2020•2021
Faculteit Industriële Ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Analysis and implementation of novel non-cryptographic hash functions

PROMOTOR :
Prof. dr. ir. Nele MENTENS
PROMOTOR :
dr. ing. Jo VLIEGEN

Thomas Claesen

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

Gezamenlijke opleiding UHasselt en KU Leuven

2020•2021

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Analysis and implementation of novel non-cryptographic hash functions

PROMOTOR :

Prof. dr. ir. Nele MENTENS

PROMOTOR :

dr. ing. Jo VLIEGEN

Thomas Claesen

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

UHASSELT

KU LEUVEN

Foreword

This thesis forms the culmination of my four-year course in Engineering Technology at UHasselt and KULeuven and months of research and testing within the context of the Master's thesis.

This project started off with the intent of implementing the HeavyKeeper algorithm to detect large flows in network streams. But about halfway through the project, we realized that the original hash function we wanted to implement (Speck), was just a bit too slow. So we decided to change the project. From then on, we focused on finding other hash functions which are fast enough to implement on hardware.

Luckily, my promotors Prof. dr. ir. Nele Mentens, and Arish Sateesan did an amazing job in guiding me towards the new goals for the project. They never failed to keep me motivated by giving me new advice and help during our weekly meetings. And I am very grateful for them to take the time out of their busy schedule to talk with me every week. I would also like to thank my external promotor dr. ing. Jo Vliegen, who helped me with the final thesis and giving me a better view of the results.

Furthermore, I would like to thank dr. Bart Dreesen in providing excellent information on the structure and planning of the master's thesis.

Finally, I would like to thank my family, who have helped me mentally during the darker parts of the lockdown restrictions.

Table of contents

List of tables

List of figures and pictures

Nomenclature

ARX cipher

An ARX cipher stands for Add-Rotate-XOR cipher. Here, only these three operations are used to calculate the ciphertext and the key state.

Avalanche metrics

Values to measure the randomness of a hash function. If it is not random enough, then the hash function can be vulnerable to attacks. It is calculated with the use of three values: Avalanche dependence, avalanche weight and avalanche entropy.

Ciphertext

A piece of text of predetermined length received from a cipher or hash function. It is dependent on the input of the cipher or hash function.

Hash function

A function that uses mathematical functions to transform a piece of text of arbitrary length into a ciphertext of fixed length. When the ciphertext is received, it should be impossible to get the original input. Also, it is very difficult to find two values which have the same ciphertext.

Internal state

Value of the plaintext after it is inserted in the cipher or hash function and before it is output as the ciphertext.

Key state

Same as the internal state, but then for the key instead of the plaintext.

Least significant bits (LSB)

The bit with the lowest value from a word or byte or any value. For example, the byte 00001001 is 9 in decimals. Here, the right most 1 is the least significant bit.

Most significant bits (MSB)

The opposite of the LSB. From the example from the LSB, the left most 0 is the most significant bit.

Plaintext

Input of a cipher or hash function. Usually it has a predetermined length, but for hash functions, it can be any length.

Sbox

A predetermined array of values. It is used inside the ciphers to replace the internal state by the values in the Sbox.

SPN cipher

An SPN cipher stands for Substitute-Permutation network cipher. It uses the substitution mathematical function, followed by permutating the internal states, and sometimes accompanied by an addition of the key.

Symmetric cipher

A symmetric cipher uses the same mathematical functions each round. These rounds are then repeated a set amount of times to get the ciphertext at the end.

Throughput

The amount of data which can be sent through the hash functions in a given amount of time.

Worst negative slack

This value shows the most negative of any single path that failed the timing constraint. It is mostly used as a value to show how badly the timing was missed.

Abstract

Hash functions are vital building blocks for many networking and security applications. In these applications, the speed of hashing is of crucial importance, affecting the overall throughput of the system. The goal of this thesis is to design novel non-cryptographic hash functions based on reduced-round versions of symmetric-key ciphers, and to analyze the avalanche properties and timing characteristics of these algorithms. The considered ciphers are Speck, Pyjamask, GIFT, AES and Skinny. The number of rounds required are determined so that satisfactory avalanche properties are met, both with and without the addition of the key.

After finding the optimal number of rounds, the timing properties are evaluated using hardware design tools. Each hash function is implemented as the reduced version of the original cipher on different FPGA platforms, both with and without the addition of a key. The maximum possible operating frequency is calculated and the resources required at that frequency are also measured. The throughput is calculated based on the maximum operating frequency. The analysis results show that different ciphers have different performance characteristics. Also, the obtained hash functions show better avalanche properties and outperform most of the existing non-cryptographic hash functions.

Abstract in het Nederlands

Hashfuncties zijn essentiële bouwblokken voor veel netwerk- en beveiligingstoepassingen. In deze toepassingen is de snelheid van hashen van cruciaal belang, met een grote invloed op de verwerkingssnelheid van de hashfuncties. Het doel van deze scriptie is om nieuwe niet-cryptografische hashfuncties te ontwerpen op basis van symmetrische-sleutel algoritmen met een gereduceerd aantal ronden en de *avalanche*-eigenschappen en timing beperkingen te analyseren. De volgende algoritmen worden beschouwd: Speck, Pyjamask, GIFT, AES en Skinny. Het aantal benodigde rondes wordt zo bepaald dat er aan de *avalanche* eigenschappen wordt voldaan, zowel met als zonder toevoeging van de sleutel.

Nadat het optimale aantal rondes is gevonden, worden de timing eigenschappen geëvalueerd met behulp van hardware-ontwerptools. Elke hashfunctie is geïmplementeerd als de gereduceerde versie van het originele cijfer op verschillende FPGA-platformen, zowel met als zonder sleutel. De maximaal mogelijke werkingsfrequentie wordt berekend en ook de bij die frequentie benodigde middelen worden gemeten. De doorvoer wordt berekend op basis van de maximale werkingsfrequentie. De analyseresultaten tonen aan dat verschillende algoritmen verschillende prestatiekenmerken hebben. Ook vertonen de verkregen hashfuncties betere *avalanche* eigenschappen en presteren ze beter dan de meeste bestaande niet-cryptografische hashfuncties.

1 Introduction

Hash functions are operations where a text of arbitrary length is converted into an encrypted message with fixed length. They are one-way functions, which means that it is not feasible to recover the original message in practice. They are used in many networking and security applications where hash functions help to reduce the memory overhead and increase the speed of the computations. Network security applications such as Bloom filters [1] for fast lookups and sketches [2] for memory efficient measurements use hash functions as one of their components. With the ever increasing data-rate of the internet, the throughput of these applications needs to be matched. However, the throughput of these applications is reliant on the efficiency of the hash functions, which makes the speed of the hash functions an important factor.

Usually, hash functions offer high levels of security. However, these are not necessary for lookup or counting architectures. Only data-rate and avalanche properties are more important. In this thesis, non-cryptographic hash functions inspired by five symmetric-key ciphers are proposed: Speck, Pyjamask, GIFT, AES and Skinny. How these ciphers work, is described in section 2. Section 3 will be an analysis of the avalanche properties for each cipher as a function of the number of rounds. These avalanche properties are measurements to indicate to which extent a change of the input has an influence on the change of the output. The optimal amount of rounds for each cipher is measured here. Lastly, section 4 will measure the throughput for these ciphers. The hash functions will be evaluated on both Zynq and Virtex Ultrascale+ FPGAs. All of these calculations are done using an input of 96 bits. Some ciphers, however, only operate with an input of 128 bits. For these, the results are generated for only 96 bits, while 32 other bits are fixed to 0.

2 Analysis of the hash functions

2.1 Speck

2.1.1 Initialization

The Speck block cipher family [3] holds a staggering 10 different variations of the Speck algorithm. Table 1 shows the difference between these variations. This thesis focuses on the variation with block size and key size of 96 bits. Speck is an ARX cipher, which means it only uses these three operations to calculate the ciphertext.

Table 1: Different variations of the Speck cipher [3]

block size $2n$	key size mn	word size n	key words m	rot α	rot β	rounds T
32	**64**	16	4	7	2	22
48	**72**	24	3	8	3	22
	96		4			23
64	**96**	32	3	8	3	26
	128		4			27
96	**96**	48	2	8	3	28
	144		3			29
128	**128**	64	2	8	3	32
	192		3			33
	256		4			34

During the rounds, the key and the plaintext are split into two parts of 48 bits. The most significant bits of the plaintext and the key are stored in plaintext 1 and key 1 respectively, while the least significant bits are stored in plaintext 2 and key 2.

2.1.2 Round Function

In Speck, each round performs the same three operations: Rotation, Addition and an XOR. First, plaintext 1 is rotated 8 bits to the right, this is followed by an addition with plaintext 2. plaintext 1 is the XORed with key 2. For plaintext 2, the operations start after the addition to plaintext 1, it is first rotated 3 bits to the left and then XORed with plaintext 1. Figure 1 shows how this is done for a single round. After each round, the output is connected to the input of the next round.

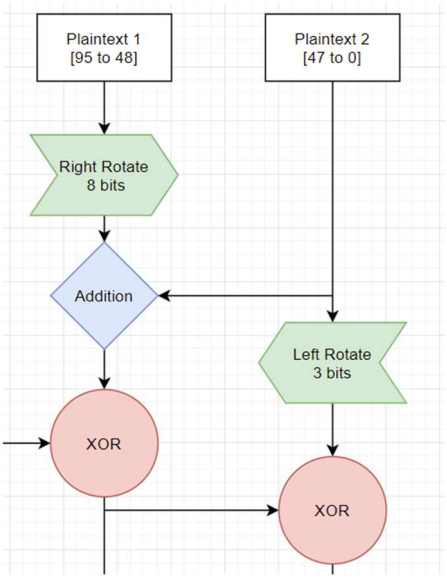


Figure 1: A single encryption round for Speck

2.1.3 Key schedule

The key changes every round for Speck, this round function works similarly to the encryption round for the plaintext. First, key 1 is rotated 8 bits to the right. Then, key 2 is added to key 1. This is followed by and XOR with the number of the round, this starts counting from 0 from the first round and increments by 1 every round. Key 2 starts with a rotation by 3 bits to the left and is then XORed with the intermediate value on the left. Figure 2 shows how these operations are executed onto the key.

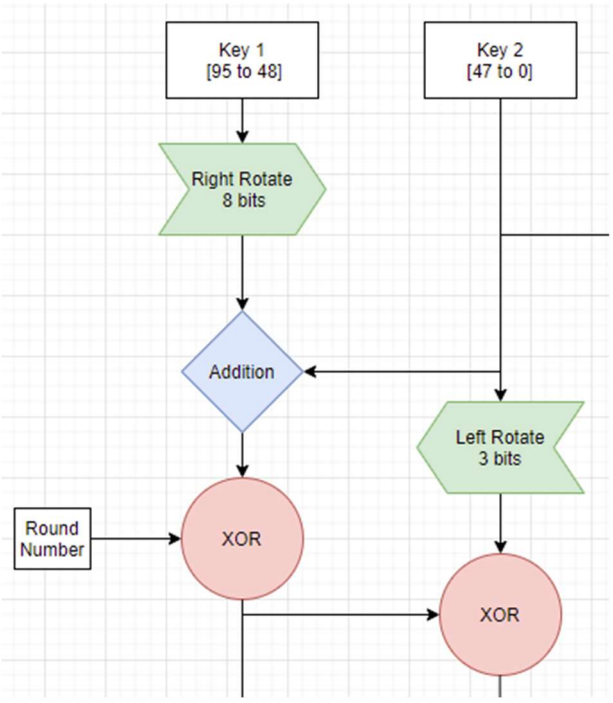


Figure 2: Key round operations in Speck

A full diagram of the first two rounds of the Speck algorithm can be found in appendix A.

2.2 Pyjamask

2.2.1 Initialization

Pyjamask is a block cipher family [4] which contains two algorithms: Pyjamask-96, which has a block size of 96 bits, and Pyjamask-128, which has a block size of 128 bits. Both algorithms use a key with a size of 128 bits, perform 14 rounds and rely on a Substitution-Permutation Network (SPN) structure to transform the plaintext into the ciphertext. The plaintext is structured in a bit by bit left-right top-down structure. Each row consists of 32 bits. Depending on the block size of the chosen algorithm, 3 or 4 rows are used. The cell which holds the lowest index represents the most significant bit of the plaintext and the cell with the highest index represents the least significant bit. The key is represented in the same way as the plaintext. Only this time, there are 4 rows used of which each contain 32 bits. Once the plaintext enters the algorithm, it will be referred to as the internal state. Figure 3 represents the structure of the plaintext for pyjamask-96.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95

Figure 3: Representation of the plaintext in Pyjamask-96

Each round is composed of three operations, which happen in chronological order: AddRoundKey, SubBytes and MixRows.

2.2.2 Round Functions

- AddRoundKey

  First, the key is bitwise added to the internal state. For Pyjamask-128, this is done with a simple XOR. For Pyjamask-96 however, only the 96 most significant bits of the key are XORed with the 96-bit internal state.

- SubBytes

  Then, all 32 columns of the internal state are extracted and compared to a constant Sbox. For Pyjamask-96, this Sbox is called S3 and for Pyjamask-128 it is called S4 . S3 and S4 are shown in Figure 4 and Figure 5 respectively. For example, in a given Pyjamask-96 round, column 1 contains the bits “100”, which is binary for the number 4. The fourth number from S3 is 2 (start counting from zero). 2 in binary gives “010”, which is then put in column 1.

$$S_3 = [1, 3, 6, 5, 2, 4, 7, 0]$$

Figure 4: Sbox for Pyjamask-96

$$S_4 = [0x2, 0xd, 0x3, 0x9, 0x7, 0xb, 0xa, 0x6, 0xe, 0x0, 0xf, 0x4, 0x8, 0x5, 0x1, 0xc]$$

Figure 5: Sbox for Pyjamask-128

- MixRows

  Finally, each row R_i of the internal state is seen as a column matrix. This matrix is then multiplied by a constant matrix M_i with $i \in \{0, 1, 2\}$ for Pyjamask-96 and $i \in \{0, 1, 2, 3\}$ for Pyjamask-128. Each row in the internal state is then replaced by $M_i \cdot R_i$. Each matrix M is a 32x32 circulant binary matrix, they are shown in Figure 6.

  After the last round, a final AddRoundKey operation is applied to the internal state and the ciphertext is created. Appendix B shows a visual representation of a round in the Pyjamask-96 algorithm.

$$\mathbf{M}_0 = \text{cir}([1,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,1,1,1,0,0,0,1,0])$$
$$\mathbf{M}_1 = \text{cir}([0,1,0,0,0,0,1,0,0,0,0,0,0,1,1,1,0,1,0,0,0,0,0,1,0,1,1,0,0,0,1,1])$$
$$\mathbf{M}_2 = \text{cir}([0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,1,1,0,0,1,1,0,1,0,0,1,0,0,1,0,1,1])$$
$$\mathbf{M}_3 = \text{cir}([0,1,1,0,0,1,0,0,0,0,0,0,1,0,0,1,0,1,0,1,0,0,1,0,1,0,0,0,1,0,0,1])$$

Figure 6: Matrices used in the MixRows operation

2.2.3 Key Schedule

Throughout the algorithm, different keys are used in each AddRoundKey. These keys are called subkeys, and all originate from the original secret key. To receive these subkeys, three operations are executed each round: MixColumns, MixRows and Constant Addition. Because Pyjamask-96 and Pyjamask-128 both use 128-bit keys, the same operations are used to create the subkeys.

- MixColumns

  First, all 32 columns C_i of each 4 bits are replaced by $C_i \cdot M$ where M is a constant matrix, which is shown in Figure 7.

$$\mathbf{M} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Figure 7: Matrix used in the MixColumns operation

- MixRows

  Then, the first row R_0 of the key state is replaced by $M_K \cdot R_0$. This operation is similar to the MixRows operation in the internal state. M_K is a 32x32 circulant matrix, which is shown in Figure 8. The second row R_1 of the key state, is rotated to the left by 8 bits. The third and fourth row R_2 and R_3 are also similarly rotated to the left by 15 and 18 bits respectively.

$$\mathbf{M}_K = \text{cir}([1,0,1,0,1,0,0,1,1,1,0,0,1,1,1,0,1,1,0,0,0,0,0,0,1,0,0,0,1,1,1,0])$$

Figure 8: Matrix used on the first row in the MixRows operation for the key state

- Constant Addition

  Finally, a 32-bit constant value is broken down into four bytes, which are then bitwise added to the key state. The four least significant bits of this constant are equal to the round number, which is between 0 and 13. Figure 9 gives a visual representation of this operation.

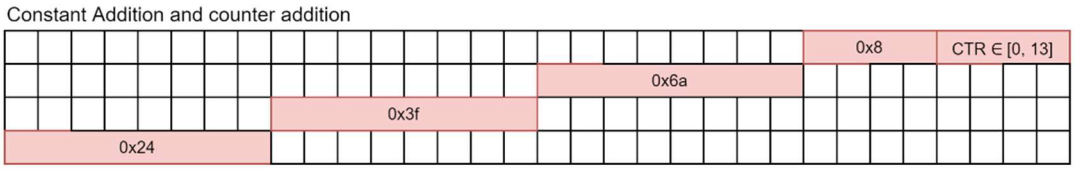


Figure 9: Visual representation of the Constant Addition operation

Appendix C gives a full visual image of a single round of the key schedule.

2.3 GIFT

2.3.1 Initialization

The GIFT block cipher family [5] contains two algorithms: GIFT-64 and GIFT-128. Both use a key size of 128 bits, while the former uses a block size of 64 bits and the latter of 128 bits. In this thesis, GIFT-128 is used. Like Pyjamask, GIFT is an SPN cipher containing 40 rounds of three operations: SubCells, PermBits and AddRoundKey. There are multiple ways the plaintext can be initialized, but this thesis uses the following method: The plaintext is initialized in a 4x32 matrix of bits and the most significant bits are stored in the cell with the lowest index, so cell 0 stores the most significant bit of the plaintext. Unlike Pyjamask, the bits are stored top-down left-right. The 128-bit key is stored like the key in Pyjamask, left-right top-down. A visual interpretation of the internal state is shown in Figure 10.

Plaintext input

0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61	65	69	73	77	81	85	89	93	97	101	105	109	113	117	121	125
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62	66	70	74	78	82	86	90	94	98	102	106	110	114	118	122	126
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63	67	71	75	79	83	87	91	95	99	103	107	111	115	119	123	127

Figure 10: Representation of the plaintext in GIFT

2.3.2 Round Functions

- SubCells

  Similar to SubBytes in Pyjamask, each 4-bit column is compared to an Sbox and replaced by the appropriate value. The Sbox is show in Figure 11. X gives the value which was in the column before the operation and GS(x) is the value which replaces x.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$GS(x)$	1	a	4	c	6	f	3	9	2	d	b	7	5	0	8	e

Figure 11: Sbox used in the SubCells operation

- PermBits

  This operation shuffles all the bits in the internal state in a given way. It is shown in Figure 12.

Permutate Bits

0	96	64	32	4	100	68	36	8	104	72	40	12	108	76	44	16	112	80	48	20	116	84	52	24	120	88	56	28	124	92	60
33	1	97	65	37	5	101	69	41	9	105	73	45	13	109	77	49	17	113	81	53	21	117	85	57	25	121	89	61	29	125	93
66	34	2	98	70	38	6	102	74	42	10	106	78	46	14	110	82	50	18	114	86	54	22	118	90	58	26	122	94	62	30	126
99	67	35	3	103	71	39	7	107	75	43	11	111	79	47	15	115	83	51	19	119	87	55	23	123	91	59	27	127	95	63	31

Figure 12: Permutation of the internal state

- AddRoundKey

  Finally, the key is added to the internal state. The key is split into 4 blocks K_i of 32 bits each. The same is done to the internal state with blocks P_i. For both the key and the internal state, $i \in \{0, 1, 2, 3\}$ where P_0 and K_0 hold the most significant bits of both the key and the internal state. Of these blocks, P_1 and P_2 are XORed with K_1 and K_2 respectively. The most significant bit of P_3 is also XORed with '1' and the least significant bits of P_3 is XORed with a constant. This value is extracted from the round constant lookup table based on which round is given. A visual representation of this operation is shown in Figure 13 and the round constant lookup table is shown in Figure 14.

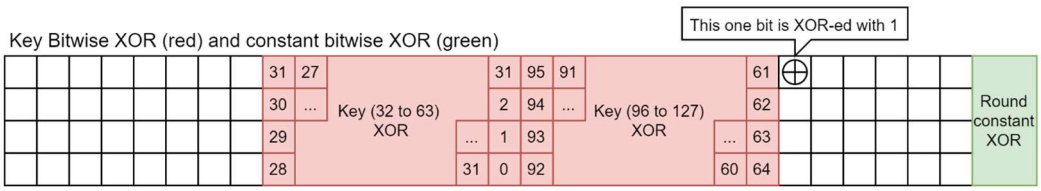


Figure 13: Visual representation of the AddRoundKey operation

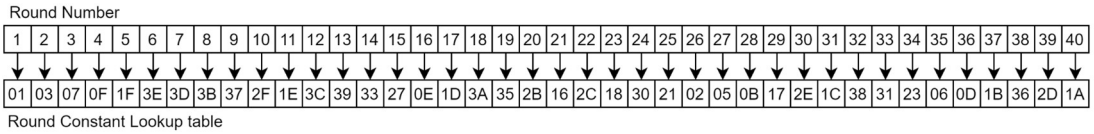


Figure 14: Round Constant Lookup Table

Appendix D gives a visual representation of a full round in the GIFT-128 algorithm.

2.3.3 Key Schedule

The key schedule is a very simple operation for GIFT compared to the key schedule for Pyjamask. The key is again split in 4 blocks K_i of 32 bits each, like in AddRoundKey. K_0, K_1 and K_2 are then shifted towards the least significant bits of the key state, while K_3 now holds the most significant bit. K_3 is then split again in two 16-bit blocks, $K_{3,1}$ and $K_{3,2}$. These two blocks are then right-rotated 2 and 12 bits respectively. Figure 15 shows how this operation works.

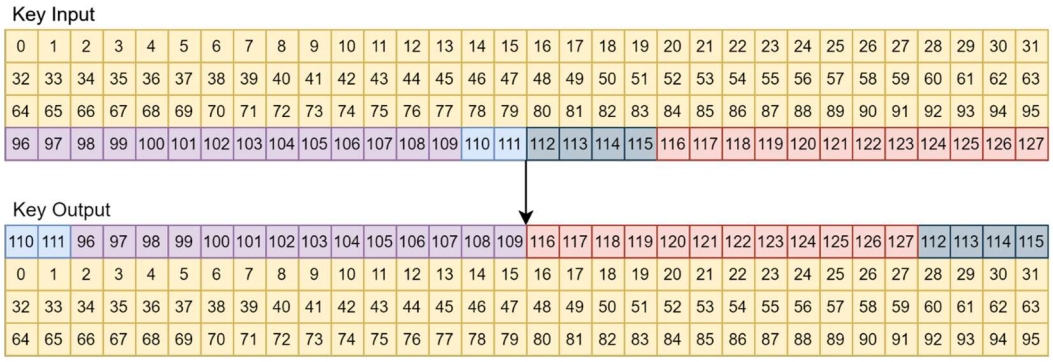


Figure 15: Visual representation of a round in the GIFT key schedule

2.4 AES

2.4.1 Initialization

AES is probably the most famous encryption cipher [6]. It has a block size of 128 bits and the key size can be either 128, 192 or 256 bits. Depending on the key size, the number of rounds are 10, 12 and 14 respectively. This thesis focuses on the algorithm with key size of 128 bits Each round consists of 4 operations: SubBytes, ShiftRows, MixColumns and AddRoundKey. However, the first and last round differ slightly from this order. The first round starts with an initial AddRoundKey and the last round does not calculate the MixColumns operation.

Just like in GIFT, the plaintext input is structured in a top-down left-right structure. The difference with GIFT, is that AES is structured byte-by-byte instead of bit-by-bit. This results in a 4x4 matrix of bytes, as shown in Figure 16. The key is structured similarly.

Plaintext Input as bytes

b_0	b_4	b_8	b_{12}
b_1	b_5	b_9	b_{13}
b_2	b_6	b_{10}	b_{14}
b_3	b_7	b_{11}	b_{15}

Figure 16: Structure of the internal state in AES

2.4.2 Round Functions

- SubBytes

  Similarly to Pyjamask and GIFT, each byte is compared and replaced by a value from the Sbox. The Sbox can be found in appendix E and is a 16x16 matrix where the column selection is determined by the last 4 bits of the byte and the row selection by the first 4 bits.

- ShiftRows

  Afterwards, each row is shifted in a given way. The first row stays the same, the second row is shifted one byte to the left, the third row is shifted two bytes to the left and the fourth row three bytes to the left. Figure 17 shows how this is done.

Shift Rows

b_0	b_4	b_8	b_{12}
b_5	b_9	b_{13}	b_1
b_{10}	b_{14}	b_2	b_6
b_{15}	b_3	b_7	b_{11}

Figure 17: ShiftRows operation in AES

- MixColumns

  Then, the MixColumns operation works similarly to the one in the key schedule in Pyjamask. Each column is multiplied by a given Matrix M, which is given in Figure 18. The results then replace each column of the internal state.

Multiplication Matrix for Mix Columns

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

Figure 18: Matrix M for the MixColumns operation in AES

- AddRoundKey

  Finally, the AddRoundKey calculates a simple XOR of the key into the internal state. This is similar to the AddRoundKey function in Pyjamask. Each round uses a different subkey. These are calculated in the key schedule.

  The full visual representation of a single round can be found in appendix F.

2.4.3 Key Schedule

The key schedule of AES, similar to GIFT, is constructed of only one operation. It is however, slightly more complex. As mentioned in the beginning of this chapter, the key state is structured like the internal state: byte-by-byte, top-down left-right. Here, each byte is numbered K_i where $i \in \{0, 1, 2, \ldots, 14, 15\}$. K_0 holds the most significant byte of the key state. During the key schedule, if i is greater than or equal to 4, every byte K_i is XORed with K_{i-4}. Furthermore, K_1, K_2 and K_3 are XORed with a value from the same Sbox as in SubBytes by extracting the value from K_{14}, K_{15} and K_{12} respectively and putting those in said Sbox. Finally, K_0 is XORed with the value from the Sbox by extracting K_{13} and by a round constant. Figure 19 gives a visual representation of this operation and Figure 20 gives the round constant lookup table.

Key schedule

k_0 xor $S(k_{13})$ xor Rconst	k_4 xor k_0	k_8 xor k_4	k_{12} xor k_8
k_1 xor $S(k_{14})$	k_5 xor k_1	k_9 xor k_5	k_{13} xor k_9
k_2 xor $S(k_{15})$	k_6 xor k_2	k_{10} xor k_6	k_{14} xor k_{10}
k_3 xor $S(k_{12})$	k_7 xor k_3	k_{11} xor k_7	k_{15} xor k_{11}

Figure 19: Key Schedule for AES

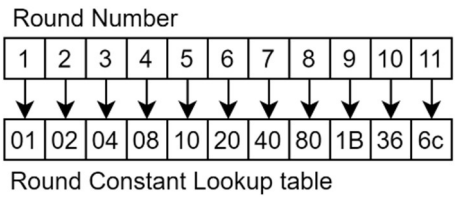


Figure 20: Round Constant Lookup Table for the Key Schedule in AES

Appendix G gives a full representation of the Key Schedule.

2.5 Skinny

2.5.1 Initialization

The final cipher that will be discussed in this thesis, is the Skinny block cipher family [7], which holds 6 algorithms:

- Skinny-64-64
- Skinny-64-128
- Skinny-64-196
- Skinny-128-128
- Skinny-128-256
- Skinny-128-384

The first numerical value is the block size used in the algorithm, the second value is the key size. This thesis focuses on Skinny-128-128.

The plaintext and key are initialized similarly to AES, but now in a left-right top-down method, as shown in Figure 21.

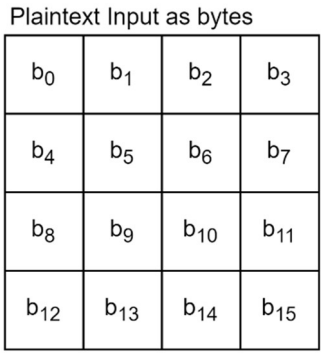


Figure 21: Internal state of Skinny

Skinny also uses a different amount of rounds depending on the algorithm used. For the algorithm used in this thesis, which is Skinny-128-128, 40 rounds are used. Each round consists of no less than 5 different operations: SubBytes, AddConstants, AddRoundKey, ShiftRows and MixColumns.

2.5.2 Round Functions

- SubBytes

  First, the SubBytes operation is calculated. This follows the exact same way as SubBytes in AES, only this time, the Sbox is different. This Sbox can be found in Appendix H.

- AddConstants

  Next, round constants are XORed to the first column of the internal state. The value in b_8, as shown in Figure 21, is XORed with a constant hexadecimal value of 0x02. Meanwhile, b_{12} stays the same, but b_0 and b_4 are XORed with a round constant, this value is a 6-bit number which changes every round. The 4 least significant bits of b_0 are XORed with the 4 least significant bits of this value and the 2 least significant bits of b_4 are XORed with the 2 most significant bits of the round constant. The lookup table for this round constant, is shown in appendix I.

- AddRoundKey

  Then, the key state is added to the internal state. Unlike the AddRoundKey function in AES, only the 8 most significant bytes of the key are XORed to the 8 most significant bytes of the internal state.

- ShiftRows

  Furthermore, the ShiftRows function is implemented. Fundamentally, this works the same way as the ShiftRows function in AES, only this time the rows are shifted to the right. Also, due to the initial structure of the internal state being different than the one in AES, this ShiftRows gives other results. Like AES, the first row is not shifted, the second row is shifted right once, the third row shifted twice and the last row shifter thrice. The result is shown in Figure 22.

Shift Rows

b_0	b_1	b_2	b_3
b_7	b_4	b_5	b_6
b_{10}	b_{11}	b_8	b_9
b_{13}	b_{14}	b_{15}	b_{12}

Figure 22: ShiftRows function in Skinny

- MixColumns

  Finally, the MixColumns function wraps up the round. Similar to MixColumns in Pyjamask and AES, each column of the internal state is multiplied by a Matrix M, which is given in Figure 23.

Multiplication Matrix for Mix Columns

1	0	1	1
1	0	0	0
0	1	1	0
1	0	1	0

Figure 23: Multiplication Matrix for the MixColumns function in Skinny

Appendix J gives a full visual diagram of a single round in Skinny.

2.5.3 Key Schedule

Just like other ciphers, the key updates every round into different subkeys. In Skinny, the key updates similarly to the key schedule in GIFT. By shifting different key bytes to other places in the key state. How the bytes shift, is shown in Figure 24.

Key Schedule

k_9	k_{15}	k_8	k_{13}
k_{10}	k_{14}	k_{12}	k_{11}
k_0	k_1	k_2	k_3
k_4	k_5	k_6	k_7

Figure 24: Key Schedule for Skinny

A full visual representation of the key schedule can be found in appendix K.

3 Performing the avalanche analysis

3.1 Avalanche metrics

The avalanche properties are metrics used to measure the influence of an input change to the change of the output. If the hash function does not show good avalanche properties, then a change at the input does not have an effect on a sufficient amount of output bits. This leaves the hash function vulnerable to attacks. There are three avalanche metrics which are used to evaluate the hash functions mentioned in this thesis: Avalanche dependence, avalanche weight and avalanche entropy [8].

These values are all calculated using the probability vector, which is a measure of the average amount of times a bit changes in the output if you only change one bit in the input. It stores the average bit changes for each bit in the output. These calculations have to be iterated a large amount of times to try to prevent outliers. Figure 25 shows the pseudocode to calculate the probability vector P. N indicates the input size, which is 96 for Speck and Pyjamask, or 128 for GIFT, AES and Skinny. A1 and A2 are inputs for the hash functions and H1 and H2 are outputs. Finally, T gives which bit needs to be toggled from A1 to get A2. The probability vector is calculated for every value of T (0 to 95 or 127) and the avalanche metrics are measured for every probability vector of T. Afterwards, the worst metrics are stored.

These calculations are done in search for the ideal number of rounds each cipher needs, to get good enough avalanche metrics. When all calculations are done, the avalanche metrics are plotted in function of the number of rounds.

```
Initialize:
        COUNTS = [0,0,...,0]
        P = [0,0,...,0]

for (i<M) do
        A1 = random()
        H1 = F(A1)
        A2 = Toggle any single bit of A2
        H2 = F(A2)
        X = H1 xor H2
        for (i<n) do
                COUNTS[i] = COUNTS[i] + X[i]

for (i<n) do
        P[i] = COUNTS[i] / M
```

Figure 25: Pseudocode to calculate the probability vector P [9]

3.1.1 Avalanche Dependence

The avalanche dependence D_{av} is defined by the number of bits which toggle in the output when there is a single-bit change in the input. Equation 1 shows how this value is calculated [9]. Here, n is the number of bits at the output, p[i] is the probability vector and g(p) is a function where g(p) = 1 if p = 0 and g(p) = 0 otherwise. The avalanche dependence is satisfied when $D_{av} = n$.

$$D_{av} = n - \sum_i g(p[i]) \quad [1]$$

3.1.2 Avalanche weight

The avalanche weight w_{av} is a measurement to define the weight of the output difference. The formula is shown in equation 2. It is in essence a sum of the probability vectors. Ideally, half of the output should change so that the randomization is at its best, which means that the ideal value for the avalanche weight is half of the output size n.

$$w_{av} = \sum_i p[i] \quad [2]$$

3.1.3 Entropy

Finally, the avalanche entropy H_{av} is a value to show the uncertainty about whether the output bits toggle or not for a single-bit input change in the input.

$$H_{av} = \sum_i (-p[i] \cdot \log_2(p[i]) - (1 - p[i]) \cdot \log_2(1 - p[i])) \quad [3]$$

3.2 Avalanche results

These three avalanche metrics are then plotted against the number of rounds that have been executed. Figure 26 and Figure 27 show these plots for ciphers with key and without key respectively. When a key is used, it is initialized at a fixed value of 0. When the key is not used, every operation involving the key is simply ignored. For each cipher, NC (Non-Cryptographic) is added to the name to indicate the round-reduced version. The X-axis gives the number of rounds used when the avalanche metrics are measured and the Y-axis gives the values for the avalanche metrics, which are in bits.

As shown in the figures, there is no significant difference between the avalanche properties when a key is used or not. Even more so, the key seems to have no effect on the avalanche properties of the hash functions. This means that when it is desired that the output of the hash function needs to be changed, the avalanche properties will not be affected.

As mentioned in section 2, Speck normally needs 28 rounds to be considered secure. Speck-NC, however, only requires 7 rounds to be used as a non-cryptographic hash function. The same happens for all other ciphers. Pyjamask requires 14 rounds, but it meets the avalanche requirements at round 3. However, 2 rounds are also a valuable solution if speed is of the essence. GIFT-NC only needs 7 rounds instead of GIFT's 40 rounds, but 6 rounds are also acceptable. AES goes from 10 rounds to 3 rounds for AES-NC and finally, Skinny-NC needs only 6 rounds while Skinny needs 40. Table 2 gives a summary of the results of the avalanche analysis.

Finally, as mentioned earlier, Speck and Pyjamask have an input and output size of 96 bits, while the other ciphers have a 128-bit input and output size. However, when the 128-bit ciphers are tested for their avalanche properties, a random input of 96 bits is padded with 32 zero bits. The placement of the padding zeros has no effect on the avalanche metrics.

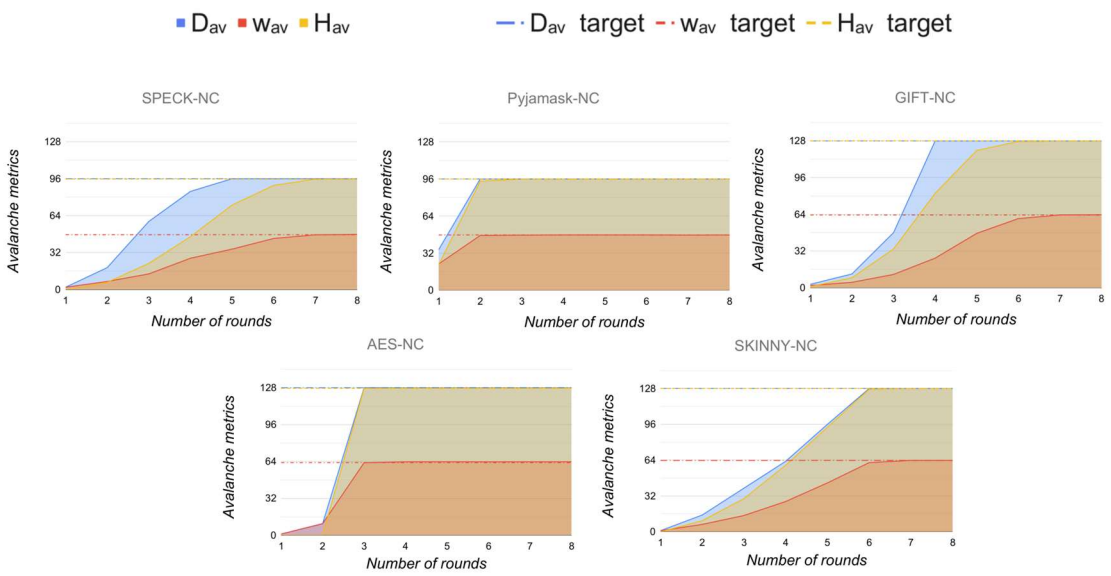


Figure 26: Results of the Avalanche analysis with key

Table 2: Results of the avalanche analysis with key

Hash function	D_{av}	w_{av}	H_{av}	r	r_{av}
Speck-NC	96	47.3848	95.6196	28	7
Pyjamask-NC	96	47.5808	95.9627	14	3
GIFT-NC	128	63.4416	127.9309	40	7
AES-NC	128	62.854	127.4397	10	3
Skinny-NC	128	61.798	127.3621	40	6

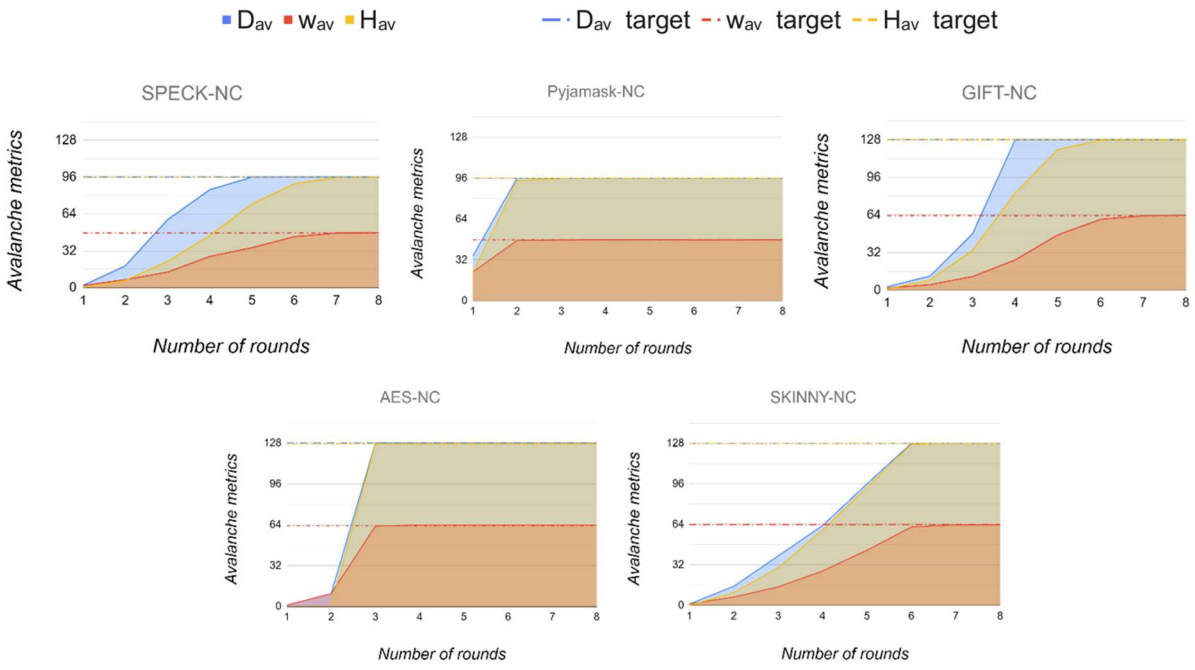


Figure 27: Results of the Avalanche analysis without key

Table 3: Results of the avalanche analysis without key

Hash function	D_{av}	w_{av}	H_{av}	r	r_{av}
Speck-NC	96	47.4476	95.6829	28	7
Pyjamask-NC	96	47.7296	95.9628	14	3
GIFT-NC	128	63.3888	127.9482	40	7
AES-NC	128	62.858	127.4356	10	3
Skinny-NC	128	61.830	127.4030	40	6

4 Hardware Evaluation

4.1 Setup

The reduced round version of the ciphers are implemented on FPGA to evaluate the performance on hardware. The platforms chosen for the evaluation are Zynq 7020, Virtex Ultrascale and Virtex Ultrascale+ board. To generate the results, the Vivado 2019.1 design tool is used. Each cipher is tested for a specific clock period and the Worst Negative Slack (WNS) is taken to check if the specified timing constraint is met. If this value is positive, then the timing constraint is met and the operating speed can be faster. If it is negative, then the clock period needs to be lowered. Figure 28 gives a general setup for the hardware implementation. As shown in the figure, only input and output registers are added and the execution of all the rounds are completed in a single clock cycle. The number of rounds for each cipher is equal to the number of rounds which are needed to maintain the avalanche properties. These are calculated in section 3 and shown in Table 2 and 3 under the value of r_{av}. The key is not a necessary parameter for traditional hash functions, as a given plaintext should always have the same ciphertext. But for some applications, like in Bloom filters, it can be useful to receive other ciphertext values for the same plaintext. This can be done by using different key values. A full visual representation of the hardware implementation of all ciphers, can be seen in appendix L.

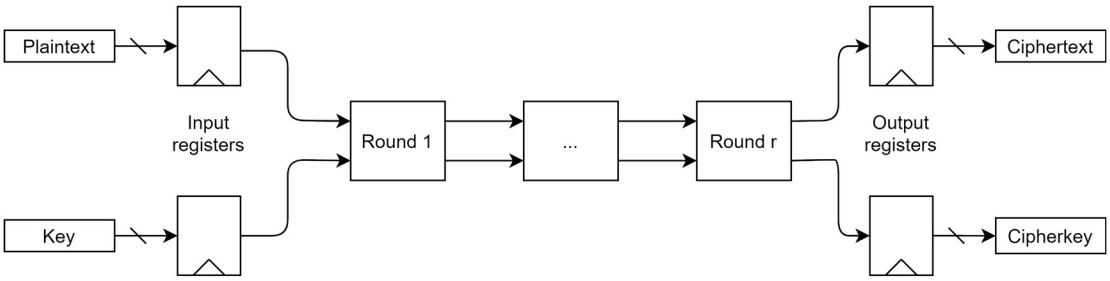


Figure 28: General setup for the hardware implementation

4.2 Hardware results

4.2.1 Timing

Figure 29 shows the results for the timing tests. The X-axis gives the clock period and the Y-axis gives the WNS. Each cipher is tested on all three boards, once with key and once without key. Using the graphs, it is possible to calculate the maximum operating frequency using equation 4,where the min clock period is taken as the lowest possible clock period with a positive WNS.

$$f_{max} = \frac{1{,}000{,}000{,}000\frac{ns}{s}}{min\ clock\ period\ (ns)} \quad [4]$$

It is clearly visible from the graphs that the key has an impact on the timing constraints for Speck, Pyjamask and AES. Speck sees an increase of operating frequency of 11-18%. Furthermore, the operating frequency of AES increases by 15-25%. While Pyjamask's operating frequency only increases on the Zynq board by 18%. GIFT and Skinny see no significant improvements when the key is removed. From all the hash functions mentioned in this thesis, GIFT-NC looks the most promising. It is the fastest hash function for both using the key and not using the key. Table 4 shows a better representation of the maximum operating frequencies of the hash functions. As expected, the frequency increases for bigger boards.

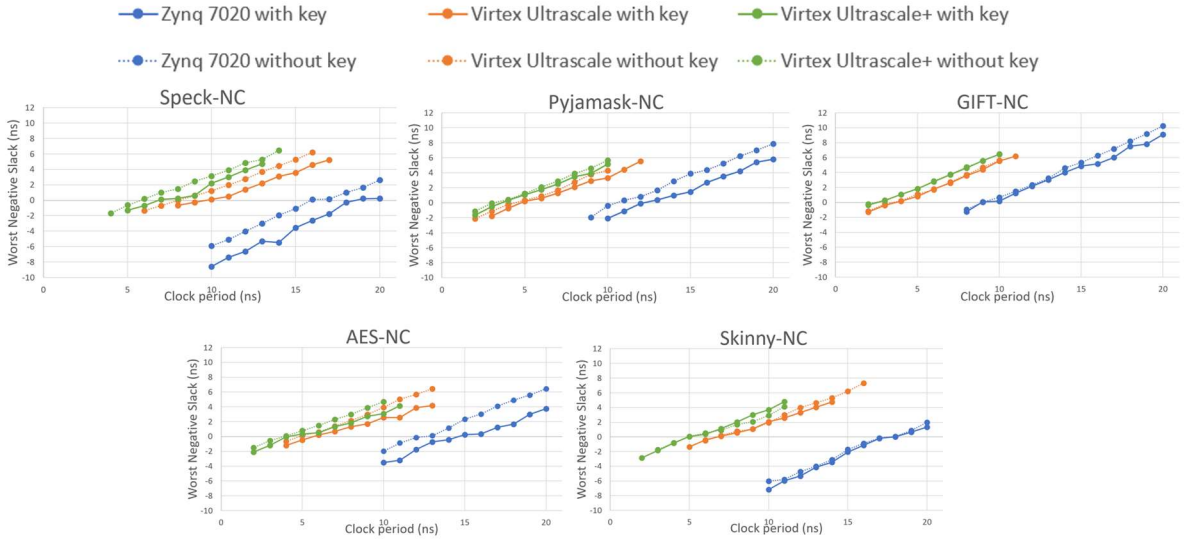


Figure 29: Timing results for the hardware evaluation

Table 4: Max frequency measured in the hardware implementation without key

	Max Frequency (without key)		
Board / Hash function	Zynq 7020	Virtex Ultrascale	Virtex Ultrascale+
Speck-NC	62.50 MHz	111.11 MHz	166.66 MHz
Pyjamask-NC	90.91 MHz	200.000 MHz	250.00 MHz
GIFT-NC	111.11 MHz	250.00 MHz	333.33 MHz
AES-NC	76.92 MHz	200.00 MHz	250.00 MHz
Skinny-NC	55.56 MHz	142.86 MHz	200.00 MHz

Table 5: Max frequency measured in the hardware implementation with key

	Max Frequency (with key)		
Board / Hash function	Zynq 7020	Virtex Ultrascale	Virtex Ultrascale+
Speck-NC	52.63 MHz	100.00 MHz	142.86 MHz
Pyjamask-NC	76.92 MHz	200.00 MHz	250.00 MHz
GIFT-NC	111.11 MHz	250.00 MHz	333.33 MHz
AES-NC	66.67 MHz	166.67 MHz	200.00 MHz
Skinny-NC	55.56 MHz	142.86 MHz	200.00 MHz

4.2.2 Resource utilization

Another value which can be received from the hardware implementation, are the resources used for each cipher. Table 6 shows these results. It is clearly visible that GIFT-NC is uses the least amount of resources of all ciphers with key, but Speck-NC is slightly better for all ciphers without key. However, when the key is added, it needs two to three times the amount of resources. Meanwhile, GIFT-NC and Skinny show negligible increase in resources when the key is added compared to the other hash functions.

Table 6: Resources used in the hardware implementation

	Without key		With key	
Hash function	LUTs	Flip Flops	LUTs	Flip Flops
Speck-NC	432	192	1273	384
Pyjamask-NC	811	448	1615	448
GIFT-NC	546	512	665	512
AES-NC	2225	512	3402	512
Skinny-NC	2176	512	2348	512

4.2.3 Throughput

The throughput measures how many bits per seconds can be passed through the hash function per second. It can be calculated using equation 5, where latency in cycles is equal to 1 because all rounds are calculated in one cycle. The block size is equal to 96 bits and the max frequency is taken from the highest value from

Table 4. The throughput is measured in bits per second and the results are shown in Table 7. Again, GIFT-NC shows the best throughput among the ciphers evaluated in this thesis. It has the highest throughput out of all ciphers with or without key. Meanwhile, Speck-NC and AES-NC show an increase in throughput of 16,67% and 25% respectively.

$$Throughput = \frac{Block\ size}{latency\ in\ cycles} \cdot f_{max} \quad [5]$$

Table 7: Throughput results for all ciphers with and without key

	Without key		With key	
Hash function	Maximum frequency	Throughput (Mbps)	Maximum frequency	Throughput (Mbps)
Speck-NC	166.66 MHz	16,000	142.86 MHz	13,714
Pyjamask-NC	250.00 MHz	24,000	250.00 MHz	24,000
GIFT-NC	333.33 MHz	32,000	333.33 MHz	32,000
AES-NC	250.00 MHz	24,000	200.00 MHz	19,200
Skinny-NC	200.00 MHz	19,200	200.00 MHz	19,200

4.2.4 Comparison with related work

The five novel non-cryptographic hash functions measured in this thesis are now related to some existing hash functions: Murmur3 [10], FNV-1a [11], SipHash [12], XORHash [12], NSGAHash7 [12] and Xoodoo-NC [13]. Here, Murmur3 and FNV-1a were implemented on

FPGA to get the timings for this thesis. Table 8 gives a full overview of the comparison. Although XORHash has the highest operating frequency, it has 5.57x less throughput compared to Speck-NC. Which has the lowest throughput among the hash functions analyzed in this thesis. This is due to XORHash having a 32-bit block size and a latency of 7 clock cycles. Furthermore, Xoodoo-NC has a slightly higher throughput over GIFT-NC, which has the highest throughput out of all the hash functions analyzed in this thesis. However, in theory the maximum frequency of GIFT-NC could be higher. This is because only round numbers have been tested as clock periods. In terms of throughput per LUT (Tp/LUT), Xoodoo-N is the highest, almost doubling the Tp/LUT of GIFT-NC. So for applications where area plays an important role, Xoodoo-NC should be used over GIFT-NC.

Finally, it is clearly visible that the hash functions analyzed in this paper have way higher throughput compared to Murmur3, FNV-1a, SipHash, XORHash and NSGAHash7. The delay is also way lower. Another important point to mention, is that SipHash, XORHash and NSGAHash7 use a 32-bit input block size, which is 3x smaller compared to the hash functions in this thesis. Xoodoo-NC uses a 96-bit input block size.

Table 8: Comparison of maximum frequency, throughput, throughput per LUT and delay with related work

Hash function	Maximum frequency	Throughput (Mbps)	Tp / LUT (Mbps / LUT)	Delay (ns)
Murmur3	120.6 MHz	2,573	4.54	24.87
FNV-1a	122.9 MHz	925	1.63	130.08
SipHash [12]	182.8 MHz	1,463	1.38	21.88
XORHash [12]	627.3 MHz	2,868	9.86	11.13
NSGAHash7 [12]	184.1 MHz	1,473	18.41	21.72
Xoodoo-NC [13]	363.6 MHz	34,906	112.96	2.75
Speck-NC	166.66 MHz	16,000	37.04	6.000
Pyjamask-NC	250.00 MHz	24,000	29.59	4.000
GIFT-NC	333.33 MHz	32,000	58.61	3.000
AES-NC	250.00 MHz	24,000	10.79	4.000
Skinny-NC	200.00 MHz	19,200	8.82	5.000

5 Conclusion

The reduced round non-cryptographic versions of symmetric-key ciphers are implemented and analyzed in this thesis. The avalanche properties are calculated first to determine how many rounds from the ciphers can be reduced. Here, a 70-85% reduction is measured for all ciphers, while still having excellent avalanche properties. It is also noted that the key implementation has no effect on the avalanche properties. Afterwards, the reduced-round hash functions are implemented on the different FPGA platforms and the timings are measured. It is concluded that all hash functions have better operating frequency when they are implemented on bigger FPGA platforms. Furthermore, GIFT-NC clearly outperforms all other ciphers in terms of timing, resource utilization and throughput. However, the other ciphers are still good enough to be used as alternatives when multiple different hash functions need to be used in an application. Also, the key implementation can be used to create different versions of each hash function. But this is paired with a decrease in operating frequency for some hash functions. This can be counteracted by utilizing larger, more powerful FPGA platforms.

References

[1] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance:Building a Better Bloom Filter," 15 May 2008. [Online]. Available: https://onlinelibrary-wiley-com.bib-proxy.uhasselt.be/doi/abs/10.1002/rsa.20208.

[2] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," 4 February 2004. [Online]. Available: https://www-sciencedirect-com.bib-proxy.uhasselt.be/science/article/pii/S0196677403001913.

[3] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks and L. Wingers, "The Simon and Speck Families of Lightweight Block Ciphers," 19 June 2013. [Online]. Available: https://eprint.iacr.org/2013/404.pdf.

[4] D. Goudarzi, J. Jean, S. Kölbl, T. Peyrin, M. Rivain, Y. Sasaki and S. M. Sim, "Pyjamask," [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/pyjamask-spec-round2.pdf.

[5] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim and Y. Todo, "GIFT: A Small Present," [Online]. Available: https://eprint.iacr.org/2017/622.pdf.

[6] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[7] C. Beierle, J. Jean, S. Kölbl and G. Leander, "The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS," [Online]. Available: https://eprint.iacr.org/2016/660.pdf.

[8] Y. Yang, F. Chen, J. Chen, Y. Zhang and K. L. Yung, "A secure hash function based on feedback iterative structure," 10 December 2018. [Online]. Available: https://web-b-ebscohost-com.bib-proxy.uhasselt.be/ehost/pdfviewer/pdfviewer?vid=1&sid=14ce3c6e-298a-45bd-bf9a-60de243644fc%40pdc-v-sessmgr02.

[9] A. Sateesan, "Analyze your hash functions: The Avalanche Metrics Calculation," Medium, 6 Juli 2020. [Online]. Available: https://arishs.medium.com/analyze-your-hash-functions-the-avalanche-metrics-calculation-767b7445ee6f.

[10] C. Esébanez, Y. Saez, G. Recio and P. Isasi, "Performance of the most common non-cryptographic hash functions," *Software: Practice and Experience,* pp. 681-698, 2014.

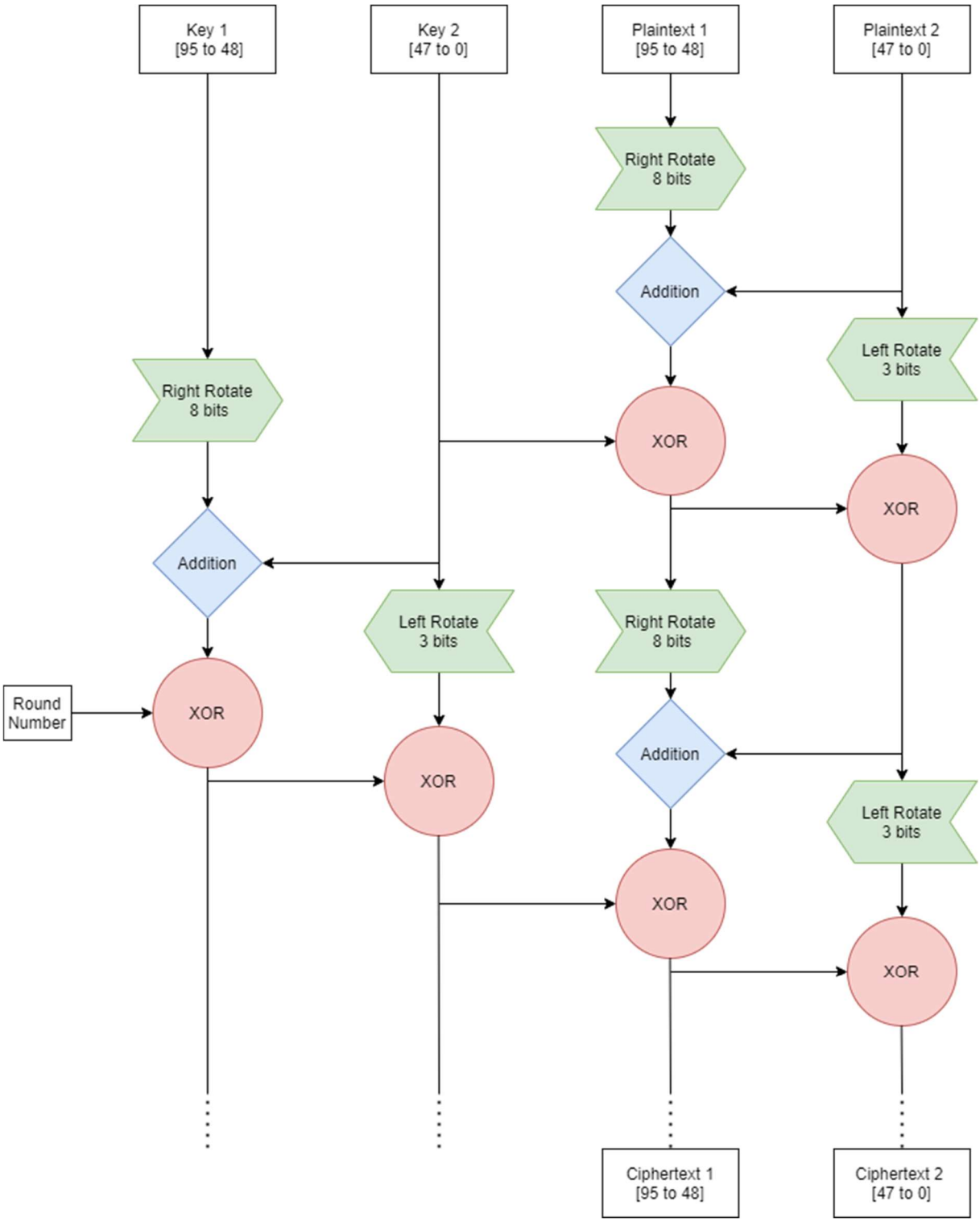
[11] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake and T. Hansen, "The FNV non-cryptographic hash algorithm," 2011.

[12] D. Sekanina and L. Grochol, "Fast Reconfigurable Hash Functions for Network Flow Hashing in FPGAs," in *NASA/ESA Conference on Adaptive Hardware and Systems*.

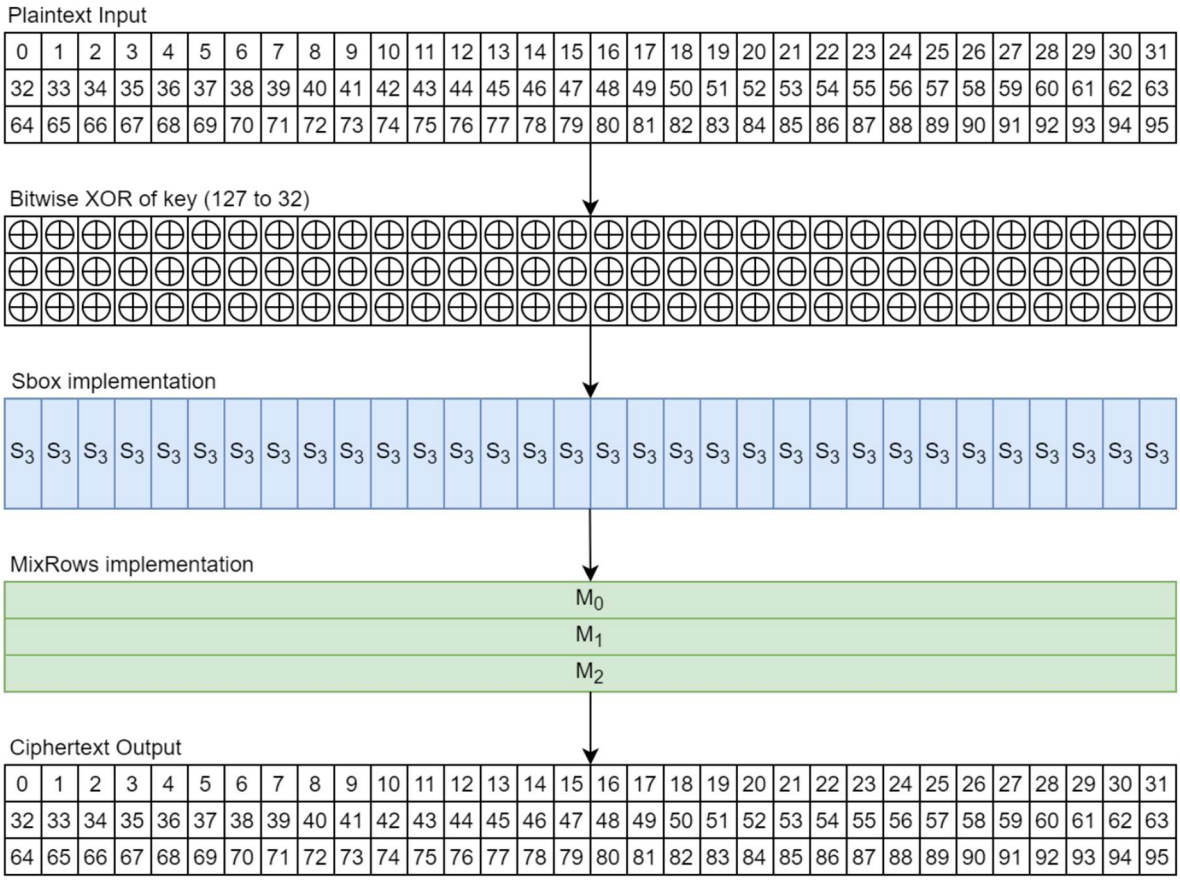
[13] A. Sateesan, J. Vliegen, J. Daemen and N. Mentens, "Novel Bloom filter algorithms and architectures for ultra-high-speed network security applications," in *23rd Euromicro Conference on Digital System Design*, 2020, pp. 262-269.

List of Appendices

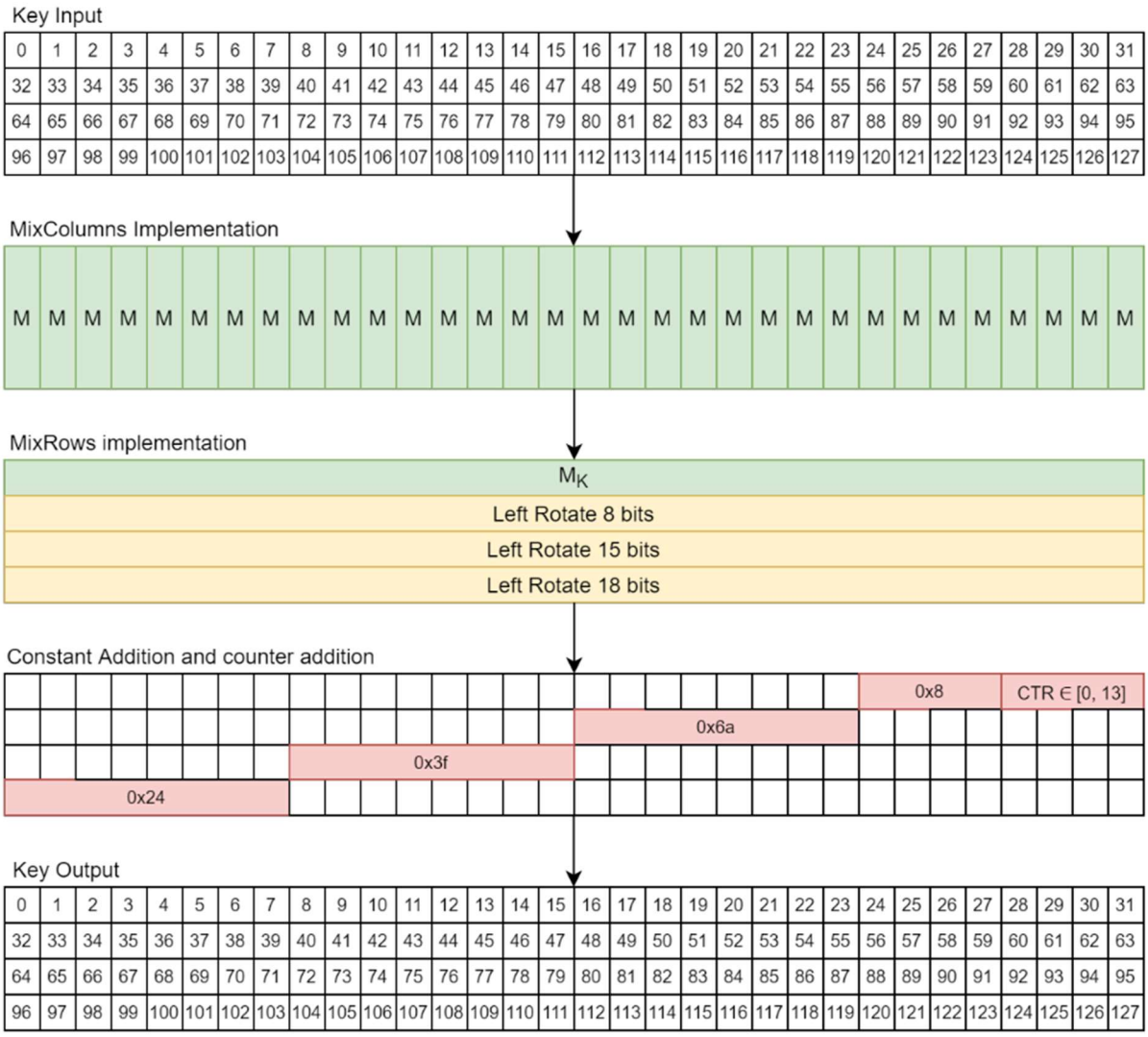
Appendix A: Diagram of the first two rounds of Speck



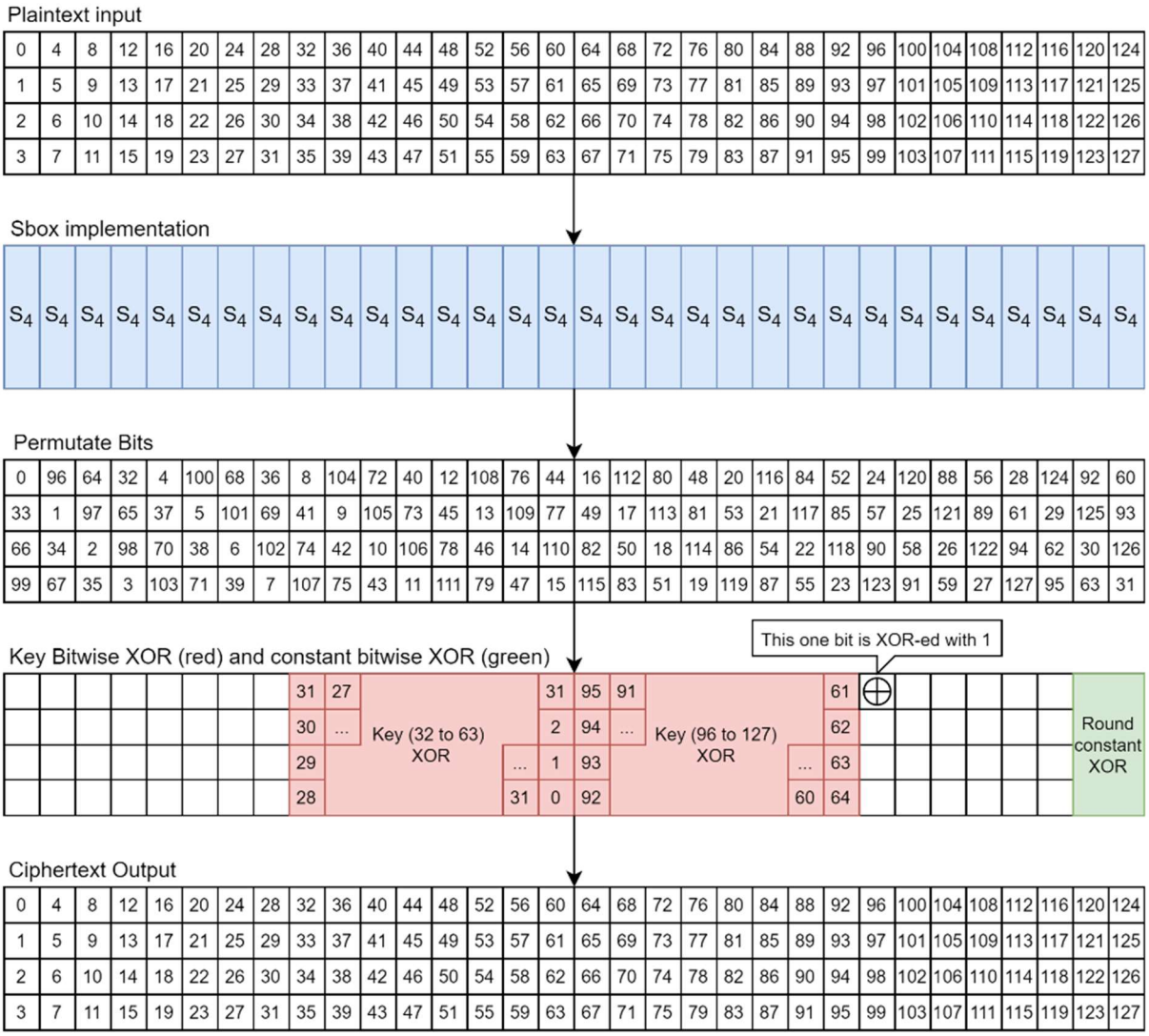
Appendix B: Visual representation of a round in the Pyjamask-96 encryption



Appendix C: Visual representation of a round of the key schedule in Pyjamask



Appendix D: Visual representation of a round in the GIFT-128 encryption

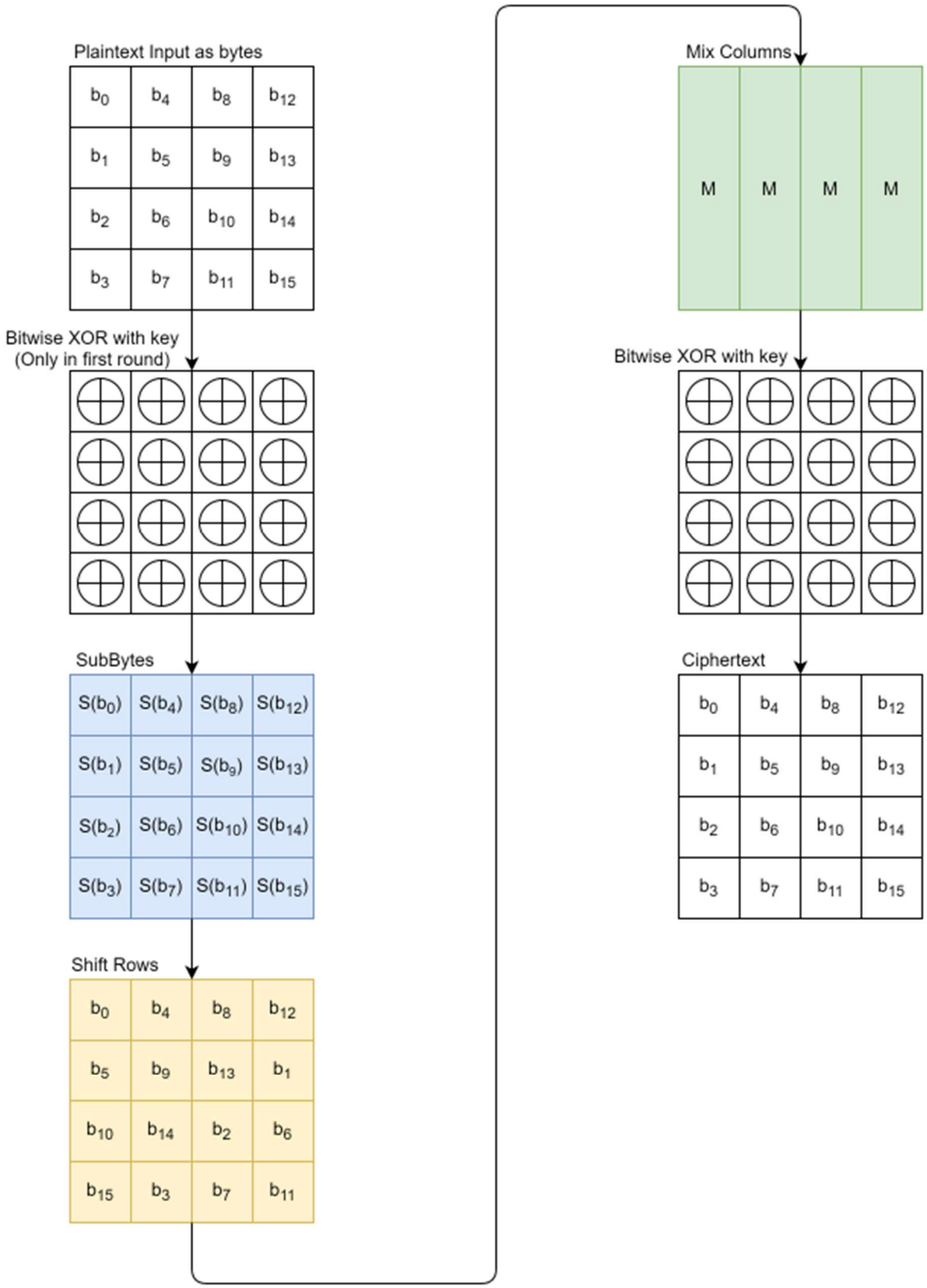


Appendix E: Sbox for the SubBytes operation in AES

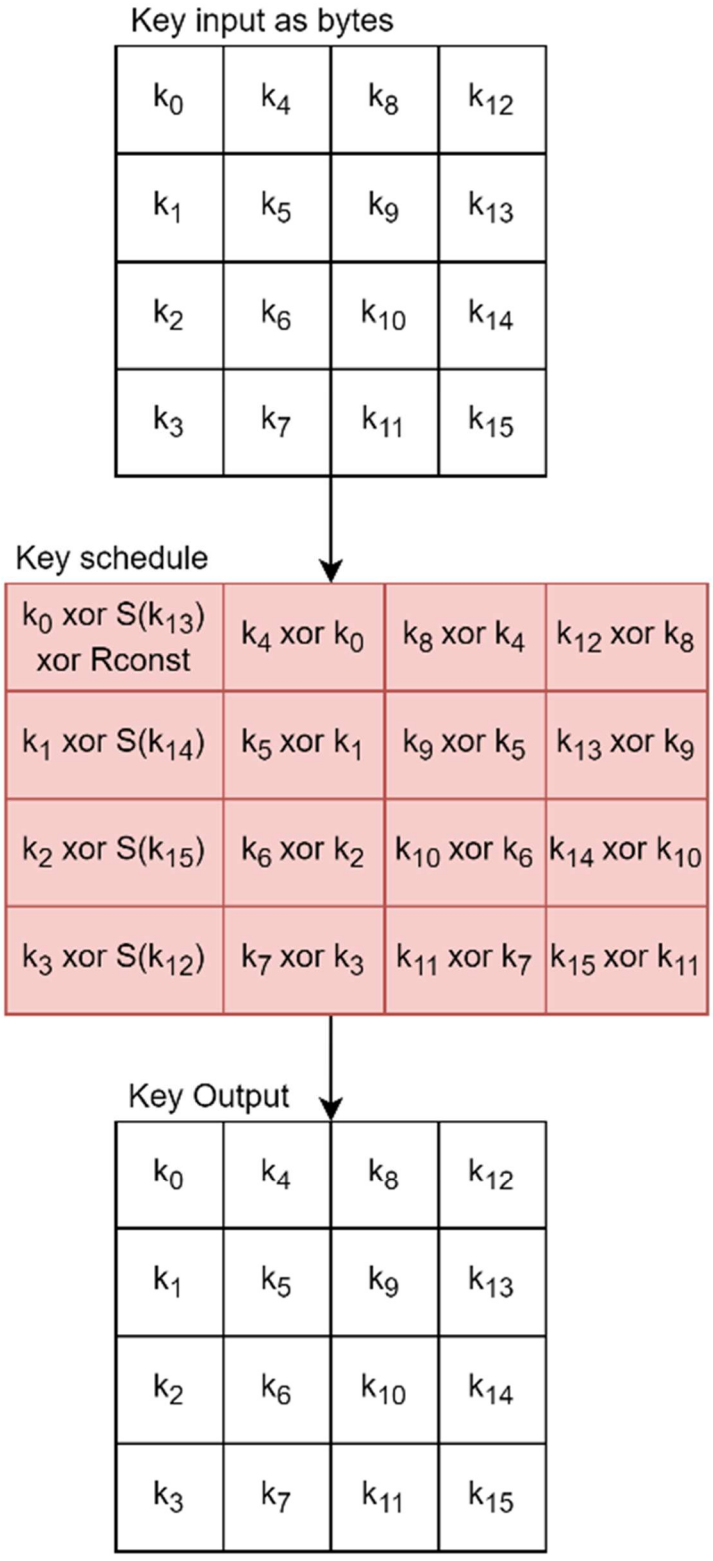
SBox Lookup Table

MSB \ LSB	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Appendix F: A single encryption round in AES



Appendix G: A single round of the key schedule in AES



Appendix H: Sbox for the SubBytes operation in Skinny

SBox Lookup Table

MSB \ LSB	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	65	4c	6a	42	4b	63	43	6b	55	75	5a	7a	53	73	5b	7b
1	35	8c	3a	81	89	33	80	3b	95	25	98	2a	90	23	99	2b
2	e5	cc	e8	c1	c9	e0	c0	e9	d5	f5	d8	f8	d0	f0	d9	f9
3	a5	1c	a8	12	1b	a0	13	a9	05	b5	0a	b8	03	b0	0b	b9
4	32	88	3c	85	8d	34	84	3d	91	22	9c	2c	94	24	9d	2d
5	62	4a	6c	45	4d	64	44	6d	52	72	5c	7c	54	74	5d	7d
6	a1	1a	ac	15	1d	a4	14	ad	02	b1	0c	bc	04	b4	0d	bd
7	e1	c8	ec	c5	cd	e4	c4	ed	d1	f1	dc	fc	d4	f4	dd	fd
8	36	8e	38	82	8b	30	83	39	96	26	9a	28	93	20	9b	29
9	66	4e	68	41	49	60	40	69	56	76	58	78	50	70	59	79
a	a6	1e	aa	11	19	a3	10	ab	06	b6	08	ba	00	b3	09	bb
b	e6	ce	ea	c2	cb	e3	c3	eb	d6	f6	da	fa	d3	f3	db	fb
c	31	8a	3e	86	8f	37	87	3f	92	21	9e	2e	97	27	9f	2f
d	61	48	6e	46	4f	67	47	6f	51	71	5e	7e	57	77	5f	7f
e	a2	18	ae	16	1f	a7	17	af	01	b2	0e	be	07	b7	0f	bf
f	e2	ca	ee	c6	cf	e7	c7	ef	d2	f2	de	fe	d7	f7	df	ff

Appendix I: Round Constant Lookup Table for Skinny

Round Number

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
01	03	07	0F	1F	3E	3D	3B	37	2F	1E	3C	39	33	27	0E	1D	3A	35	2B	16	2C	18	30	21	02	05	0B	17	2E	1C

Round Constant (RC) Lookup table

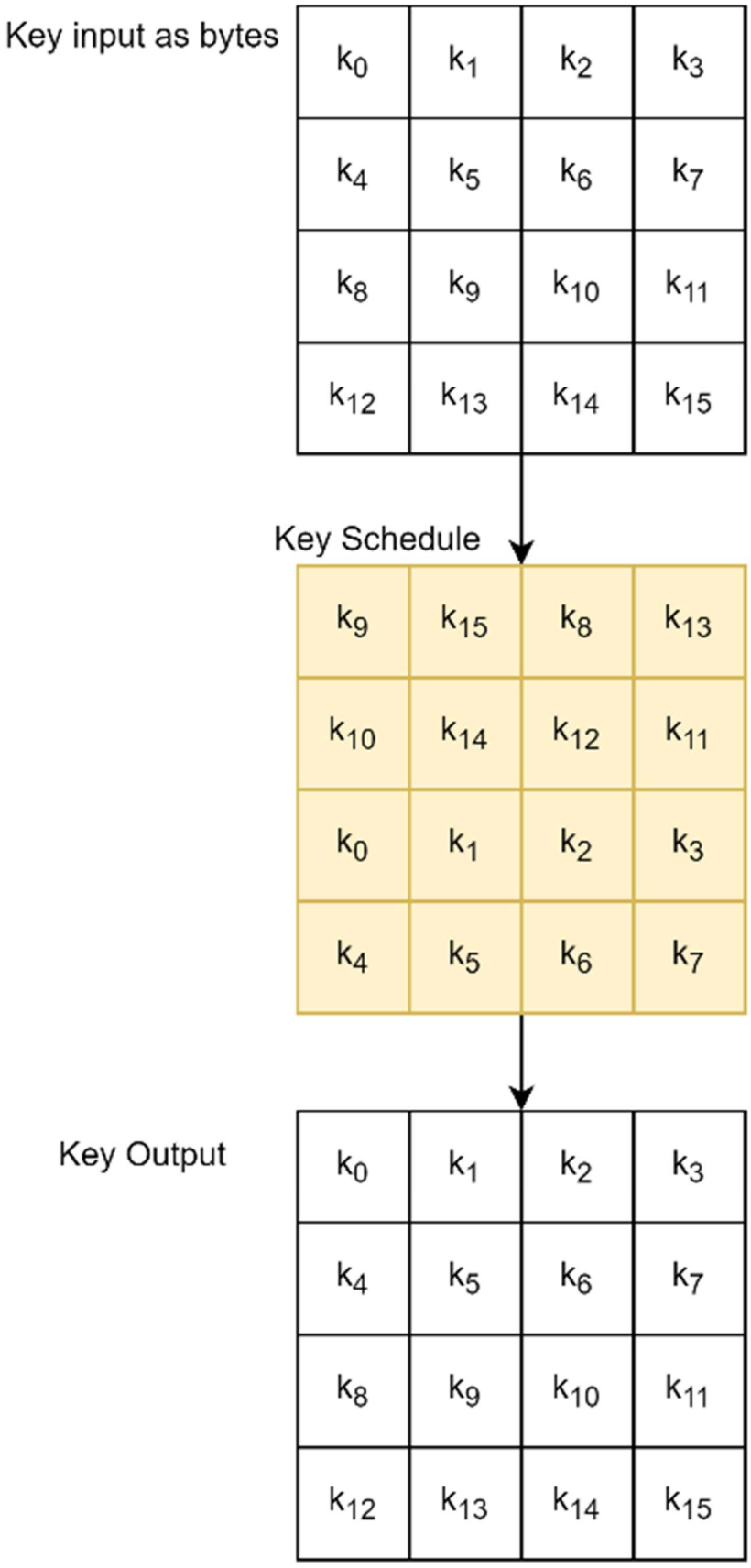
Round Number

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62
38	31	23	06	0D	1B	36	2D	1A	34	29	12	24	08	11	22	04	09	13	26	0C	19	32	25	0A	15	2A	14	28	10	20

Round Constant (RC) Lookup table

Appendix J: A single encryption round in Skinny

Appendix K: A single round of the key schedule in Skinny



Appendix L: Diagrams of the hardware implementation