2020. [Online]. Available: https://tosc.iacr.org/index.php/ToSC/article/view/8619

[2] C. Beierle, A. Bogdanov, G. Cassagne, T. Fuhr, E. Lallemand, F.-X. Standaert, and G. Leurent, "SKINNY: A Lightweight Block Cipher for Low-Resource Devices," Cryptology ePrint Archive, Paper 2016/660, 2016, https://eprint.iacr.org/2016/660.

[3] D. A. Wagner and J. Kelsey, "Tweakable block ciphers," 2002, accessed: 2024-12-07. [Online]. Available: https://people.eecs.berkeley.edu/ daw/papers/tweak-crypto02.pdf

APPENDIX

The following Verilog code initializes the S-box and round constants for use in the encryption.

S-Box Initialization

```
/* SKINNY Sbox */
uint8_t S8[256] = {
 0x65,0x4c,0x6a,0x42,0x4b,0x63,0x43,0x6b,0x55,0x75,0x5a,0x7a,0x53,0x73,0x5b,0x7b,
 0x35,0x8c,0x3a,0x81,0x89,0x33,0x80,0x3b,0x95,0x25,0x98,0x2a,0x90,0x23,0x99,0x2b,
 0xe5,0xcc,0xe8,0xc1,0xc9,0xe0,0xc0,0xe9,0xd5,0xf5,0xd8,0xf8,0xd0,0xf0,0xd9,0xf9,
 0xa5,0x1c,0xa8,0x12,0x1b,0xa0,0x13,0xa9,0x05,0xb5,0x0a,0xb8,0x03,0xb0,0x0b,0xb9,
 0x32,0x88,0x3c,0x85,0x8d,0x34,0x84,0x3d,0x91,0x22,0x9c,0x2c,0x94,0x24,0x9d,0x2d,
 0x62,0x4a,0x6c,0x45,0x4d,0x64,0x44,0x6d,0x52,0x72,0x5c,0x7c,0x54,0x74,0x5d,0x7d,
 0xa1,0x1a,0xac,0x15,0x1d,0xa4,0x14,0xad,0x02,0xb1,0x0c,0xbc,0x04,0xb4,0x0d,0xbd,
 0xe1,0xc8,0xec,0xc5,0xcd,0xe4,0xc4,0xed,0xd1,0xf1,0xdc,0xfc,0xd4,0xf4,0xdd,0xfd,
 0x36,0x8e,0x38,0x82,0x8b,0x30,0x83,0x39,0x96,0x26,0x9a,0x28,0x93,0x20,0x9b,0x29,
 0x66,0x4e,0x68,0x41,0x49,0x60,0x40,0x69,0x56,0x76,0x58,0x78,0x50,0x70,0x59,0x79,
 0xa6,0x1e,0xaa,0x11,0x19,0xa3,0x10,0xab,0x06,0xb6,0x08,0xba,0x00,0xb3,0x09,0xbb,
 0xe6,0xce,0xea,0xc2,0xcb,0xe3,0xc3,0xeb,0xd6,0xf6,0xda,0xfa,0xd3,0xf3,0xdb,0xfb,
 0x31,0x8a,0x3e,0x86,0x8f,0x37,0x87,0x3f,0x92,0x21,0x9e,0x2e,0x97,0x27,0x9f,0x2f,
 0x61,0x48,0x6e,0x46,0x4f,0x67,0x47,0x6f,0x51,0x71,0x5e,0x7e,0x57,0x77,0x5f,0x7f,
 0xa2,0x18,0xae,0x16,0x1f,0xa7,0x17,0xaf,0x01,0xb2,0x0e,0xbe,0x07,0xb7,0x0f,0xbf,
 0xe2,0xca,0xee,0xc6,0xcf,0xe7,0xc7,0xef,0xd2,0xf2,0xde,0xfe,0xd7,0xf7,0xdf,0xff
};
```

Round Constant Initialization

Rounds	Constants
1 - 16	01, 03, 07, 0F, 1F, 3E, 3D, 3B, 37, 2F, 1E, 3C, 39, 33, 27, 0E
17 - 32	1D, 3A, 35, 2B, 16, 2C, 18, 30, 21, 02, 05, 0B, 17, 2E, 1C, 38
33 - 48	31, 23, 06, 0D, 1B, 36, 2D, 1A, 34, 29, 12, 24, 08, 11, 22, 04
49 - 62	09, 13, 26, 0C, 19, 32, 25, 0A, 15, 2A, 14, 28, 10, 20

Input Test Vectors

Tweakey :`df889548cfc7ea52d296339301797449`
`ab588a34a447f1ab2df9e8293fbea9a5`
`ab1afac2611012cd8cefe952618c3e8e`
Plaintext :`a3994b66ad85a3459f44e92b08f550cb`
Ciphertext :`94ecf589e20176b138c6346a10dcfa`