DESIGN OF DECRYPTION ROUTINE WITH MINIMAL OVERHEAD IN UGHTWEIGHT BLOCK CIPHERS

By

Abdur Rebman Raza Kban

A Cbeaia mbmil!ed to the &.cul!J ofDepartmeat ofllllfmmation Securi!J, Military Col'l-&e of Sipall, Natiooal Ulli.wrai!Jof Scienra llllld Teclmology, hlemabad, PatiSiaiJ,iD partial fnlfillment of tbe ftfilliJri\Ji+nh for the degree of MS iJJ. Information Security

Sepcemll«2019

THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS Thesis written by **Abdur Rehman Raza Khan** Registration No. **00000240986,** of Military College of Signals has been vetted by undersigned, found complete in all respect as per NUST Statutes/ Regulations, is free of plagiarism, errors and mistakes and is accepted as partial, fulfillment for award of MS degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the said thesis.

Signature: _____

Supervisor: Dr. Mehreen Mzal

Date: _____

Signature (HoD): _____

Date: _____

Signature (Dean/ Principal): _____

Date: _____

DECLARATION

I hereby declare that no portion of work presented in this thesis has been submitted in support of another award or qualification either at this institution or elsewhere.

DEDICATION

This thesis is dedicated to

MY FATHER, GRAND PARENTS, BELOVED TAYA AND KHALA,

MAY ALLAH GRANTS THEM HIGH RANKS IN JANNAT. (Ameen)

ACKNOWLEDGMENTS

I am thankful to ALLAH who bestowed me with the strength to accomplish this thesis.

I am in debt to my Mother, who sacrificed her life in up bringing me and my brother.

I am thankful to my Brother and Wife for their love and encouragement.

I am grateful to Dr. Mehreen Mzal and committee members for supervising my thesis work.

In the end, I would like to appreciate all of my friends for their endless support, specially Salman, Abbas, Khawir and Ikram.

Raza

ABSTRACT

The rapid development of wireless sensor networks and RFID technologies is transforming every aspect of human life ranging from personal fitness companions to goods tracking in the supply chain industry. The sensitive nature of the data which these devices handle has created the demand for lightweight cryptography as existing standards of security and privacy are impractical for these tiny resource-constrained devices. In addition to the limitation of computational resources, the challenges for lightweight block cipher design include low gate count, power consumption, cycle count, latency, resistance to side-channel attacks and support for decryption with minimal overhead on top of encryption. Meeting all of these additional constraints while maintaining the required level of security is a challenging task. This Master's thesis focuses on reducing the decryption cost of Substitution Permutation Network (SPN) lightweight block ciphers. We describe techniques to implement a lightweight block cipher in both hardware and software platforms followed by how to incorporate the decryption routine with encryption. The traditional way to solve this problem is either by reducing the inverse implementation cost of existing components or constructing new components which support inversion with minimal overhead. Our contribution spans over both. First, we find methods to efficiently implement inverse of a Maximum Distance Separable (MDS) matrix with minimum additional cost. On average, our methods enable this implementation with 40% lesser xor operations. Moreover, in the best case, only 12 additional xor operations are required to support inverse matrix multiplication. Secondly, we define constructions of non-involutive cryptographic components for both confusion and diffusion layers which use similar implementation for its inverse. This helps in further reducing the implementation cost for the inverse transform of the cryptographic primitives. In the end, based on these primitives, we propose an SPN structure to support decryption routine with minimal overhead.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

Advance Enccryption Standard	AES
Application-Specific Integrated Circuit	ASIC
Ciphertext Feed Back	CFB
Circulant Permutation Inverse	CPI
Cyclic Block Chain	CBC
Derived Implementation	DI
Diagonal-Serial Invertible	DSI
Field-Programmable Aate Array	FPGA
Galois Counter Mode	GCM
International Organization for Standardization	ISO
Internet of Things	loT
Involutory Circulant Maximum Distance Separable	ICMDS
Light Encryption Device Least	LED
Significant Bits Maximum	LSB
Distance Separable Message	MDS
Authentication Codes Mixed	MAC
Implementation	MI
Most Significant Bits	MSB
National Institote of Standards and Technology	NIST
Output Feed Back	OFB
Row Permutation Inverse	RPI
Shortest Linear straight-line Program	SLP
Side Channel Attacks	SCA
Single Instruction, Multiple Data	SIMD
Substitution Permutation Network	SPN

Chapter 1

INTRODUCTION

The proliferation of loT devices, ranging from personalized fitness companions to smart home sensors is gradually transforming every aspect of human life in fundamental and diverse ways. Furthermore, the amalgamation of loT devices with cloud technology and big data is bringing together physical, industrial and biological worlds [1]. According to a prediction by Gartner, there will be 20 billion loT devices connected to the Internet by year 2020 [2]. These loT devices are constantly producing huge volumes of data which is shared between devices for collaboration and forming ubiquitous systems. These networks of loT devices also necessitate central processing for state-of-the-art intelligent services such as analytics, mining and prediction. This requirement is met by integrating loT devices with cloud-based technology resulting in a scalable, robust and highly available collaboration which entails huge potentials and benefits at individual, society as well as global levels. A key concern regarding the loT devices is the nature of the data accessed and shared by these devices with one another as well as over the cloud infrastructure. This data includes sensitive personal or mission critical information for which the most significant factors are privacy and security. Lack of privacy and security diminishes the efficacy as well as utility of the loT devices. This, in tum, acts as the primary barrier which needs to be provably surpassed for practical utilization ofloT.

The peculiar cloud-based loT ecosystem compounds the privacy and security requirements. A balanced approach is sought that deals with resource-constrained loT devices at one end and performance requirement for large number of simultaneous cloud-connected devices at the other. Existing standards of National Institute of Standards and Technology (NIST) for encryption (AES [3]) and hash function (SHA-m [4]) can not be efficiently implemented in resource-constrained environments [5]. Therefore, the more suitable options for these tiny loT devices are low cost, lightweight cryptographic block and stream ciphers, hash functions and Message Authentication Codes(MAC) [6, 7, 8, 9].

The paramount requirement in the realm of cloud-based loT scenario is data confidentiality

i.e. ensuring that all data is transferred amongst loT devices and with cloud servers in an encrypted manner. The prime candidates for data confidentiality are light weight block and stream ciphers. Block ciphers are a versatile cryptographic primitive which have an upper edge over stream ciphers. They can act as a stream cipher by running in counter mode and provide authentication as in Galois Counter Mode (GCM) [10]. The high clock cycle count for initialization phase of the stream ciphers makes them less suitable for hardware implementation in constrained devices where key changes occur frequently [11]. In addition to it, the art of designing block ciphers seems to be more understood and well established as compared to stream ciphers [12].

Over the past decade a number of lightweight block ciphers have been designed such as IDGHT [13], KLEIN [14], LED [15], MIBS [16], SPARX [17], SKINNY [18]. The two block ciphers CLEFIA [19] and PRESENT [12] form part of ISO standard for lightweight block ciphers [20]. Mostly the lightweight block ciphers are designed to support compact hardware implementation in terms of gate count and power consumption. However, FeW [21], ITUbee [22], Robin and Fantomas [23] are also suitable for implementation in software based platforms. For software implementations, the goal is to reduce the memory requirement and increase the throughput. Various designs support additional constraints such low latency, masked implementation and support for both encryption and decryption with minimal overhead. PRINCE [11] is a low latency block cipher for pervasive computing applications. It supports encryption of data in hardware within one clock cycle with a very competitive chip area. Moreover, the implementation cost of decryption routine on top of encryption is negligible. The block cipher Zorro [24] is a variant of AES which is easy to mask. It makes the implementation to resist against side channel attacks(SCA). It is often the case that block ciphers are first proposed and then masking schemes are constructed. However the designers of PICARO [25] took the reverse approach: for a proven masking scheme [26], design a block cipher according to masking constraints. Few designs have been proposed which improves upon or combined the ideas of existing lightweight block ciphers like SIMECK [27] combines the best features of two ciphers Simon and Speck [28]. I-PRESENT [29] is an involutive design based on PRESENT [12]. The involution part is inspired from block cipher PRINCE [11] and encryption is identical to decryption except the round keys are used in reverse order.

1.1 Problem Statement

Lightweight block ciphers are often designed and specified for encrypt-only routine and efforts are made to keep the encryption specification as lightweight as possible. As these ciphers are intended for resource constrained devices, it is assumed that these lightweight primitives will be employed with a block cipher mode of operation which does not necessitate the presence of block cipher decryption-core [12]. However it may not be the case always. The loT devices may have to be deployed in an already existing network which uses mode of operation such CBC that requires implementation of the decryption routine [30]. Feistel structures inherently support decryption using the same encryption circuit by using round keys in reverse order [28, 31, 19]. However, these have slow diffusion as only one half of the state is processed in each round. This leads to more executions of the round function to achieve same level of security as compared to SPN structures [32]. Moreover, output of the round function is mixed with unprocessed half of the state by xor operations which may increase the length of critical datapath [33]. To overcome these limitations of feistel structures, few involutive SPN ciphers were proposed which used the same datapath for encryption and decryption [34, 35, 32]. But involutive components have large number of fixed points which makes them distinguishable from random permutations [36] and vulnerable to Invariant Subspace and Related key Attacks [37, 38]. One the other hand, the block ciphers PRINCE [11] and I-PRESENT [29] support decryption with minimal overhead by incorporating inverse round transformation in the encryption path (reflection ciphers) i.e $Ek = F.M.F^{-1}$ [36]. The decryption is performed by using round keys in reverse order on the same circuit but this also increases the implementation cost of encryption routine. The lightest implementation of these two ciphers require 2953 and 2796 GE where as encrypt-only implementation of PRESENT needs 1570 GE only [39, 33]. Thus, implementation cost becomes almost equal to as in case of supporting both encryption and decryption by encrypt-only designs.

1.2 Research Objectives

The research in this thesis aims to achieve the following objectives:

- Study of software and hardware implementations of lightweight block ciphers.
- Customization of an existing lightweight block cipher to support decryption with minimum overhead.

- Comparative analysis of proposed and existing structure in terms of security and efficiency.

1.3 Contributions

The significant contributions of this thesis are summarized as

- Detailed analysis of hardware and software implementations of lightweight block ciphers and impact of decryption routine implementation on resource consumption.
- Construction of lightweight MDS matrices and implementation methods to support inverse Matrix multiplication with lesser implementation cost.
- Construction of lightweight non-involutive 8-bit bijective sBox that supports self inversion after a linear operation on the output.
- Design of d-SPN, an SPN structure to support decryption with minimal overhead

1.4 Thesis Outline

Including the current Introduction chapter, this research work is composed of six chapters. Outline of the remaining chapters is as follow:

- Chapter 2 provides a detail account on hardware and software implementations of the block ciphers. Different implementation techniques are explained for a lightweight block cipher LED [15]. In the end, implementation of the decryption routine of the block cipher is explained.
- Chapter 3 deals with construction and implementation of lightweight MDS matrices over $GF(2^4)$. It provides methods to implement the inverse matrix multiplication with 40% lesser xor operations.
- Chapter 4 studies the mixColumns operation of the AES [3] block cipher and provides lightweight alternatives. It defines constructions to make non-involutive MDS matrices over $GF(2^8)$ from lightweight MDS matrices over $GF(2^4)$ with reduced implementation cost. Moreover, these constructions support inverse transform with little additional implementation cost.

- Chapter 5 presents dSPN which is an SPN structure that supports decryption with minimal overhead. The dSPN is constructed from non-involutive components that support self inversion after a linear operation is performed on the output.

- Chapter 6 concludes the thesis and highlights the directions for future work.

Chapter2

IMPLEMENTING A LIGHTWEIGHT SPN BLOCK CIPHER

2.1 Introduction

Implementations of cryptographic primitives are categorized into software and hardware based implementations. The former deals with implementations in general purpose processors and micro-controllers while the later is intended for dedicated devices such as FPGAs and ASICs. In this chapter we provide a comprehensive account on different software and hardware based implementation techniques. We explain these implementations for LED [15] block cipher because it is a Substitution Permutation Network (SPN) with architecture similar to AES [3]. AES is NIST's standard for block ciphers and in fact, the most extensively studied design. Its wide trail design strategy provides concrete security bounds against differential and linear cryptanalysis [40]. Over the years, many lightweight block ciphers including LED have been designed with a structure similar to AES such as KLEIN [14], Midori [32], Mysterion [41], Skinny [18], and Zorro [24]. Therefore, explaining implementation techniques for LED helps in covering a large range of lightweight block ciphers to which these techniques can be easily extended. In addition to it, LED employs recursive MDS matrix in permutation layer which helps in realizing the 4-bit Serial implementation. Showing the real essence of serial implementation may have not been possible if some other SPN block cipher would have been selected.

2.2 LED Block Cipher

LED is lightweight block cipher which supports 64-bit block length and key lengths of 64 to 128 bits in multiples of 4. It does not employ any key schedule, rather the user provided master key is used as-is where required. Moreover, the round key is mixed into the plaintext after every four rounds, called STEP. This helps in realizing compact hardware implementation. Although the non-existence of key-schedule seems dangerous and makes the cipher vulnerable to different types of attacks [42, 43], special care has been taken in the design of LED to thwart against these e.g. resistance to slide-attacks [15].

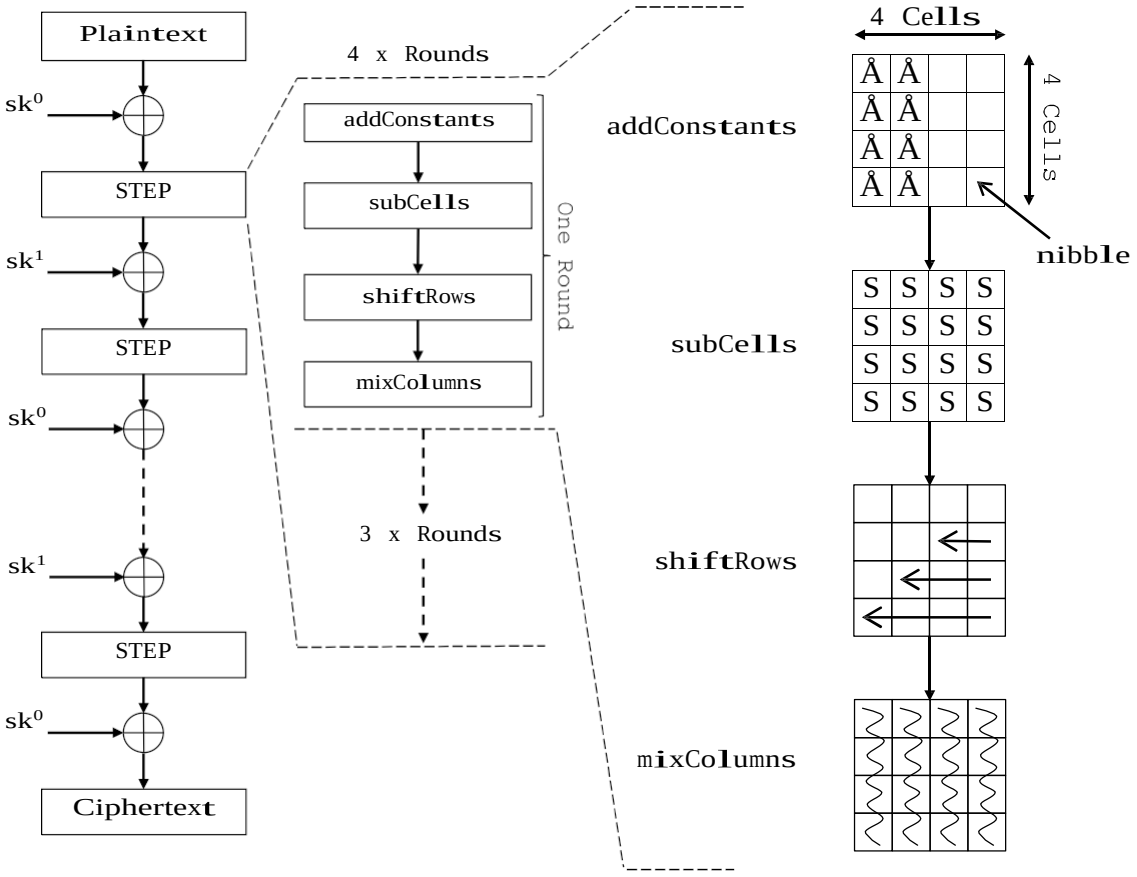


Figure 2.1: Encryption Operation of LED Block Cipher.

The 64-bit plaintext block is conceptually arranged in a 4×4 state matrix of 16 nibbles(4-bit) as

$$\begin{bmatrix} n_0 & n_1 & n_2 & n_3 \\ n_4 & n_5 & n_6 & n_7 \\ n_8 & n_9 & n_{10} & n_{11} \\ n_{12} & n_{13} & n_{14} & n_{15} \end{bmatrix}$$

Each nibble is an element of $GF(2^4)$ with underlying polynomial for finite field multiplication as $X^4 \oplus X \oplus 1$. Then a series of round transformations are applied to the state matrix. Each round consists of four operations namely addConstants, subCells, shiftRows and mixColumns. Number of rounds depend upon the key length. For 64-bit key length, it has 32 rounds and for key lengths greater than 64 and upto 128-bit, LED performs 48 round transformations. Figure 2.1 shows the encryption operation of LED block cipher.

addConstants. In each round, 8 bits of the key size constant and 6 bits of the round constant

are mixed with the first and second column of the state matrix by xor operation.

subCeUs. It updates the sixteen nibbles of the state matrix in each round, using 4-bit sbox of the block cipher PRESENT [12].

shiftRows. The shiftRows operation performs left cyclic rotation for i number of positions on *ith* row of the state matrix.

mixColumns. Each column of the state matrix is updated by multiplying it with a 4 x 4 diffusion matrix *M*. It is possible to implement this matrix multiplication serially in 4 clock cycles by using a 4 x 4 matrix A where

$$A = \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 4 & 1 & 2 & 2 \end{vmatrix} \quad ===\} \quad A4 = \begin{vmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ b & e & a & 9 \\ 2 & 2 & f & b \end{vmatrix} = M$$

2.3 Software Implementations

The available software based implementation techniques to implement a block ciphers include lookup-table based (LUT), bit-sliced and use of Single Instruction Multiple Data (SIMD) instructions. The lookup-table based implementation is done by pre-computing the small chunks of data, then selecting and aggregating it at runtime. The bit-slice technique introduced in [44], implements the block cipher without lookup tables. It involves breaking down the block cipher into logical bit operations in order to perform N parallel encryptions on anN-bit microprocessor [45]. The use of SIMD instructions for accelerating the AES was presented in [46]. Precisely, the vector permute (vperm) instruction set is used to perform parallel table lookups in order to increase the throughput. This technique has later been applied to various block ciphers for accelerated implementations and resistance against side channel attacks [47, 48, 49, 31].

2.3.1 4-bit Serial Implementation

The nibbles of the state matrix, key and sbox are stored in three arrays of sixteen bytes. The key mixing is performed by applying xor operation on corresponding nibbles of the state and key. The subCells operation requires sixteen lookups to update the state by sbox values. The subCells and shiftRows operation are performed together by storing values from sbox lookup at appropriate positions in the state matrix while keeping in view the shiftRows

```
byte xTimes(byte y) {
   y << 1;
   if ((y & 0xl0)    0xl0)
      y  - poly;
   return (y & 0xOF)


byte xxTimes(byte y)
   return xTimes(xTimes(y));
```

Listing 2.1: xTimes- Multiplying y by 2 and 4 in $GF(2^4)$.

operation. The mixColumns operation employs the serial matrix A instead of MDS matrix M because it consists of only 3 distinct elements i.e 1, 2, 4. This helps in reducing the memory cost, but now each column of the state matrix is multiplied 4 times with the serial matrix A. Multiplication with 1 does not require any resources and multiplication by 2 is performed by left shift of one bit position and a conditional xor operation. Listing 2.1 shows multiplication of a nibble by 2 and 4 in $GF(2^4)$.

2.3.2 4-bit LUT Implementation

The 4-bit LUT Implementation is similar to 4-bit Serial implementation except it employs multiplication with MDS matrix M in mixColumns operation. The state, key and values of the sbox are stored in three byte arrays of size sixteen each. The operation addKey and addConstants is performed by 16 and 8 xor operations respectively. MDS matrix M consists of ten distinct elements other than 1. In order to perform multiplication with these, ten arrays of size sixteen are used. Multiplication results of every element in $GF(2^4)$ with the ten distinct elements of MDS matrix M are precomputed and stored in these arrays. Thus 4-bit LUT implementation requires additionall60 bytes of memory as compared to 4-bit serial implementation. Moreover, the effect of sbox is combined with the matrix multiplication by computing the multiplication tables for each distinct element m of the MDS matrix as

$$mulTablemf.i] = m \times sbox[j] \quad j : 0\text{---+}\ 15$$

Combining the effect of sbox into multiplication tables helps in implementing subCells, shiftRows and mixColumns operation together. This reduces the number of lookup operations required to implement one round of the encryption but it requires more memory to store the multiplication tables.

2.3.3 8-bit LUT Implementation

In this implementation, the two consecutive nibbles of the state matrix and key are joined together to make a byte as

$$\begin{array}{cc} \text{n1sln14} & \text{n3ln2} \quad \text{n1lno} \\ \hline b7 & \text{bl} \quad bo \end{array}$$

$$\begin{vmatrix} no & \text{n1} & \text{n2} & \text{n3} \\ n4 & ns & n6 & n7 \\ ns & ng & nlO & nn \\ \text{n12} & \text{n13} & \text{n14} & \text{n1s} \end{vmatrix} \quad \text{--+} \quad \begin{vmatrix} bo & \text{bl} \\ \text{b2} & ba \\ b4 & bs \\ b6 & b7 \end{vmatrix}$$

Then the values of the state matrix and key are stored in two arrays of the size 8 bytes each and the key mixing is completed by 8 xor operations. The subCells operation employs a larger lookup table of size 256 bytes to store sbox values. This larger sbox is computed as

$$sbox8[b] = sbox[bmsb(4)]1 \quad sbox[bzsb(4)]i \quad \text{b: } 0 \text{ -+ } 255$$

With this byte oriented sbox, the complete state is updated by 8 lookup operations. Similarly, the multiplication tables are computed for 8-bit input and output. These larger multiplication tables require 2560 bytes of memory which is far more as compared to 4-bit LUT implementation. But the number of lookup operation is reduced to half. In previous implementations, the shiftRows operation was performed by moving values to the corresponding indices, however in this implementation the shiftRows operation is performed as follows

$$\begin{vmatrix} b_0 & b_1 \\ b_2 & b_3 \\ b_4 & b_5 \\ b_6 & b_7 \end{vmatrix} \quad \begin{vmatrix} bo & \text{bl} \\ \text{(b3 } 4)\text{1(»4)} & \text{(b3 »4)|(b2 } 4) \\ bs & b4 \\ (b7 \ 4)\text{l(b6 »4)} & (b7 \ 4)\text{l(b6 »4)} \end{vmatrix}$$

2.3.4 16-bit LUT Implementation

The sixteen nibbles of the state matrix and key are stored in two ushort(16-bit unsigned integer) arrays of length 4 as

$$\begin{array}{cc} n_{13}|n_9|n_5|n_1 & n_{12}|n_8|n_4|n_0 \\ \hline & uo \end{array}$$

$n_{15}|n_{11}|n_7|n_3 \qquad n_{14}|n_{10}|n_6|n_2$

The addKey operation uses 4 xor operations to mix key into the state. This particular packing of state nibbles into ushort words helps in performing rnixColumns operation with lesser operations. The 16-bit LUT implementation uses 4 lookup tables $(T_0, T1, T_2, T_3)$ of 4-bit input and 16-bit output. Each lookup table $T;$ takes input of ith 4-bit nibble of 16-bit string $(U = U_3 U2 U1 U_0)$ and outputs multiplication of nibble $U;$ with all four elements of the ith column of matrix $M.$ Then 16-bit outputs from all four lookup tables are xored together to produce output of the matrix multiplication.

$$M \times U = U' \text{ -----+ } \begin{vmatrix} 4 & 1 & 2 & 2 \\ S & 6 & 5 & 6 \\ b & e & a & 9 \\ 2 & 2 & f & b \end{vmatrix} \times \begin{vmatrix} Uo \\ u1 \\ u2 \\ Ua \end{vmatrix} = \begin{vmatrix} U_0' \\ U_1' \\ U \\ U_3' \end{vmatrix}$$

$$\text{To}[Uo] = 2.Uolb.Uoi8.Uoi4.Uo$$
$$\text{Ell} \quad T1[U1] = 2.Ulle.Uli6.U1ILU1$$
$$\text{Ell} \quad T2[U2] = f.U2Ia.U2I5.U2I2.U2$$
$$\text{Ell} \quad Ta[Ua] = b.Uai9.Uai6.Uai2.Ua$$
$$U' = U\ IUIUfiU$$

With these pre-computed tables, rnixColumns operation is performed by 4lookups and 3 xor operations. Moreover, these tables are computed after incorporating lookups from the sbox, thus subCells and shiftRows operation are combined with the matrix multiplication.

Similar to 16-bit lookup table implementation, it is possible to implement the encryption operation for 32 and 64-bit words. These implementations will store the internal state in 32 and 64-bit words and will employ lookup tables with larger output. Table 2.1 shows number of basic operations and memory bytes required to implement LED-64 encryption by the above mentioned lookup table methods.

2.3.5 Vector Permute Implementation

SlMD engine present in the modem computers allows implementation of one operation on multiple data elements in parallel. Authors in [46] extended the use of SMID engine to bock ciphers. Basically the idea is to use Vector Permute (vperm) instructions to implement

Table 2.1: Memory and Number of basic Operations required to implement LED-64 Encryption.

#	Type	Memory	xor	and	shift	lookup	xTimes
1	4-bit Serial	52	1936	64	32	576	2048
2	4-bit LUT	244	1936	0	0	2112	0
3	8-bit LUT	2944	1096	0	256	1152	0
4	16-bit LUT	194	484	512	384	544	0

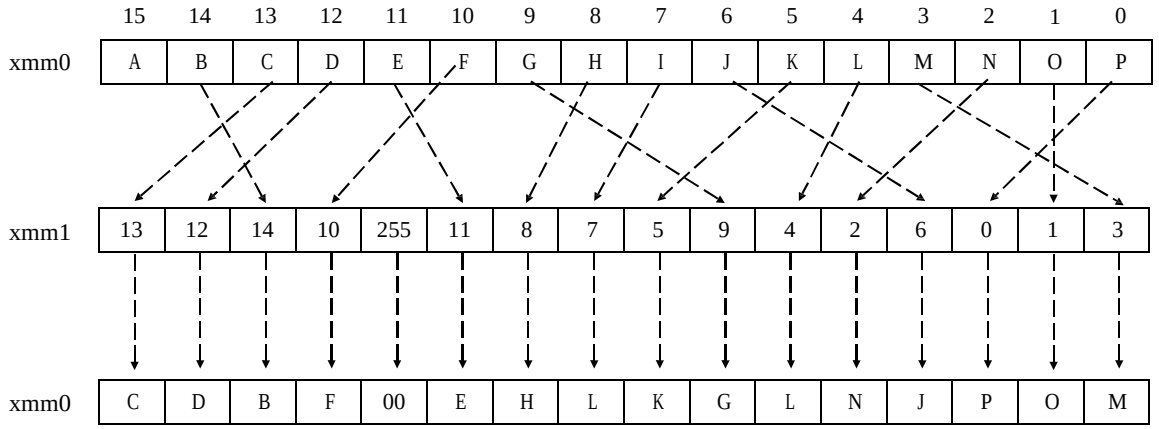


Figure 2.2: Vector Permute Instruction pshufb operating on Registers xmm0 and xmm1.

parallel table lookups to accelerate the performance. The two added benefits of implementation by vperm instructions is increase in throughput and resistance against timing based side channel attacks. At present, technique is applicable to small size lookup tables only i.e (upto 4-bit lookup table). The byte shuffling instruction pshufb is used to perform the permutation using two 128-bit (16 byte) registers xmm0 and xmm1. The contents of the 4-bit sbox are loaded into xmm0 register by placing each nibble into a byte. The current state nibbles are loaded into xmm1 register and the shuffled bytes are stored in xmm0 register. If most significant bit of a byte in xmm1 register is set to 1, then resultant byte in xmm0 is set to zero. Only low 4-bits of each byte in xmm1 are used as permutation mask. Figure 2.2 shows the permutation performed by pshufb instruction on xmm0 and xmm1 registers.

LED block cipher employs 4-bit sbox which has 16 possible input and output values. Since vperm also performs the sixteen index shuffle, the subCells operation can be effectively performed by the pshufb instruction. The sixteen nibbles of the state matrix and sbox values are stored in lower 4 bits of the sixteen byte register xmm1 and xmm0 respectively. Then pshufb instruction performs lookup on complete state matrix in parallel. The same can be

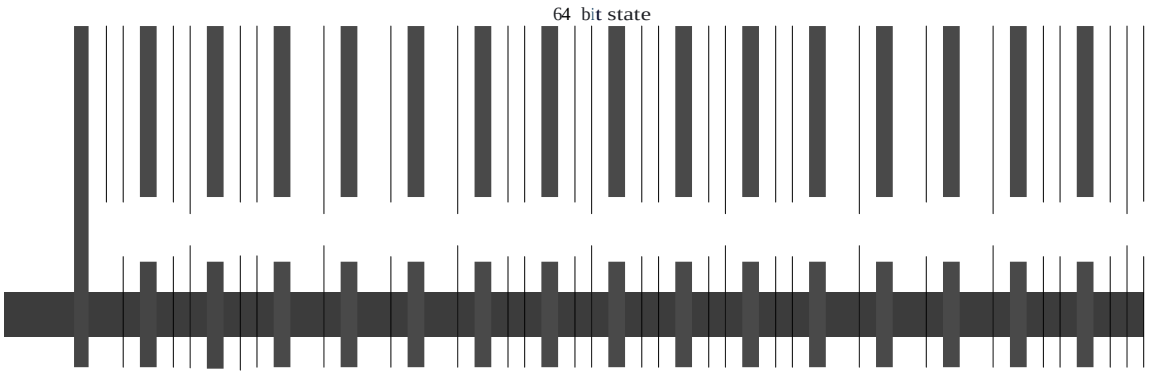


Figure 2.3: Packing of 64 input blocks for Bitslice Implementation.

extended to implement mixColumns operation. In 4-bit LUT implementation, we performed matrix multiplication by performing lookups from precomputed arrays of size sixteen which can also be done using pshufb instruction. Similarly shiftRows can be performed by pshufb instruction to shuffle the nibbles from old to the new indices. The remaining operations like addConstants and round key mixing are performed by xor instruction.

2.3.6 Bitslice Implementation

In bitslice implementation complete block cipher is viewed as a gate circuit. Multiple input blocks are split into N-bit chunks and encryption operations are performed on these chunks in parallel. It is possible to choose different values for N depending upon the choice of underlying block cipher. Setting N=l is similar to performing hardware implementation using 2-bit input gates. All the encryption operations are then performed using basic operations such as AND, XOR, NOT etc. For N=l, the 64 input blocks of the plaintext are packed in 64 words as shown in Figure 2.3. First bit of ith plaintext block is copied to the ith position in first 64-bit word. Second bit of ith plaint text block is copied to the ith position in second 64-bit word and so on. Now all the encryption operations are performed on these 64-bit words which result in parallel encryption of 64 input blocks.

The subCells operation is often performed by a precomputed lookup table, but in bit-slice implementation it is computed by basic operations. Listing 2.2 shows inplace bitslice implementation of LED sbox operating on 4 input bits. Similarly, mixColumns operation is per-

formed using basic operations. The shiftRows operation is in fact relabeling tbe 64 words. In order to perform addConstants and addKey operations, bits of tbe round constants and round key are to be packed in bitslice fashion. The bitslice implementation requires more memory resources as 64 input blocks are treated simultaneously, but it has a major performance gain over otber implementation techniques.

```
b2   XOR(b2,bl);   b3 - XOR(b3,bl);   t - b2;   b2 - AND(b2,b3);
bl   XOR(bl,b2);   t   XOR(t,bO);   b2 - bl;   bl - AND(bl,t);
bl   XOR(bl,b3);   t   XOR(t,bO);   t - Ob(t,b2);   b2 - XOR(b2,b0);
b2   XOR(b2,bl);   t   XOR(t,b3);   b2 = !b2;   bO   XOR(bO,t);
b3   b2;   b2 - AND(b2,bl);   b2 = XOR(b2,t);   b2   !b2;
```

Listing 2.2: Bitslice lnlplementation of Sbox.

2.4 Hardware Implementation

Hardware implementations are deployed in dedicated hardware devices such as FPGAs and ASICs. The three ways to implement a block cipher in hardware are serial, round based and unfolded implementation. In all of tbese, tbe encryption operation is written as combinational logic operating on input bits. In round based implementation, one round of tbe block cipher is implemented and state signals are passed through it for n number of times in n clock cycles. Contrary to software implementations where byte is tbe smallest unit of storage, hardware implementations have access to individual bits as signal wires. So it is often tbe case tbat while implementing for ASICs, complete round function is expressed by basic gates(AND, OR, XOR etc). However, in FPGAs, precomputed arrays are also stored in LUTs to shorten tbe lengtb of critical patb. For round based implementation of LED block cipher, tbe 64 bit input and key is stored in a state register. The subCells and mixColumns operation is performed using basic logic gates. Sixteen parallel instance of subCells are implemented and complete state is updated in parallel. Sinlilarly, 4 instance of mixColumns operation are implemented, each operating on sixteen state signals corresponding to one column of tbe state matrix. The shiftRows operation is simply rewiring of tbe state signals. The entire state is updated by round transformation in one clock cycle. The control logic is implemented to keep track of number of rounds and after 32 cycles, tbe state holds tbe ciphertext. Figure 2.4 shows an overview of round based hardware implementation.

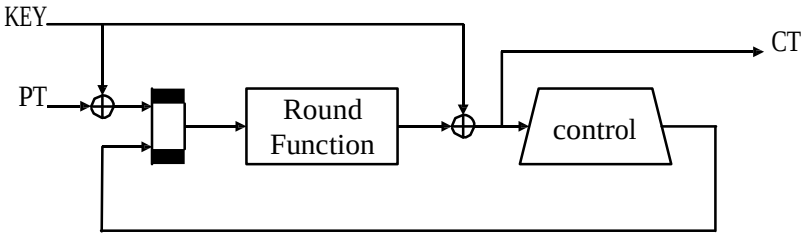


Figure 2.4: Round based Implementation of a Block Cipher in Hardware.

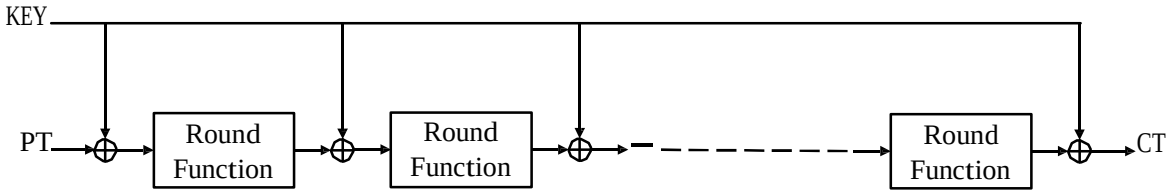


Figure 2.5: Unfolded Implementation of a Block Cipher in Hardware.

The unfolded implementation involves implementing complete rounds of the block cipher one after another. In each clock cycle, current round circuit processes the state signal and passes it to the next round's circuit. This increases the resource consumption but now new input block can be fed to the circuit in every next clock cycle. This first plaintext block is converted to ciphertext after 32 cycles, but after that, each subsequent cycle produces the ciphertext of next block. This increases the throughput manifolds at the expense of extra resources. Figure 2.5 shows undfolded implementation of a block cipher.

In serial implementation, one round of the block cipher is further divided into smaller pieces which operates on some portion of the state signal in one clock cycle. This is often done to save implementation resources at the expense of extra clock cycles. While doing serial implementation of block cipher, a major choice decision has to be made about width of the data path which will be processed in each cycle. For serial implementation of LED block cipher with 4-bit data path, only one instance of the subCells operation is implemented. Four bits of a nibble are processed in one cycle which updates the complete state by subCells operation in sixteen cycles. Similarly, mixColumns operation is implemented using serial matrix A which updates one column of the state in 4 clock cycles. The complete encryption operation is performed in 1248 cycles with an area requirement of 966 GE [15].

2.5 **Decryption**

By use of CTR and OFB block cipher modes of operation, it is possible to perform decryption of a ciphertext without actually implementing the decryption routine of the underlying block cipher. However, with block cipher mode of operation such as CBC, it becomes necessary to implement the decryption routine. The decryption of an SPN cipher is similar to encryption except inverse transform of each component is applied in reverse order. Since all the operations of LED block cipher are non-involutive, their inverse has to be implemented separately. This requires additional resources which are usually equal to the resources required to implement encryption process. Thus support for decryption in CBC like mode of operations doubles the implementation cost for such ciphers. All the implementation techniques explained in previous sections are also applicable to the decryption process. However, while implementing the decryption by lookup table based implementations, a major question arises: "How to combine the inverse of substitution and permutation layer in a single lookup table?". In the forward direction(encryption), we computed the lookup tables by combining the effect of subCells and mixColurnns operations of one round, but this is not possible for the reverse direction (decryption). So two possible alternatives are

- Decryption-1: Implement the inverse of subCells and mixColurnns operations in separate lookup tables. This almost doubles the number of lookup operations required to implement each round as compared to encryption routine and results in lower throughput.

- Decryption-2: Combine the inverse subCells of *round;* with inverse mixColumns operation of *round;*$_{-1}$. This way both the operations can be performed in single lookup as was done in encryption. However, now the values of interleaved operations such as the inverse of addConstants and addKey need to be recomputed and then added to the state. Moreover, mixColumns operation from the last round and subCells operation from the first round are still to be computed by separate non-combined lookup tables. So the two sets of lookup tables are to be stored. This almost doubles the memory requirement but results in higher throughput as compared to the Decryption -1 method.

Chapter3

LIGHTWEIGHT MDS MATRICES OVER $GF(2^4)$

3.1 Introduction

The main objective of lightweight cryptography is to design cryptographic primitives with lesser implementation cost in terms of chip area and energy consumption [50]. For the block ciphers, this is achieved by constructing lightweight confusion and diffusion layers with strong cryptographic properties [51]. Out of these two, the later is employed to spread internal dependencies of the plaintext as much as possible [52]. In wide trail SPN designs, MDS [3, 15] and almost MDS matrices [53, 32] are preferred to construct a secure cipher as compared to other choices for the diffusion layer [54]. Although MDS matrices have higher implementation cost than almost MDS matrices, they have optimal branch number and exhibit fast diffusion. This reduces the number of rounds required to achieve desired level of security as compared to other diffusion mechanisms [55]. In this chapter, we discuss different MDS matrix constructions over $GF(2^4)$ and provide methods for multiplication with MDS matrix M and its inverse (M^{-1}) for minimal overhead.

3.2 Finite Field

The block ciphers often perform arithmetics in some finite field. A finite field with 2n elements and irreducible polynomial $p(X)$ of degree n is denoted by $GF(2n)/p(X)$. Two finite fields over different irreducible polynomials of same degree n are isomorphic [56]. The number of irreducible polynomials Mn of degree n over $GF(2)$ is given by

$$Mn(2) = \frac{1}{n} \sum_{d|n} \mu(d) 2^{n/d}$$

where $\mu(d)$ is Mobius function [57].

The elements of $GF(2n)$ can be written in two ways: 1) in polynomial representation as

$$\sum_{i=0}^{n-1} b_i X^i$$

and 2) in bitwise representation as

$$b_{n-1}b_{n-2}\ldots b_2b_1b_0$$

where b_i E GF(2). This way, a 4-bit string 1001 i.e Ox09 in hexadecimal, corresponds to $X^3 + 1$ in polynomial representation. Addition of two elements in $GF(2n)$ is performed by bitwise xor of coefficients of the polynomial representation of the elements. The multiplication of two elements is equal to product of polynomial representation of both elements modulo irreducible polynomial $p(X)$. In this chapter, we use the finite field $GF(2^4)$with irreducible polynomial $p(X) = X^4 + X + 1$. For simplicity, we often write this irreducible polynomial $p(X)$ in hexadecimal notation as Ox13.

3.3 Xor Count

Block ciphers often employ MDS matrices defined over a finite field in the diffusion layer. These matrices are implemented as fully unrolled circnits in round based hardware implementations. Thus in order to reduce the implementation cost of the diffusion layer, an effort is made to keep the number of gates required for the implementation as minimum as possible. It was a common belief that multiplication with a low hanuning weight finite field element has low hardware implementation cost [58] . Thus most of the block ciphers used matrices with simple finite field elements e.g MDS matrix of AES for encryption routine consists of only OxOl, Ox02 and Ox03 [59]. This was due to the case that field multiplication of an arbitrary element $f3$ with Ox02 is simply left rotation by one bit position modulo irreducible polynomial (see Appendix A) and multiplication with other elements can be derived from it [40]. For example, field multiplication of an arbitrary element $f3$ with Ox03 in $GF(2^4)/0x13$ is computed as $3 \times f3 = (2 \times fJ) \oplus \{3$. Let binary representation of Ox03 and $f3$ be (0, 0, 1, 1) and (b3, b2, $b1o$ bo), then

$$\begin{aligned}(0,0,1,1)(b3, b1, bo) &= (b1.bo \text{ ffi } b3, b3)\text{ffi} \\ &\quad (b3, b2, b1, bo) \\ &= (b3 \text{ ffi } b2, \text{ffi } blo \\ &\quad b3 \text{ ffi } b1 \text{ ffi } bo, b3 \text{ ffi } bo)\end{aligned}$$

We reference this type of field multiplication implementation as direct multiplication (DM).

The implementation cost of the complete matrix multiplication is then determined by two costs 1) field multiplication cost of individual elements of the matrix, 2) summation cost: the number of xor operations required to add these multiplication results. Thus the overall xor count of the matrix was reduced by using field elements with low hamming weight as summation cost was considered inevitable [50]. Moreover, the number of unique field elements in the matrix were kept to minimum by reusing these in each row of the matrix so that field multiplication results could also be reused. Further improvement was made by algorithms presented in [60] which reduced the implementation cost by iteratively finding and eliminating the common sub-expressions. In 2014, A new method to count the number of xor gates required to implement the field multiplication was proposed in [61]. It calculates the field multiplication cost by counting the number of 1's in each row of the multiplication matrix minus the number of rows. This is referred as d-xor in literature. The field multiplication of an arbitrary element with Ox03 as shown in the above example can be represented by a matrix multiplication over $GF(2)$ as

$$\begin{vmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{vmatrix} \quad \begin{vmatrix} ba \text{ Efl } b2 \\ \text{Ef)} \quad b! \\ ba \text{ Ef) } b1 \text{ Efl } bo \\ ba \text{ Efl } bo \end{vmatrix}$$

Thus d-xor cost of field multiplication with Ox03 is 5 i.e 9-4 = 5. The authors also showed that field multiplication of higher hamming weight elements can also be implemented with lower xor count. Appendix B shows multiplication matrices and bitwise field multiplication for each element of the finite field $GF(2^4)j\ 0x13$. Several researchers then used this new metric d-xor to estimate and report new matrices with lower xor count [52, 62, 63, 64]. But the d-xor method provided an overestimation of the xor count and did not take into account the reuse of intermediate results. In [65], the authors proposed a better metric (s-xor) to calculate the field multiplication cost. The s-xor is the minimum number of xor operations required to implement matrix multiplication, where the minimum is taken over all possible implementation sequences. For the above matrix multiplication of d-xor, it is clear that term b_3 Ef) b_0 appears in the third and fourth row. Thus, the term b_3 Ef) b_0 can be computed once

and reused again. In hardware, these intermediate results can be easily reused without any additional memory elements as these are just labels of wires between the gates [50]. Appendix C shows the comparison between number of xor operations required for each type of field multiplication implementation. The above mentioned work mostly focused on reducing the implementation cost by local optimization of the field multiplication by individual elements. These techniques can be extended to globally optimize the matrix multiplication by converting matrix over GF(2n) to a binary matrix over GF(2). But these become impractical for even smaller size MDS matrices as size of the matrix over $GF(2)$ increases rapidly e.g a 4 x 4 matrix over GF(2^4) is a 16 x 16 matrix over GF(2). Moreover, It was shown that implementing the binary matrix multiplication with least possible xor count also known as finding Shortest Linear straight-line Program (SLP) over $GF(2)$ [66, 67] is an NP-hard problem [68, 66]. So researchers mainly focused on reducing the matrix implementation cost by optimizing field multiplication of individual elements and then reusing the intermediate results. However, an efficient solution to similar problem was already known from a different line of research i.e combinational logic minimization. In [69], the authors presented a new heuristic to efficiently implement binary linear layers with reduced circuit. The idea was to keep a setS of bases which contains all known/ computed signals (binary values). At start, the setS contains only input signals. Then a distance vector D is calculated to find how many additional xor operations are required to compute the output signals from the bases of the set S. A new base is computed by adding two of the existing bases and if this new base reduces distance to the output, then it is added to the set S. Thus, the distance to output expressions is iteratively reduced by computing and adding new bases. Tie between existing and new distance vectors is resolved by Euclidean norm. Appendix D shows complete run of the heuristic for the above mentioned example of field multiplication by Ox03.

We refer this heuristic as SLP heuristic. It was later improved in [70] to find an optimal implementation for the dense matrix i.e where the number of 1's is more than 50%. This was done by finding an intermediate value which contained most variables and then running the original algorithm. Authors in [71] modified the SLP heuristic to find the optimal implementation circuit of cryptographic linear layers for a given depth. This reduced the length of critical path for latency conscious applications at the expense of additional xor operations. Moreover, running time increased as the heuristic needed to traverse more paths to find the

optimal solution. In [50], authors reduced the implementation cost of already known matrices by using SLP. The authors found implementation of MDS matrix used in AES with 97 xor operations, while previously known best implementation used 103 xor operations [72]. In this chapter, we use SLP heuristic to find 4 x 4 lightweight MDS matrices over $GF(2^4)$ using different MDS matrix constructions. Moreover we define two new methods to implement matrix multiplication with inverse of an MDS matrix for minimal overhead.

3.4 MDS Matrices

Definition 1. *[52] The branch number of matrix M having order k over a finite field GF(2n) is basically the minimum number of non-zero components of the input vector v and output vector v′ = M · v ranged over all non-zero v* E [GF (2n)]k.

$$BM = min\{W(v) + W(v')\} \quad where \quad v \neq 0$$

Here W is count of non-zero elements in the input and output vector v and v' respectively.

Definition 2. [73] *A Maximum Distance Separable (MDS) matrix with order k is a matrix which attains an optimal branch number of k* + 1.

Use of an MDS matrix with high branch number ensures that a small difference in the input will propagate a large difference in the output. MDS matrices also have following characteristics.

- *A square matrix M is an MDS matrix if and only if all square sub-matrices of M are non-singular [74].*
- *The inverse of an MDS matrix is also an MDS matrix [75]*
- *Transpose of an MDS matrix M is also an MDS matrix [40].*
- *The branch numbers of an MDS matrix M and its inverse* M^{-1} *are same [58].*
- *If a matrix* M^{-1} *is obtained after performing permutations on row or columns of an MDS matrix M, then* M^{-1} *is also an MDS matrix [52].*
- *If M is an MDS matrix over* GF_{2^n}*, then c · M is also MDS matrix for any non-zero c* E *GF2n [75].*

Definition 3. *A self inverse matrix is called involutory matrix i.e its second power is an identity matrix*

$$A = A^{-1} \quad or \quad A^2 = I$$

Involutory MDS matrices are of great interest as the same circuit can be used to implement the inverse of the matrix. However, these involutory matrices have large number of fixed points $(f(x) = x)$ which can be used to distinguish it from random permutation [36]. An involutory permutation with $2n$ inputs has $2^{11} + 1$ fixed points where as a randomly chosen permutation over same space has only one fixed point[76]. If a primitive employs components with a large number of fixed points, then special care must be taken to thwart against cryptanalysis attacks like the Invariant Subspace Attack [77, 78, 37].

3.4.1 Circulant Matrix

A 4×4 right circulant matrix M is denoted by elements of its first row as $Cir(a_0, a1, a2, a3)$ and each subsequent row of the matrix is determined by right rotation of the previous row.

$$Cir(a_0, a_1, a_2, a_3) = \begin{vmatrix} a_0 & a1 & a2 & aa \\ aa & ao & a1 & a2 \\ a2 & aa & ao & a1 \\ a1 & a2 & aa & ao \end{vmatrix}$$

The right circulant matrices have been used in block ciphers AES [3], Piccolo [30] and KLEIN [14]. A more generalized form of circulant matrix is cyclic matrix where each subsequent row is some permutation of the first row. Circulant MDS matrices generally have lower hardware implementation cost as compared to other types of matrices because of less number of distinct elements [52]. The probability of finding a circulant MDS matrix is much higher than finding a random square MDS matrix [79]. But the use of circulant matrices comes with a problem i.e non-existance of Involutory Circulant MDS (ICMDS) matrices. Authors in [80] showed that 4×4 ICMDS matrices do not exist. Later on, it was proved that ICMDS matrices of any order do not exist [75]. This phenomenon increases the implementation cost if both the matrix and its inverse are to be used in a cryptographic primitive. Here we introduce a new class of cyclic matrix which supports inverse matrix multiplication with less overhead.

Definition. *A cyclic matrix Miscalled Circulant Permutation Inverse (CPI) matrix if rows of its inverse matrix* M^{-1} *are some permutations of the first row of the matrix* M.

$$CPI(ao,al,a2,aa)^{-1} = \begin{vmatrix} P.(ao & a1 & a2 & aa) \\ P,(ao, & a1, & a2, & aa) \\ P_9(ao, & a1, & a2, & aa) \\ Ph(ao, & a1, & a2, & aa) \end{vmatrix}$$

Here $P;$ is ith permutation from the list of all available permuations of a vector with 4 elements. The two main benefits of CPI matrices are

- The number of fixed points for a CPI matrix are far lesser than those of involutory matrix. It is conjectured that a CPI matrix has 2 fixed points.
- The inverse of a CPI matrix can be implemented with fewer resources as compared to inverse of a non-involutory cyclic matrix as both consists of same distinct elements.

3.4.2 Recursive Matrix

A 4 x 4 serial matrix M is denoted by elements of its last row and is of the form

$$Ser(ao, a1, a2, aa) = \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ ao & a1 & a2 & aa \end{vmatrix}$$

A recursive MDS matrix M is a matrix which can be derived from ith power of a serial matrix for some positive integer i. This recursive MDS matrix is denoted as $Ser(a_0, ab\ a_2, a_3)i.$ The main characteristic of recursive MDS matrices is that they can be implemented in serial fashion with lesser implementation cost but more clock cycles. The use of serial matrices in cryptographic primitives was first proposed in PHOTON hash function [81] and later used in LED block cipher [82]. The inverse of recursive MDS matrix can be computed by raising inverse of the underlying serial matrix to the power i. Thus, recursive MDS matrix and its inverse can be implemented in serial fashion. The inverse of serial matrix $Ser\ (a_0, a_1, a_2, a_3)$ is of the form

$$Ser(ao, a1, a2, aa)^{-1} = \begin{vmatrix} a, & a, & as & 1 \\ ao & ao & a0 & ao \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{vmatrix}$$

3.4.3 Hadamard Matrix

Given a set of four elements (a_0, a_1, a_2, a_3), a 4 x 4 hadamard matrix H is constructed as follows

$$Had(a_0, a_1, a_2, a_3) = \begin{vmatrix} ao & a1 & a2 & aa \\ a1 & ao & aa & a2 \\ a2 & a3 & ao & a1 \\ a_3 & a_2 & a_1 & a_0 \end{vmatrix}$$

Hadamard matrices are bisymmetric $(H = HT)$ and often used in constructing involutory MDS matrices. If a k x k hadamard matrix H is multiplied by itself, then $H^2 = c? \cdot I$ where $c = EB:;:- a;$. This implies that a hadamard matrix H is involutory if summation of its first row elements result in 1 [83]. Moreover, a non-involutory hadamard matrix can be converted to involutory matrix by dividing it with the sum of elements of the first row [62].

3.5 Implementation of the MDS Matrix Multiplication

Consider a right circulant matrix $Cir(a_0, a_1, a_2, a_3)$ is to be multiplied with input vector (w, x, y, z). Then the output of this multiplication is computed as

$$\begin{vmatrix} ao & a1 & a2 & aa \\ aa & ao & a1 & a2 \\ a_2 & a_3 & a_0 & a_1 \\ a_1 & a_2 & a_3 & a_0 \end{vmatrix} \begin{vmatrix} w \\ x \\ y \\ z \end{vmatrix} = \begin{vmatrix} aow + a1x + a2y + aaz \\ aaw + aox + a1y + a2z \\ a2w + aax + aoy + a1z \\ a1w + a2x + agy + aoz \end{vmatrix}$$

In round based implementation, the complete multiplication is implemented in hardware and all four outputs are computed in one clock cycle. However, in serial based implementation, only the first row of the circulant matrix is implemented and one output is computed in one

clock cycle. For example, the first output of the multiplication is computed as

$$\begin{vmatrix} a_0 & a_1 & a_2 & a_3 \end{vmatrix} \begin{vmatrix} w \\ x \\ y \\ z \end{vmatrix} = \begin{vmatrix} a_0w + a_1x + a_2y + a_3z \end{vmatrix}$$

Now, in order to compute the second output, the input vector is left rotated by one position (x, y, z, w) and multiplied with first row of the matrix in next clock cycle.

$$\begin{vmatrix} ao & a1 & a2 & aa \end{vmatrix} \begin{vmatrix} x \\ y \\ z \\ w \end{vmatrix} = \begin{vmatrix} aox + a1y + a2z + aaw \end{vmatrix}$$

Similarly, the process is repeated two more times to compute the remaining outputs [52]. Thus, the complete matrix multiplication is performed in 4 clock cycles by just implementing the first row of the circulant matrix. However, partial results from the multiplication in each clock cycle are required to be stored separately from the original input vector. Because orignal input vector is to be repeatedly used as input till the time all 4 outputs have not been computed. This approach is also extendable to cyclic, CPI and hadamard matrices. Since all the rows of a these matrices are some permutation of the first row, the inverse permutation of ith row is applied on the input vector before multiplying it with first row of the cyclic matrix. In fact more trade-offs in terms of circuit area and clock cycle are also possible. For example, multiplication for two rows of the circulant matrix can be implemented. Then it is possible to perform the complete matrix multiplication in 2 clock cycles. Without taking into account the cost of memory and control logic, serial implementations of these matrices need lesser area but have high latency. While on the other hand, round based implementations require larger resources but compute output in one clock cycle. Table 3.1 shows serial and round based implementation cost of lightweight 4 x 4 MDS matrices. The recursive MDS matrix used in LED block cipher requires 51 xor operations where as lightest MDS matrix shown at serial3 in Table 3.1 can be implemented by just 44 xor operations.

Table 3.1: Serial and Round based Implementation Cost of Involutory and non-Involutory MDS Matrices and its Inverse.

#	Type	MDSMatrixM	CostA	CostM	CostA-^{1}	CostM-^{1}
1	*Cir(9,* 2, 1, 8)	9218 8921 1892 2189	16	40	21	53
2	*Cyc(D,* 2, 3, 9)	93D2 2D39 D293 392D	18	38	18	40
3	*Ser(l,* 9, 8, 9)	1989 9CD5 5226 6671	16	44	16	44
4	*Had(6,1,4,9)*	6149 1694 4961 9416	18	38	18	43
5	*CPI(1,4,9,D)*	149D 41D9 D941 9D14	16	*50*		
6	*Had;(A,2,5,C)*	A25C 2AC5 5CA2 C52A	20	46		

3.6 Support for Inverse Matrix Multiplication

In this section, we propose two methods for implementing matrix multiplication with the inverse of an MDS matrix for minimal overhead. We explain it with the help of an example. Let M be an MDS matrix such that

$$M = \begin{vmatrix} 2 & C & 9 & D \\ D & 2 & C & 9 \\ 9 & D & 2 & C \\ C & 9 & D & 2 \end{vmatrix} \quad \& \quad M^{-1} = \begin{vmatrix} F & 2 & C & D \\ D & F & 2 & C \\ C & D & F & 2 \\ 2 & C & D & F \end{vmatrix}$$

Then implementation cost for matrix multiplication in $GF(2^4)$ with MDS matrix M and M^{-1} is 45 and *50* xor operations respectively. It implies that use of matrix M in a cryptographic primitive will need additional *50* xor operations to support inverse matrix multiplication in decryption routine. Thus total cost to support matrix multiplication with both M and M^{-1} is 45 + 50= *95* xor operations. Following two methods reduce this overhead.

3.6.1 Mixed Implementation (MI)

- Create matrix *Mm;* by combining MDS matrix M and its inverse M^{-1} as

$$Mm; = \begin{bmatrix} M \\ M^{-1} \end{bmatrix}$$

- Create multiplication matrix Map_2 over GF(2) by replacing each element of the *Mm;* by its multiplication matrix over GF(2). Appendix E shows *Mm;* and MaF_2 for matrix M and M^{-1}.

- If there exists duplicate target signals in *MaF2.* keep the first instance and remove remaining while keeping a record of all signals that are equal.

- Apply SLP heuristic to compute the remaining target signals.

By MI method, total cost for multiplication with M and M^{-1} reduces to 57 xor operations which is just 12 more xor operations than implementation cost of the MDS matrix M where as separate implementation for M^{-1} requires *50* xor operations. The output of the SLP heuristic report implementation for 32 target signals. The first sixteen targets of the output

correspond to multiplication of input with matrix M while remaining targets provide the matrix multiplication result for M^{-1}. In this implementation, a large number of intermediate signals contribute in computation of the target signals for matrix multiplication with M and M^{-1}. Moreover, the target signals corresponding to M^{-1} are often used in computing the same for multiplication with matrix M. The increased used of intermediate signals helps in realizing the whole multiplication with lesser number of xor operations.

However, by MI method, it is hard to realize a separate implementation for M with lesser operations. So contrary to separate implementation of both M and M-1, cost for matrix multiplication with M during encryption is almost equal to the total cost for M and M^{-1}. To deal with this issue, we propose a second method called Derived Implementation (DI) which does not increase the cost of matrix multiplication with M.

3.6.2 Derived Implementation (DI)

- Make multiplication matrix Map_2 over GF(2) by replacing each element of matrix M by its multiplication matrix over GF(2).
- Apply SLP heuristic to compute the target signals for matrix M.
- Make multiplication matrix $M0\}_2$ over GF(2) by replacing each element of matrix M^{-1} by its multiplication matrix over GF(2).
- If a target signal from $M[]\}_2$ is equal to some target signal of *MaF2*. remove it from $M0\}_2$ and keep a record of it.
- Apply SLP heuristic with bases already computed for Map_2 to compute remaining target signals for $M0\}_2$.

The DI method reports 45 xor operations for matrix M and 69 xor operations to support multiplication with M and M^{-1}. Appendix F provides implementation details for Matrix Multiplication with M and M^{-1} by MI and DI methods.

To measure the performance of these methods, we randomly selected 100 matrices of each type and computed implementation cost by MI and DI methods. Table 3.2 summarizes the results of this experiment. Both the methods reported lesser implementation costs than the separate implementation of M and M^{-1} for all matrix types. For Hadamard matrices, the

Table 3.2: Comparison of MI and DI methods to support matrix multiplication with inverse of an MDS matrix with minimal overhead.

TYPe	CostM	CostM-^{1}	Total	MI	DI
Hadamard	52.3	53.2	105.5	64.6	76.7
Circulant	53.2	53.6	106.8	79.2	86.8
Serial	52.5	52.7	105.2	84.7	90.7

overhead to support inverse matrix multiplication by MI method is 13 xor operations on average. On the other hand, the serial matrices require more number of xor operations to support inverse matrix multiplication.

Application to LED Block Cipher. The recursive MDS matrix employed in LED block cipher have implementation cost of 51 and 49 xor operations for multiplication with matrix M and M^{-1} respectively. Thus, in order to support multiplication with both M and M^{-1}, the separate implementation cost reaches a sum of 100 xor operations. By MI and DI methods, this cost reduces to 79 and 89 xor operations respectively. However, the following recursive MDS matrix $Mser$ has implementation cost of 66 and 74 xor operations by MI and DI methods respectively. Use of the matrix $Mser$ in mixColumns operation of LED block cipher will save 13 xor operations for each instance of the mixColumns operation.

$$Mser = \begin{bmatrix} F & D & C & 3 \\ 2 & B & A & 9 \\ E & D & D & 2 \\ D & 7 & 6 & B \end{bmatrix}$$

Chapter4

LIGHTWEIGHT MDS MATRICES OVER $GF(2^8)$

4.1 Introduction

The security of loT devices created demand for lightweight cryptography. This was met by either constructing lightweight primitives or reducing the implementation cost of existing standards. In this chapter we deal with construction of lightweight MDS matrices over $GF(2^8)$ which are often employed in byte oriented block ciphers. As a case study we investigate MixColumns operation of AES [3] block cipher and find corresponding lightweight design choices that support inverse matrix multiplication with minimal overhead. Moreover we define a matrix construction that incorporates inverse shift rows with MixColunms operation.

4.2 AES Block Cipher

In 1997, NIST announced an initiative to develop new block cipher encryption standard to replace DES. The block cipher Rijndael designed by two Belgian cryptographers Rijmen and Daemen was selected as Advanced Encryption Standard (AES) in 2000. It is a 128-bit block cipher with key lengths of 128, 192 and 256 bits. The 128-bit plaintext is treated as 4 x 4 state matrix of 16 bytes. The block cipher transforms the plaintext into ciphertext by performing multiple executions of a round transformation preceded by an initial key addition. Each round consists offour operations such as SubBytes, ShiftRows, MixColumns, and AddRoundKey. However, the last round omits the MixColumns operation. The number of rounds depend upon the key length and these are 10, 12 and 14 for key length of 128, 192 and 256 respectively. Separate round keys are generated for each round by ronning a key schedule algorithm onto the user supplied master key. The cipher has stood firm against rigorous cryptanalysis efforts of researchers spanning over two decades. The biclique attack achieves results slightly better than exhaustive search, but its still impractical [84]. Similar to cryptanalysis, great efforts were made to code efficient and compact implementations of the AES accross different hardware and software platforms. Table 4.1 summarizes the compact

Table 4.1: Compact Hardware Implementations of the AES Block Cipher. The letter E and D stands for support for Encryption and Decryption respectively.

#	Reference	Type	Area(GE)	Cycles
1	[85]	ED	2060	246
2	[86]	E	2400	226
3	[87]	ED	2605	226
4	[88]	ED	3400	1032
5	[89]	ED	4037	336
6	[90]	ED	5400	54

hardware implementations of the AES block cipher.

4.3 The MixColumns Operation

The MixColumns operation operates on columns of the state matrix. Its design criteria included dimensions, linearity, diffusion and performance on 8-bit processors [40]. The size of the column was set to 4 bytes by keeping in view the performance of loopkup table based implementations on 32-bit architectures. The bytes of a column are considered polynomial over $GF(2^8)$ and multiplied with a fixed polynomial $c(x)$ modulo $x^4 + 1$. The performance criterion also dictated the selection of coefficients for the polynomial $c(x)$. Coefficients with simple values such as 0, 1, 2 and 3 are best suited for this criteria as multiplication with 0 and 1 involve no processing. It is possible to perform multiplication with 2 efficiently by a left shift and a conditional xor. The multiplication with 3 is performed by multiplying the input with 2 and then xor the result with input. The wide trail design of AES imposed the restriction of linearity and high diffusion. Thus simple coefficients were selected in such a way that MixColumns operation have optimal branch number of five. The polynomial for MixColumns operation is

$$c(x) = 3 \cdot x^3 + 1 \cdot x^2 + 1 \cdot x + 2$$

It is possible to illustrate the modular multiplication with polynomial $c(x)$ as matrix multiplication. Fignre 4.1 shows the MixColumns operation of the AES block cipher on the state matrix. The polynomial $c(x)$ is co-prime with modular polynomial $x^4 + 1$, thus its invertible. The inverse of MixColumns operation during decryption is performed by multiplying each

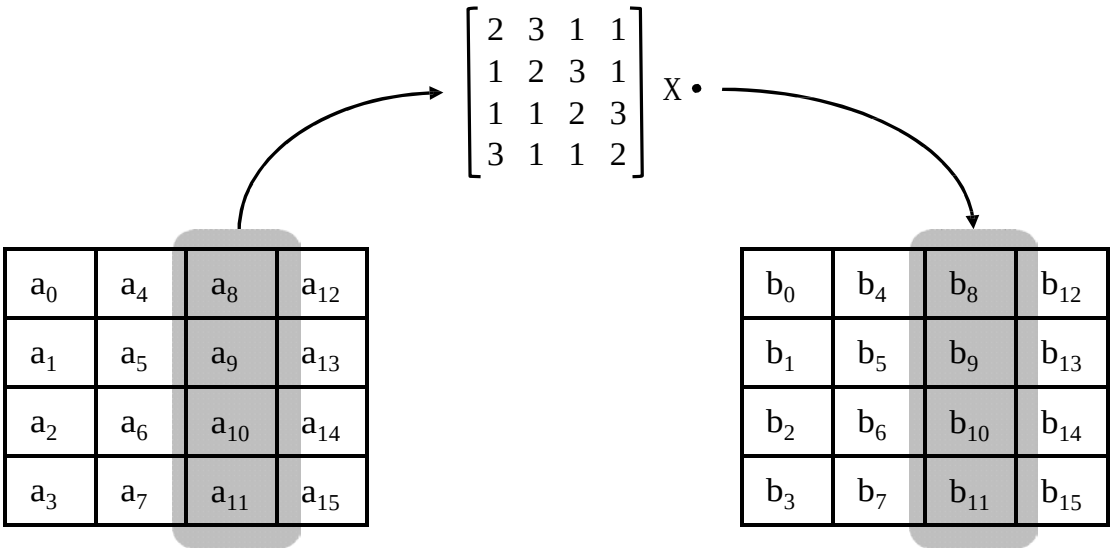


Figure 4.1: MixColumns Operation of the AES Blockcipher.

column of the state matrix by inverse polynomial $d(x)$, defined as

$$d(x) = B \cdot x^3 + D \cdot x^2 + 9 \cdot x + E$$

4.4 Implementation of MixColumns and InvMixColumns Operation

AES employs following MDS matrices M and M^{-1} in MixColumns and InvMixColumns operation during encryption and decryption respectively.

$$M = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \quad \& \quad M^{-1} = \begin{bmatrix} E & B & D & 9 \\ 9 & E & B & D \\ D & 9 & E & B \\ B & D & 9 & E \end{bmatrix}$$

Performance on large range of platforms was a major design criteria for AES competition. However, it was hard to completely fulfill this requirement for both encryption and decryption routine. So the designers focused on optimizing the components involved in encryption routine. This led to selection of polynomial $c(x)$ with simple coefficients and lesser implementation cost. On the other hand, inverse of the polynomial $c(x)$ which is required for decryption consists of coefficients with relatively higher implementation cost. Although block cipher mode of operations such as CTR, OFB and CFB support decryption of the ciphertext without actually implementing the decryption routine of the underlying block cipher, use of

these may not be possible in all scenarios. So implementing the decryption routine becomes inevitable. In case of diffusion layer, the problem boils down to efficient implementation of InvMixColurnns alongwith the MixColurnns operation. Instead of implementing both the matrices M and M^{-1} separately, authors in [90] proposed to decompose the M^{-1} matrix into simple matrices as follows

$$M^{-1} = M + \begin{vmatrix} 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 \end{vmatrix} + \begin{vmatrix} 4 & 0 & 4 & 0 \\ 0 & 4 & 0 & 4 \\ 4 & 0 & 4 & 0 \\ 0 & 4 & 0 & 4 \end{vmatrix}$$

Multiplication of an input operand with element 4 and 8 require 5 and 7 xor gates respectively. These multiplication results are computed once and then reused for each row of the matrix multiplication. By this decomposition, the implementation cost for multiplication with both the matrices reduces to 195 xor gates. However, another efficient implementation by Paulo Barreto is reported in [87]. It works by factorizing the inverse matrix as

$$M^{-1} = M \cdot Mfact = M \cdot \begin{vmatrix} 5 & 0 & 4 & 0 \\ 0 & 5 & 0 & 4 \\ 4 & 0 & 5 & 0 \\ 0 & 4 & 0 & 5 \end{vmatrix}$$

In order to implement the InvMixColurnns operation by factorization method, the input column $A = (ao, a1, a2, ag)$ is first multiplied by the matrix $Mfact$.

$$\begin{vmatrix} bo \\ b1 \\ b2 \\ bg \end{vmatrix} = \begin{vmatrix} 5 & 0 & 4 & 0 \\ 0 & 5 & 0 & 4 \\ 4 & 0 & 5 & 0 \\ 0 & 4 & 0 & 5 \end{vmatrix} \begin{vmatrix} ao \\ a1 \\ a2 \\ ag \end{vmatrix}$$

This multiplication by matrix $Mfact$ is implemented using 58 xor gates and then the original input A or the output $B = (b0, b1, b2, b3)$ is fed to the MixColurnns circuit depending upon the encryption or decryption routine. The complete process is implemented by 155 xor gates

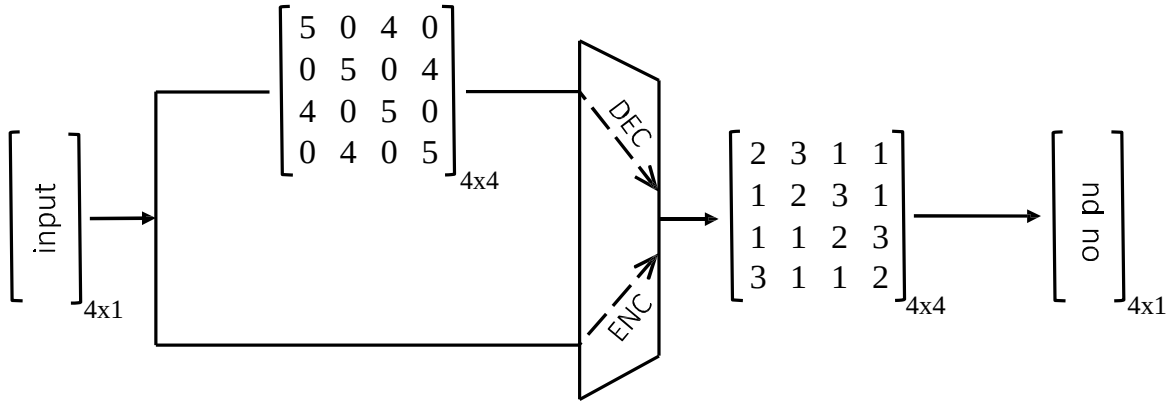


Figure 4.2: Implementation of MixColumns and InvMixColumns operations by Factorization Method.

and a 32-bit multiplexer. Figure 4.2 shows implementation circuit for MixColumns and InvMixColumns operations by factorization method.

In [85], the authors reported that matrix M^{-1} is infact cube of the MDS matrix M.

$$\begin{bmatrix} E & B & D & 9 \\ 9 & E & B & D \\ D & 9 & E & B \\ B & D & 9 & E \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}^3$$

Thus, in order to achieve functionality of InvMixColumns operation, MixColumns operation is performed three times onto the state matrix. It helps in implementing the inverse matrix multiplication without any additional circuitry. However this implementation requires 3 clock cycles to perform InvMixColumns operation which makes it unsuitable for latency conscious applications.

4.5 Subfield MDS Matrix

Definition 1. A subfield $n \times n$ MDS matrix M' with elements $m_{i,j} \in GF(2^{2k})$ is constructed by running two copies of an $n \times n$ MDS matrix Q with elements $q_{i,j} \in GF(2^k)$ such that

$$m_{i,j} = \begin{pmatrix} q_{i,j} & 0 \\ 0 & q_{i,j} \end{pmatrix} \in GF(2^{2k})$$

In fact, the basic idea is to construct an MDS matrix over $GF(2^8)$ by running two copies of the MDS matrix over $GF(2^4)$ in parallel, each operating on one half of the $GF(2^8)$ element

[61]. In order to perform the matrix multiplication $B = M'.A$, each element of the input vector $A = (a_0, a_1, a_2, a_3)$ over GF(2^8) is split in two nibbles consisting of left and right half $a;$ = (afllaf). Then MDS matrix Q over GF(2^4) is multiplied with each half of the input vector and results are concatenated to form the output vector over GF(2^8).

$$A = (ao,a,a2,aa) = (a\&'ila, \text{afllaf}, a\ lla,\ a\ iia:)$$
$$(zo,z,z2,za) = Q \cdot (a\&',af,a,a)$$
$$(z4,z5,z5,z7) = Q \cdot (a\ ,af,a\ ,a:)$$
$$B = (b_0, b_1, b_2, b_3) = (z_0||z_4,\ z_1||z_5,\ z_2||z_6,\ z_3||z_7)$$

It was shown in [91] that such a transform is an MDS matrix over GF(2^8) if Q is an MDS matrix over GF(2^4). Similarly, involutory MDS matrices over higher finite fields can be constructed from involutory MDS matrices over smaller fields [50]. The MDS matrix over GF(2^8) used in MixColumns operation of the AES block cipher requires 97 xor gates. However, this cost can be reduced by subfield MDS matrix construction. Authors in [61] suggested to use circulant matrix $C = \text{circ}(1, 1, 4, 9)$ over GF(2^4) to reduce the implementation cost. The MDS matrix C has an implementation cost of 46 xor operations and two instances of it will save 5 xor gates for each instance of MixColumns operation as compared to use of the original MDS matrix of the AES block cipher. However, the lightest MDS matrix M_{lw} over GF(2^4) as shown below requires 38 xor gates only. The subfield MDS matrix over GF(2^8) constructed from M_{lw} can be implemented with 21 xor gates lesser than the AES MDSmatrix.

$$Mtw = \begin{vmatrix} 9 & 3 & D & 2 \\ 2 & D & 3 & 9 \\ D & 2 & 9 & 3 \\ 3 & 9 & 2 & D \end{vmatrix}$$

The major advantage of subfield construction is that MDS matrices over smaller finite fields have lesser implementation cost. Moreover, it is possible to serialize the matrix multiplication by implementing only one instance of the MDS matrix over GF(2^4) and then performing multiplication by left half of the input vector in first clock cycle followed by the

right half in next. This way the implementation cost can be reduced to half at the expanse of an additional clock cycle. Similarly, subfield construction can be extended to make diffusion matrices over GF(2P) with branch number B by running pfq copies of the matrix over smaller finite field $GF(2q)$ with same branch number where p divides q [61].

4.6 Support for InvMixColumns Operation with minimal overhead

In section 4.5, we explained how subfield construction can be used to make MDS matrices over higher finite fields from MDS matrices over smaller finite fields. We used subfield construction to reduce the implementation cost of MDS matrix multiplication for the forward direction(encryption). In fact, the subfield construction can be easily extended to construct MDS matrices which support implementation of its inverse matrix multiplication with minimal overhead.

Definition 2. Given an MDS matrix Q and its inverse Q' with elements $q;J, q:,i$ E $GF(2k)$, the inverse-subfield(iSubfield) MDS matrix P is constructed with elements PiJ E $GF(2^{2}k)$ such that

$$PiJ = \begin{cases} qoi,j \\ \end{cases} ,;, \quad \text{E } ()1'(2'')$$

Contrary to subfield construction, the iSubfield MDS construction uses one instance of each i.e Q and its inverse Q' instead of using two instances of the MDS matrix Q. This helps in implementing inverse matrix multiplication with minimal overhead. Each element of the input vector $A = (a_{0,}a_{1,}a_{2,}a_{3})$ over $GF(2^{8})$ is split in two nibbles consisting of left and right half $a; = (af \,||af)$. Then MDS matrix Q and its inverse Q' is multiplied with left and right half of the input vector respectively and results are concatenated to form byte output vectors over $GF(2^{8})$.

$$A = (a_0, a_1, a_2, a_3) = (a_0^L||a_0^R,\ a_1^L||a_1^R,\ a_2^L||a_2^R,\ a_3^L||a_3^R)$$

$$(zo, Z1, z2, z3) = Q \cdot (a, af, af, af)$$

$$(z4, zs, Z6, zr) = Q' \cdot (a, a, a, af)$$

$$B = (b_0, b_1, b_2, b_3) = (z_0||z_4,\ z_1||z_5,\ z_2||z_6,\ z_3||z_7)$$

In order to perform inverse transformation, a nibble swap operation is performed before and after the multiplication with iSubfield MDS matrix P. This nibble swap operation changes

the positions of left and right halves of the input vector. Thus the left half values which were previously multiplied with MDS matrix Q, now gets multiplied with Q' and vice-a-verse. Since Q and Q' are inverse of each other i.e $Q \cdot Q' = 1$, swapping the nibbles of input vector and then applying the same transformation results in identity.

$$B = (bo,bb\ ,ba) = \text{Mllb, bfllbf, } b\ llb,\ b\ llbf)$$

$$NibbleSwap\ \ (b_0^R llb_0^L,\ b_1^R llb_1^L,\ b_2^R llb_2^L,\ b_3^R llb_3^L)$$

$$(z_0, z_1, z_2, z_3) = Q \cdot (b_0^R, b_1^R, b_2^R, b_3^R) \qquad (z_4, z_5, z_6, z_7) = Q' \cdot (b_0^L, b_1^L, b_2^L, b_3^L)$$

$$(zo,z1, z2, za) = Q \cdot Q' \cdot (a, af, a, a:) \qquad (z4,zs,z6,z7) = Q' \cdot Q \cdot (a\ ,af,a\ ,a\ )$$

$$(zo,zl,z2,za) = 1 \cdot (a\ ,af,a\ ,a:) \qquad (z4,zs,z6,z7) = 1 \cdot (a\ ,af,a\ ,a\ )$$

$$A' = (ao,al,a2,aa) = (\text{zollz4, z1llzs, z2llz6, zallz7})$$

$$NibbleSwap\ \ (z_4 || z_0,\ z_5 || z_1,\ z_6 || z_2,\ z_7 || z_3)$$

$$= (a_0^L || a_0^R,\ a_1^L || a_1^R,\ a_2^L || a_2^R,\ a_3^L || a_3^R) = (a_0, a_1, a_2, a_3) = A$$

Similar to involutory matrices, the iSubfield MDS matrix makes use of same set of operations for its inverse. However the number of fixed points in iSubfield MDS matrices are far lesser than the involutory matrices. In fact, the number of fixed points in an iSubfield MDS matrix is equal to the number of fixed points in the underlying MDS matrices Q and its inverse Q'. Use of iSubfield MDS matrix in MixColumns operation reduces the implementation cost of InvMixColumns operation. It is possible to implement nibble swap operation with relatively lesser cost in both hardware and software. In hardware, it simply translates to rewiring of the signal wires. In software, it requires 2 shift and one OR operation for each input byte, making it relatively negligible as compared to the cost of matrix multiplication operation. For following choices of MDS matrices Q and Q' over $GF(2^4)$, the MixColumns and InvMixColumns operations can be implemented with 78 xor gates only.

$$Q = \begin{vmatrix} 9 & 3 & D & 2 \\ 2 & D & 3 & 9 \\ D & 2 & 9 & 3 \\ 3 & 9 & 2 & D \end{vmatrix} \quad \& \quad Q' = \begin{vmatrix} D & 1 & F & 8 \\ 8 & F & 1 & D \\ F & 8 & D & 1 \\ 1 & D & 8 & F \end{vmatrix}$$

4.7 Incorporating Inverse ShiftRows (InvShiftRows) with MixColumns

The ShiftRows operation of the AES shifts the bytes in ith row for i number of position to the left where 0 i 3.

$$state;n = \begin{vmatrix} bo & b4 & bs & b12 \\ bl & b5 & bg & b13 \\ b2 & b6 & blO & b14 \\ ba & b7 & bn & b15 \end{vmatrix} \underline{ShiftRmns} \begin{vmatrix} bo & b4 & bs & b12 \\ b5 & bg & b13 & bl \\ blO & b14 & b2 & b6 \\ b15 & ba & b7 & bn \end{vmatrix} = stateuut$$

In order to perform inverse transform of the ShiftRows operation, bytes of the ith row has to be right shifted for i number of positions. Thus implementing the InvShiftRows operation requires additional resources. Authors in [87] made an observation that left shift row transformation applied on *Oth* and *2nd* row are self inverse. Moreover, ShiftRows operation on the 1st and *3rd* row are inverse of each other. Thus swapping the 1st & *3rd* row (SwapRows) of the state matrix and then applying the ShiftRows operation infact brings the InvShiftRows transformation.

$$stateuut \xrightarrow{SwapRows} \begin{vmatrix} bo & b4 & bs & b12 \\ b15 & ba & b7 & bn \\ blO & b14 & b2 & b6 \\ b5 & bg & b13 & bl \end{vmatrix} \underline{ShiftRmns} \begin{vmatrix} bo & b4 & bs & b12 \\ ba & b7 & bn & b15 \\ b2 & b6 & blO & b14 \\ bl & b5 & bg & b13 \end{vmatrix} \underline{SwapRows} \quad state;n$$

Incorporating the InvShiftRows by ShiftRows and SwapRows operation will require use of an MDS matrix which support such row swapping.

Definition 3. A 4 x 4 non-involutory MDS matrix M is called Row Permutation Inverse (RPI) Matrix, **if** it supports inverse transformation using same circuit when SwapRows operation is performed on the output. Let

$$B = M.A = M. \begin{vmatrix} ao \\ a_1 \\ a_2 \\ a_3 \end{vmatrix} = \begin{vmatrix} bo \\ b_1 \\ b_2 \\ b_3 \end{vmatrix}$$

Then

$$B' = B \;\; SwapR.ows \left| \begin{matrix} bo \\ bg \\ \\ bl \end{matrix} \right| \; and M.B' = M. \left| \begin{matrix} bo \\ bg \\ \\ bl \end{matrix} \right| \qquad \left| \begin{matrix} SwapRows \\ \hline \\ \\ \\ \end{matrix} \right| \left| \begin{matrix} ao \\ a1 \\ a2 \\ aa \end{matrix} \right| = A$$

Following RPI matrix *Mrp;* over GF(2^4) requires 49 xor operations. Using the matrix *Mrp;* to construct Subfield matrix over GF(2^8) will help in implementing the InvMixColumns and InvShiftRows operation by using the implementation of MixColumns and ShiftRows operation respectively.

$$Mrpi = \left| \begin{matrix} 2 & 5 & C & A \\ A & C & 5 & 2 \\ C & A & 2 & 5 \\ 5 & 2 & A & C \end{matrix} \right|$$

Example. Let P be a Subfield matrix over GF(2^8) constructed from *Mrp;* over GF(2^4), $A = (15, 26, 37, 48)$ be an input vector over *G*F(2^8) then

$$B = P.\ A = P.\ (15,26,37,48)$$

$$= p.\ (lll5,2ll6,3ll7,4ll8)$$

$$=?\ Mrp;.\ (1, 2, 3, 4) = (1, 6, A, 9)$$

$$and \;\; =?\ Mrpi\ .\ (5, 6, 7, 8) = (A, 1, 0, 7)$$

$$then \quad B = (bo,b1,b2,b3) = (1llA,6ll1,Alll0,9ll7) = (1A,61,A0,97)$$

So for the inverse transformation, apply SwapRows onto the output and then multiply with Subfield matrix P.

$$B' = B \;\; SwapR.ows \;\; (1A,97,A0,61) = (1llA,9ll7,Alll0,6ll1)$$

$$=?\ Mrp;\ .\ (1, 9, A, 6) = (1, 4, 3, 2)$$

$$=?\ Mrp;.\ (A, 7, 0, 1) = (5, 8, 7, 6)$$

$$then \quad (1ll5,4ll8,3ll7,2ll6) \;\; SwapR.ows \;\; = (1ll5,2ll6,3ll7,4ll8) = A$$

ChapterS

dSPN: A TOY SPN CIPHER TO SUPPORT DECRYPTION WITH MINIMAL OVERHEAD

5.1 Introduction

The added advantage SPN networks have over Feistel is that they process complete state in one round. This enables them to attain required level of confusion and diffusion with relatively lesser number of rounds [33]. However, SPN networks need extra resources to support inverse transformation. Few designs have solved the issue using involutive components. But it is often the case that additional logic and resources are requried to protect against attacks posed by large number of fixed points of involutive components [32]. On the other hand, reflection ciphers did it by incorporating inverse transformation in later half of the encryption routine [11, 29]. However, this increases the implementation cost of encryption routine and often results in almost double the cost of encrypt-only designs. In this chapter, we define dSPN which is an SPN structure thats supports decryption with minimal overhead. The cipher employs non-involutive components which acts as self inverse after some linear operation on the output. In fact the idea of iSubfield matrix is extended to both confusion and diffusion layers to reduce the decryption overhead.

5.2 Specification of dSPN

The dSPN is a lightweight block cipher with 128-bit plaintext and key length. The ciphertext is produced after 10 rounds and each round consists of operations similar to AES [3] and LED [15] block cipher i.e addConstants, subCells, shiftRows and mixColumns. Figure 5.1 shows the encryption operation of the dSPN block cipher. The 128-bit plaintext block and user supplied master key is arranged in a 4 x 4 matrix of 16 bytes as

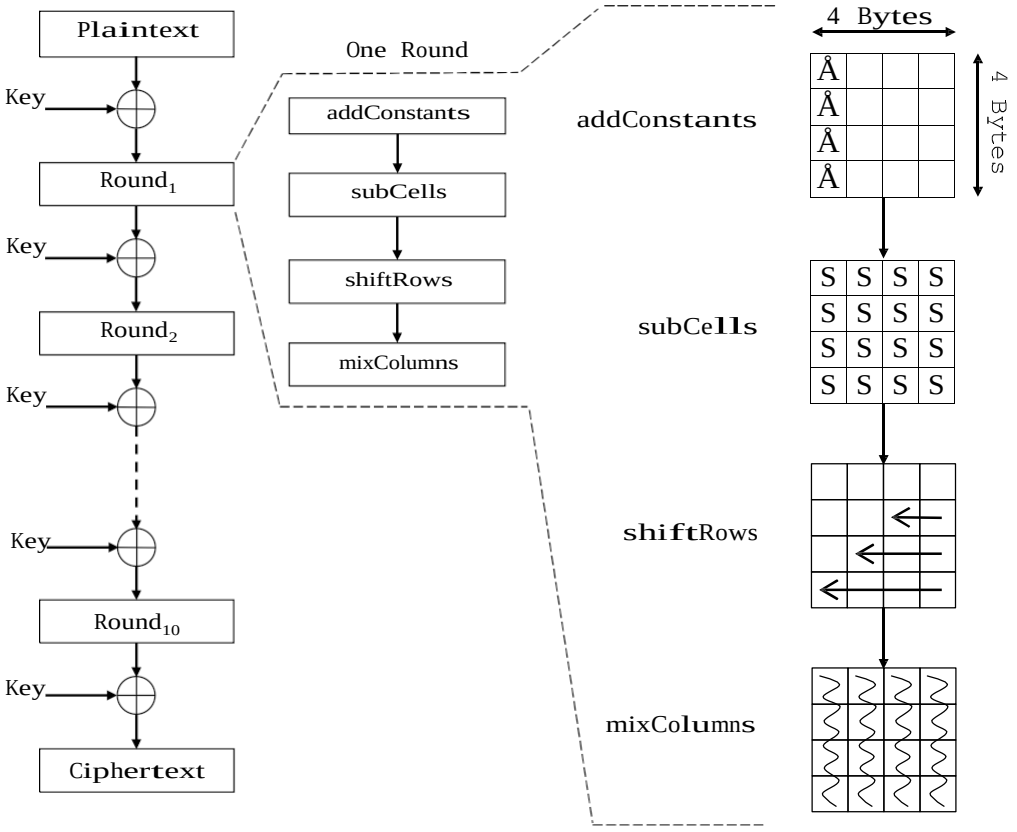


Figure 5.1: Encryption Operation of dSPN Block Cipher.

$$state_{plaintext} = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ b_8 & b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix} \qquad Key = \begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{bmatrix}$$

The key is mixed into state by performing xor operation on corresponding bytes of the state and key matrix.

subCells. Each byte of the state matrix is updated by lookuping values from an 8-bit sBox. Table 5.1 shows values of the 8-bit sBox.

shiftRows. The ith row of the state matrix is left rotated for i number of bytes where $0 \leq i \leq 3$.

mixColumns. The mixColumns operation divides each column of the state matrix into left and right and half. Then left half is multiplied with matrix M and right half is multiplied with M^{-1}. Infact it is an iSubfield MDS matrix over $GF(2^8)$ by constructed by two MDS matrices M and M^{-1} over $GF(2^4)$.

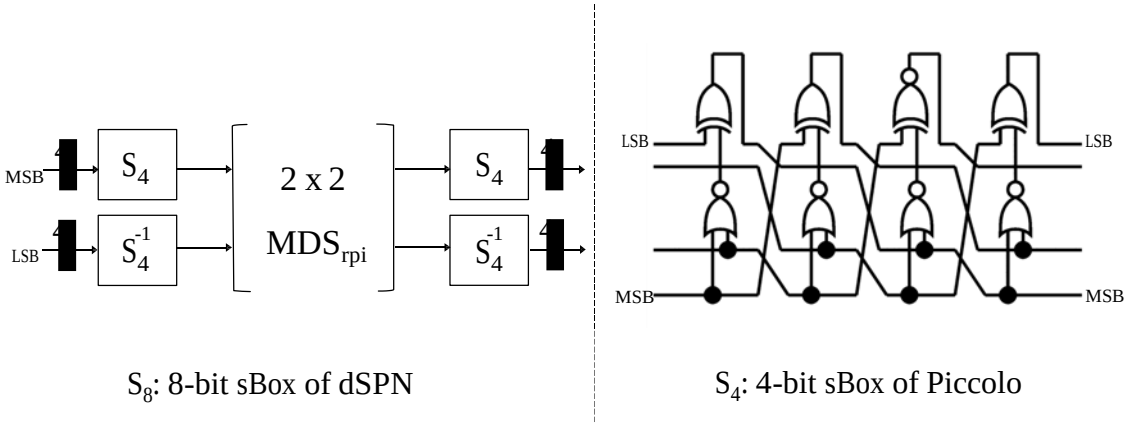
Table 5.1: The 8-bit sBox used in dSPN.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	ossloruamn mAAM@7Dl%															
1	DB	B6	lE	92	E8	AD	45	71	27	FF	3	6A	89	CO	34	SC
2	EC	C7	9D	13	D5	OE	58	SF	66	31	A2	20	74	BA	F9	4B
	3Aln %9MM@ OCIDID ffiMW															
4	16	8B	D9	El	90	54	OA	C2	3C	63	4F	FS	BE	78	2D	A7
5	29	SE	36	FO	61	87	CF	8	DD	95	7A	E3	AB	42	lC	B4
6	FD	4	6C	25	33	CB	82	SA	99	DO	B8	11	47	AF	E6	7E
7	BO	D8	81	79	C6	3F	67	9E	55	OD	F4	4C	12	EB	A3	2A
	8 MO @ o mM 1741															
	9 5 ID OOTIC1 1BWM mm															
A	64	4D	F7	3A	2F	76	Bl	A5	EE	18	80	D2	OC	53	9B	C9
B	m	M	W		n	OC	TI		14	M	G			1	m	W
C	CA	ES	7F	84	B7	Fl	26	1D	48	AE	39	SB	93	DC	2	60
D	97	7C	E4	DF	lA	49	AO	B3	FB	22	51	38	CD	85	6E	6
E	52	21	AS	OB	4E	15	ED	F6	BF	77	9C	C4	30	69	SA	D3
F	OF	F3	4A	57	A4	EO	19	2C	72	BB	D6	8E	65	3D	C8	91

$$M = \begin{vmatrix} 3153 \\ 3 & 2 & F & 9 \\ A & 3 & C & D \\ 1 & 1 & 8 & 2 \end{vmatrix} \quad \& \quad M^{-1} = \begin{vmatrix} 9EF4 \\ 4 & 7 & F & 9 \\ D & 4 & 9 & D \\ E & E & 2 & 7 \end{vmatrix}$$

addConstants The addConstants operation mixes the current round number *(rd#)* in right nibbles of the first column starting from 1 to 10. The matrix representation of addConstants operation is as follows

$$RoundCanstants = \begin{vmatrix} Olrd\# & 00 & 00 & 00 \\ Olrd\# & 00 & 00 & 00 \\ Olrd\# & 00 & 00 & 00 \\ Olrd\# & 00 & 00 & 00 \end{vmatrix}$$



S_8: 8-bit sBox of dSPN

S_4: 4-bit sBox of Piccolo

Figure 5.2: Construction of 8-bit sBox S_8 from piccolo sbox.

5.3 Design Rationale

sBox. The 8-bit sBox (S_8) employed in dSPN is specifically constructed to work in conjunction with iSubfield MDS matrix used in mixColumns operation. In order to keep the implementation cost minimal, it has been constructed from a non-involutive 4-bit sbox (S_4) of the lightweight block cipher Piccolo [30]. Figure 5.2 shows the construction of S_8 and S_4. The eight bit input is divided in two nibbles of 4-bit each. The four Most Significant Bits(MSB) are updated by value from S_4 and the four Least Significant Bits(LSB) are updated from inverse of the S_4. Then these two nibbles are multiplied by a 2×2 Row Permutation Inverse MDS (MDS_{rpi}) matrix. The MDS_{rpi} is a non-involutive matrix that supports inversion if order of the output nibbles is reversed. In the end, the nibbles are again updated by S_4 and its inverse. The specific placement of S_4, its inverse (S_4^{-1}) and use of MDS_{rpi} helps in constructing a non-involutive 8-bit sBox that supports inversion after the NibbleSwap operation. Table 5.2 shows sBox of block cipher piccolo. The MDS_{rpi} matrix is as follows

$$MDS_{rpi} = \begin{bmatrix} 1 & 4 \\ 4 & 2 \end{bmatrix}$$

shiftRows. The shiftRows operation employed in dSPN is directly from AES [3] and it is used for diffusing the change accross different columns. The left rotation of bytes by specific number of positions alongwith the use of a 4×4 MDS matrix with branch number of 5 helps in achieving maximum number of active non-linear components in wide-trail designs.

Table 5.2: sBox of Piccolo block cipher.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$S_4[x]$	e	4	b	2	3	8	0	9	1	a	7	f	6	c	5	d

mixColumns. The mixColumns operation uses an iSubfield matrix over $GF(2^8)$ constructed from a lightweight matrix M and its inverse. This iSubfield matrix supports inverse after the NibbleSwap operation. This helps in reducing the implementation resources required for inverse transformation. Moreover, the MDS matrix M is chosen such that it is lightweight and supports serial implementation from a simple matrix A.

$$A = \begin{vmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 9 & 2 \\ 1 & 0 & 0 & 0 \end{vmatrix} \quad ==> \quad A4 = \begin{vmatrix} 3 & 1 & 5 & 3 \\ 3 & 2 & F & 9 \\ A & 3 & C & D \\ 1 & 1 & 8 & 2 \end{vmatrix} = M$$

The serial matrix A is infact Transpose of a Diagonal-Serial Invertible (DSI) Matrix which were introduced in [58]. The DSI matrices support serial implementation of MDS matrix multiplication by just 10 xor operations over 4 cycles. However the lightest MDS matrix constructed from a DSI matrix requires 43 xor operations for its round based implementation and 46 xor operations for its inverse implementation. So in order to reduce the implementation cost further, I conducted search on Transpose of DSI matrices and found the matrix M which requires 41 and 44 xor operations for forward and reverse multiplication. Thus total cost to construct iSubfield matrix from M and M^{-1} is 85 xor operations. Moreover, implementation of these two matrices M and M^{-1} can be serialized using *DSfi'* matrices A and A^{-1} which require 20 xor operations and 4 clock cycles to perform the mixColurnns operation.

addConstants. The block ciphers often employ a key schedule to generate separate round keys for each round of the block cipher. This removes the self similarity between the round transformations and helps in protection against slide [42] and related key attacks [92]. Since dSPN does not employ a proper key schedule and user supplied master key is used in each round, this makes all the round transformations similar. Thus, in order to remove the self-similarity of round transformations, addConstants operation has been employed. Moreover,

Table 5.3: Number of Active Sboxes in 10 rounds of dSPN.

Round	1	2	3	4	5	6	7	8	9	10
Active-Sbox	4	5	9	25	29	30	34	50	54	55
DiffProb	2^{-17}	2^{-22}	2^{-39}	2^{-110}	2^{-127}	2^{-132}	2^{-149}	2^{-220}	2^{-238}	2^{-242}
Linear Prob	2^{-16}	2^{-20}	2^{-36}	2^{-100}	2^{-116}	2^{-120}	2^{-136}	2^{-200}	2^{-216}	2^{-220}

to keep the implementation cost minimum, the current round number *(rd#)* is used as round constant and it is only xored with right nibble of the first column.

5.4 Security Analysis

Following subsections presents the security strength of the dSPN block cipher against different cryptanalysis attacks.

5.4.1 Differential & Linear Cryptanalysis

The resistance of a bock cipher against linear [93] and differential [94] cryptanalysis is determined by number of active sboxes in a linear or differential trail [95]. The maximum differential probability (MDP) and linear approximation bias of the sbox S_8 used in dSPN is $2^{-4.4}$[1] and 2^{-3} respectively. Given the linear approximation bias ε, the correlation of linear characteristic is computed as $(2\varepsilon)^2$ [96]. Thus correlation potential of the sbox S_8 is 2^{-4}. In order to mount the differential or linear cryptanalysis on an n-bit block cipher, the attacker requires the differential characteristic and correlation potential to be larger than the 2^{-n} [95]. Based on the differential probability and correlation potential of S_8, dSPN requires 30 differentially and 32 linearly active sboxes to resist against these attacks. From extensive cryptanalysis work on AES and wide trail design strategy, it is known that there are atleast 25 differential and linear active sboxes in any four rounds of the AES cipher [15]. Table 5.3 shows that there are atleast 34 active sboxes in 7 rounds of the dSPN which raises the differential and linear attack complexity to $2^{-149.9}$ and 2^{-136} respectively. Thus 10 round dSPN is secure against differential and linear cryptanalysis.

5.4.2 Boomerang Attack

The boomerang attack and its variants [97, 98, 99] divides the cipher in two halves and then treat each half as a separate sub-cipher. It works by finding boomerang quartet with high probability over these two sub-ciphers. The probability of finding a boomerang quartet is

bounded by multiplication of sbox differential probability with sum of the minimum number of active sboxes in each sub-cipher [32]. From Table 5.3, any combination of two sub-ciphers for 10 rounds of dSPN will have atleast 52 active sboxes (there are atleast 26 active sboxes in any 5 round differential trail of dSPN). Thus complexity of the boomerang attack against full dSPN increases to $4.41 \times 52 = 2^{229.3}$ which is far greater than the brute force.

5.4.3 Algebraic Attack

The algebraic attacks works by modeling the complete cipher in system of equations. The only non-linear component in dSPN is its sbox S_8 which has an algebraic degree of $d = 6$. Table 5.3 shows that any 4 consecutive rounds of dSPN have atleast 25 active sboxes. Thus after four rounds the algebraic degree of the whole cipher reaches its maximum as $d \times 25 = 150 > n$ where n is 128 (the block size). Moreover, any 4-bit sbox S_4 can be described by $e = 21$ quadratic equations over 8 input and output variables over GF(2) [14]. Since there are 32 S_4 in each round of the dSPN, the complete 10 rounds of the cipher consists of $32 \times 10 \times 21 = 6720$ quadratic equations in $32 \times 10 \times 8 = 2560$ variables whereas the 10 rounds AES consists of 6400 equations in 2560 variables. Solving such a large system of equations is still an open problem.

5.4.4 Slide Attack

The slide attack [42] works by exploiting the degree of self similarity in round functions of the block cipher. If all the rounds of a block cipher are identical, then slid pair is found by sliding one instance of the encryption process against another such that both are one round apart [42]. Figure 5.3 illustrates this process. Since addConstants operation mixes round dependent constants in sixteen bits of the state in each round, this removes the similarity between round functions. Moreover, the difference from these 16 bits is spread over 32 bits after the substitution layer and to the complete state after shiftRows and mixColunms operation. Thus performing slide attack over few rounds of the dSPN seems impossible.

5.4.5 Integral Attack

Integral cryptanalysis was introduced with the block cipher SQUARE [79]. It is applicable to SPN ciphers with structure similar to SQUARE such as AES, LED and dSPN. It exploits particular design structure of these SPN ciphers and works independent of the sbox, finite field or key schedule choices. It investigates sum of particular byte values in a set of plain-

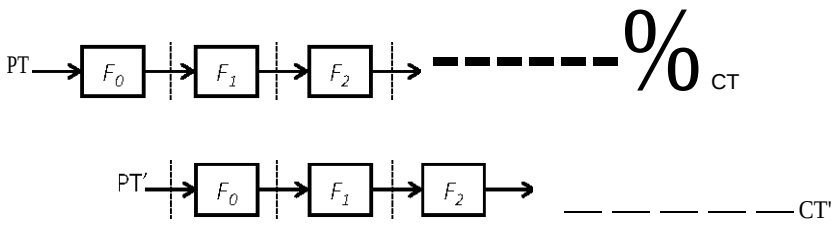
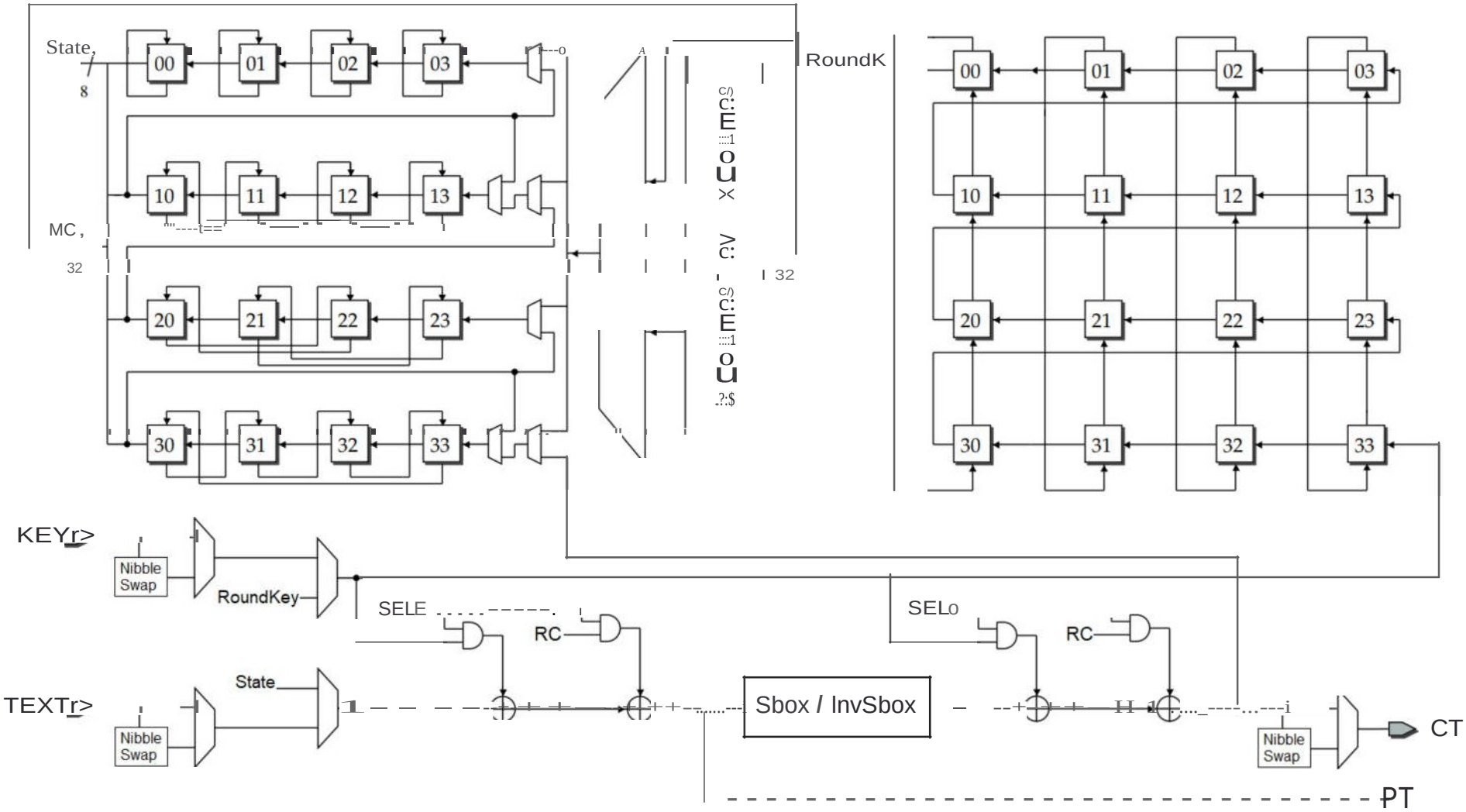


Figure 5.3: Finding Slid Pair for Slide Attack.

text/ cipertext pairs whereas as differential attack work with differences between the pairs. The attacker generates a set of 256 plaintext blocks by setting all bytes to same value except one. Then sum of all changed bytes will be zero after 3 rounds or four rounds with the mix-Columns operation removed. This enable the attcker to recover 4th or 5th round key. Since the integral property exists for only small number of rounds, the full round dSPN is secure against integral cryptanalysis.

5.5 **Implementation Details**

Based on the serial implementations of AES presented in [86, 87], dSPN requires approximately 425 GE lesser than the similar implementation of AES to support both encryption and decryption. The major portion of the resources (around 1466 GE) for both the ciphers is consumed by storage of state and key bits. Figure 5.4 shows the 8-bit implementation architecture of the dSPN. The cost reduction of 425 GE is mainly because of sBox and mix-Columns operation of the dSPN. These operations require an area of 253 and 323 GE for AES where as for dSPN, the cost is reduced to 64 and 165 GE respectively [87]. In order to perform the decryption operation in dSPN, the text and key bytes are loaded after swapping the nibbles. This enables the sBox and mixColumns operations circuit to act as invSbox and invMixColumns respectively. The inverse of shiftRows and addConstants is implemented by a separate circuit The complete implementation requires an area of 2220 GE and 226 clock cycles to process one block by completing one round of dSPN in 21 cycles.



oo

Figure 5.4: The 8-bit architecture of dSPN for Encryption & Decryption.

Chapter6

CONCLUSION & FUTURE WORK

The lightweight block ciphers are designed to have reduced implementation cost and work well in resource constrained environments. But this optimization often leads to design and usage of cryptographic components which have higher implementation cost for the inverse transform. Although feistel and few SPN ciphers attempt to solve the issue but they have their own limitations such as feistel often has higher number of rounds, involutive SPN ciphers have large number of fixed points and reflection ciphers have higher implementation cost for encrypt-only implementation. This thesis dealt with the problem by finding methods to reduce the implementation cost for inverse MDS matrix multiplication and designing the non-involutive cryptographic components which have lesser fixed points but use same implementation for their inverse transform. This helped in reducing the implementation cost of two block ciphers LED and AES. In the end an SPN structure made from these components is proposed which supports decryption with minimal overhead. The thesis provided security analysis of the dSPN structure for few cryptanalysis attacks, thus its use without further analysis is not recommended. The future work guidelines include

- This thesis did not take into account the problem of implementing inverse of the key-schedule with minimal overhead. That is why dSPN used the user supplied key in each round. On the other hand, the block ciphers mostly employ a key-schedule to generate round keys from the master key to thwart related key attacks and its implementation in decryption routine may require handful of resources. Thus studying different key-schedule constructions and improving them to support round key generation in reverse order during decryption with minimal overhead is still an open problem.

- The implementations of cryptographic primitives are often prone to side chaunel attacks and various masking techniques are used to protect against such attacks which have a performance overhead. Thus improving the sbox construction proposed in this thesis for protection against SCA with little overhead is another way forward.

BIBLIOGRAPHY

[1] K. Schwab, *The fourth industrial revolution.* Crown Business, 2017.

[2] M. Hung, "Leading the iot, gartner insights on how to lead in a connected world," *Gartner Research,* pp. 1-29,2017.

[3] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[4] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak sponge function family main document," *Submission to NIST (Round 2),* vol. 3, no. 30, 2009.

[5] W. Wu and L. Zhang, "Lblock: a lightweight block cipher," in *International Conference on Applied Cryptography and Network Security.* Springer, 2011, pp. 327-344.

[6] C. H. Lim and T. Korkishko, "mcrypton-a lightweight block cipher for security of low-cost did tags and sensors," in *International Workshop on Information Security Applications.* Springer, 2005, pp. 243-258.

[7] C. De Canniere, "Trivium: A stream cipher construction inspired by block cipher design principles," in *International Conference on Information Security.* Springer, 2006, pp. 171-186.

[8] J.-P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia, "Quark: A lightweight hash;' in *Cryptographic Hardware and Embedded Systems, CHES 2010.* Springer Berlin Heidelberg, 2010, pp. 1-15.

[9] A. Shamir, "Squash-a new mac with provable security properties for highly constrained devices such as did tags," in *International Workshop on Fast Software Encryption.* Springer, 2008, pp. 144--157.

[10] M. J. Dworkin, "Recommendation for block cipher modes of operation: Galois/counter mode (gem) and gmac," Tech. Rep., 2007.

[11] J. Borghoff, A. Canteaut, T. Giineysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger *et al.,* "Prince-a low-latency block cipher for pervasive computing applications," in *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 2012, pp. 208-225.

[12] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. V!kkelsoe, "Present: An ultra-lightweight block cipher," in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2007, pp. 450-466.

[13] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong *et al.,* "Hight: A new block cipher suitable for low-resource device;• in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2006, pp. 46-59.

[14] Z. Gong, S. Nikova, andY. W. Law, "Klein: a new family of lightweight block ciphers," in *International Workshop on Radio Frequency Identification: Security and Privacy Issues.* Springer, 2011, pp. 1-18.

[15] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw, "The LED block cipher," in *Cryptographic Hardware and Embedded Systems-CHES 2011*. Springer Berlin Heidelberg, 2011, pp. 326-341.

[16] M. Izadi, B. Sadeghiyan, S. S. Sadeghian, and H. A. Khanooki, "Mibs: a new lightweight block cipher;' in *International Conference on Cryptology and Network Security*. Springer, 2009, pp. 334--348.

[17] D. Dinu, L. Perrin, A. Udovenko, V. Velichkov, J. GroBschiidl, and A. Biryukov, "Design strategies for arx with provable bounds: Sparx and lax," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 484--513.

[18] C. Beierle, J. Jean, S. Kolbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sirn, "The SKINNY family of block ciphers and its low-latency variant MANTIS," in *Advances in Cryptology-CRYPTO 2016*. Springer Berlin Heidelberg, 2016, pp. 123-153.

[19] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata, "The 128-bit blockcipher clefia;' in *International Workshop on Fast Software Encryption*. Springer, 2007, pp. 181-195.

[20] I. O. for Standardization, "Information technology - security techniques - lightweight cryptography- part 2: Block ciphers," *ISOREC 29192-2:2012,2012*.

[21] M. Kumar, S. K. Pal, and A. Panigrahi, "Few: A lightweight block cipher." *IACR Cryptology ePrint Archive*, vol. 2014, p. 326, 2014.

[22] F. Karako, H. Demirci, and A. E. Harmanc1, "Itubee: a software oriented lightweight block cipher," in *International Workshop on Lightweight Cryptography for Security and Privacy*. Springer, 2013, pp. 16-27.

[23] V. Grosso, G. Leurent, F.-X. Standaert, and K. Vanct, "Ls-designs: Bitslice encryption for efficient masked software implementations," in *International Workshop on Fast Software Encryption*. Springer, 2014, pp. 18-37.

[24] B. Gerard, V. Grosso, M. Naya-Plasencia, and F.-X. Standaert, "Block ciphers that are easier to mask: How far can we go?" in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2013, pp. 383-399.

[25] G. Piret, T. Roche, and C. Carlet, "Picaro-a block cipher allowing efficient higher-order side-channel resistance," in *International Conference on Applied Cryptography and Network Security*. Springer, 2012, pp. 311-328.

[26] M. Rivain and E. Prouff, "Provably secure higher-order masking of aes," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2010, pp. 413-427.

[27] G. Yang, B. Zhu, V. Suder, M. D. Aagaard, and G. Gong, "The simeck family of lightweight block ciphers;' in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2015, pp. 307-329.

[28] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers, "The simon and speck lightweight block ciphers," in *Design Automation Conference (DAC), 2015 52ndACMIEDAC/JEEE*. IEEE, 2015, pp. 1--6.

[29] M. R. Z'aba, N. Jamil, M. E. Rusli, M. Z. Jamaludin, and A. A.M. Yasir, "I-presenttm: An involutive lightweight block cipher," *Journal of Information Security,* vol. *5,* no. 03, p. 114, 2014.

[30] K. Shibutani, T. Isobe, H. Hiwatari, A. Mitsuda, T. Akishita, and T. Shirai, "Piccolo: an ultra-lightweight blockcipher;' in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2011, pp. 342-357.

[31] T. Suzaki, K. Minematsu, S. Morioka, and E. Kobayashi, "TWINE: A lightweight block cipher for multiple platforms," in *Selected Areas in Cryptography.* Springer Berlin Heidelberg, 2013, pp. 339-354.

[32] S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni, "Midori: a block cipher for low energy," in *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 2014, pp. 411-436.

[33] G. Hatzivasilis, K. Fysarakis, I. Papaefstathiou, and C. Manifavas, "A review of lightweight block ciphers," *Journal of Cryptographic Engineering,* vol. 8, no. 2, pp. 141-184, 2018.

[34] J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen, "Nessie proposal: Noekeon," in *First Open NESSIE Workshop,* 2000, pp. 213-230.

[35] F.-X. Standaert, G. Piret, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat, "Iceberg: An involutional cipher efficient for block encryption in reconfigurable hardware," in *International Workshop on Fast Software Encryption.* Springer, 2004, pp. 279-298.

[36] C. Boura, A. Canteaut, L. R. Knudsen, and G. Leander, "Reflection ciphers," *Designs, Codes and Cryptography,* vol. 82, no. 1-2, pp. 3-25,2017.

[37] J. Guo, J. Jean, I. Nikolic, K. Qiao, Y. Sasaki, and S.M. Sim, "Invariant subspace attack against full midori64." *IACR Cryptology ePrint Archive,* vol. 2015, p. 1189, 2015.

[38] L. R. Knudsen and H. Raddum, "On noekeon, public reports of the nessie project: Nes/doc/uib/wp3/009," 2001.

[39] L. Batina, A. Das, B. Ege, E. B. Kavun, N. Mentens, C. Paar, I. Verbauwhede, and T. Yal tm, "Dietary recommendations for lightweight block ciphers: power, energy and area analysis of recently developed architectures," in *International Workshop on Radio Frequency Identification: Security and Privacy Issues.* Springer, 2013, pp. 103-112.

[40] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard.* Springer Science & Business Media, 2013.

[41] A. Journault, F.-X. Standaert, and K. Varici, "Improving the security and efficiency of block ciphers based on Is-designs," *Designs, Codes and Cryptography,* vol. 82, no. 1-2, pp.495-509,2017.

[42] A. Biryukov and D. Wagner, "Slide attacks," in *International Workshop on Fast Software Encryption.* Springer, 1999, pp. 245-259.

[43] ——, "Advanced slide attacks," in *International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 2000, pp. 589-606.

[44] E. Biharn, "A fast new des implementation in software," in *International Workshop on Fast Software Encryption.* Springer, 1997, pp. 260-272.

[45] C. Rebeiro, D. Selvakumar, and A. Devi, "Bitslice implementation of aes," in *International Conference on Cryptology and Network Security.* Springer, 2006, pp. 203-212.

[46] M. Hamburg, "Accelerating aes with vector permute instructions," in *Cryptographic Hardware and Embedded Systems-CHES 2009.* Springer, 2009, pp. 18-32.

[47] S. Matsuda and S. Moriai, "Lightweight cryptography for the cloud: exploit the power of bitslice implementation," in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2012, pp. 408-425.

[48] T. Park, H. Seo, and H. Kim, "Fast implementation of simeck family block ciphers using avx2," in *20I8 International Conference on Platform Technology and Service (PlatCon).* IEEE, 2018, pp. 1-6.

[49] B. Lac, A. Canteaut, J. J. Fournier, and R. Sirdey, "Thwarting fault attacks against lightweight cryptography using simd instructions," in *Circuits and Systems (ISCAS), 20I8 IEEE International Symposium on.* IEEE, 2018, pp. 1-5.

[50] T. Kranz, G. Leander, K. Stoffelen, and F. Wiemer, "Shorter linear straight-line programs for mds matrices," *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 4, pp. 188-211, Dec. 2017. [Online]. Available: https://tosc.iacr.orglindex.php/ToSC/article/view/813

[51] S. Li, S. Sun, C. Li, Z. Wei, and L. Hu, "Constructing low-latency involutory mds matrices with lightweight circuits," *IACR Transactions on Symmetric Cryptology*, pp. 84-117,2019.

[52] M. Liu and S. M. Sim, "Lightweight MDS generalized circulant matrices," in *Fast Software Encryption.* Springer Berlin Heidelberg, 2016, pp. 101-120. [Online]. Available: https://doi.org/10.1007/978-3-662-52993-5_6

[53] R. Avanzi, "The qarrna block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *IACR Transactions on Symmetric Cryptology*, pp. 4-44,2017.

[54] A. Mahmoodi Rishakani, M. R. Mirzaee Sharnsabad, S. Dehnavi, M. A. Amiri, H. Maimani, and N. Bagheri, "Lightweight 4x4 mds matrices for hardware-uriented cryptographic primitives," *The ISC International Journal of Information Security*, vol. 11, no. 1, pp. 35-46, 2019.

[55] A. M. Rishakani, Y. F. Dabanloo, S. M. Dehnavi, M. M. Sharnsabad, and N. Bagheri, "A note on the construction of lightweight cyclic mds matrices." *IJ Network Security*, vol.21,no.2,pp.269-274,2019.

[56] S. Sarkar and S. M. Sim, "A deeper understanding of the xor count distribution in the context of lightweight cryptography," in *International Conference on Cryptology in Africa.* Springer, 2016, pp. 167-182.

[57] R. Lidl and H. Niederreiter, *Finite fields.* Cambridge university press, 1997, vol. 20.

[58] D. Toh, J. Teo, K. Khoo, and S. M. Sim, "Lightweight MDS serial-type matrices with minimal fixed XOR count," in *Progress in Cryptology – AFRICACRYPT 2018.* Springer International Publishing, 2018, pp. 51-71. [Online]. Available: https://doi.org/10.1007/978-3-319-89339-6_4

[59] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[60] C. Paar, "Optimized arithmetic for reed-solomon encoders," in *Proceedings of IEEE International Symposium on Information Theory.* IEEE, 1997, p. 250.

[61] K. Khoo, T. Peyrin, A. Y. Poschmann, and H. Yap, "Foam: searching for hardware-optimal spn structures and components with a fair comparison," in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2014, pp. 433-450.

[62] S. M. Sim, K. Khoo, F. Oggier, and T. Peyrin, "Lightweight mds involution matrices," in *International Workshop on Fast Software Encryption.* Springer, 2015, pp. 471-493.

[63] Y. Li and M. Wang, "On the construction of lightweight circulant involutory mds matrices;' in *International Conference on Fast Software Encryption.* Springer, 2016, pp. 121-139.

[64] S. Sarkar and H. Syed, "Lightweight diffusion layer: Importance of toeplitz matrices," *IACR Transactions on Symmetric Cryptology,* pp. 95-113,2016.

[65] J. Jean, T. Peyrin, S. M. Sim, and J. Tourteaux, "Optimizing implementations of lightweight building blocks," *IACR Transactions on Symmetric Cryptology,* pp. 130--168, 2017.

[66] J. Boyar, P. Matthews, and R. Peralta, "Logic minimization techniques with applications to cryptology," *Journal of Cryptology,* vol. 26, no. 2, pp. 280--312,2013.

[67] C. Fuhs and P. Schneider-Kamp, "Synthesizing shortest linear straight-line programs over gf (2) using sat," in *International Conference on Theory and Applications of Satisfiability Testing.* Springer, 2010, pp. 71-84.

[68] J. Boyar, P. Matthews, and R. Peralta, "On the shortest linear straight-line program for computing linear forms," in *International Symposium on Mathematical Foundations of Computer Science.* Springer, 2008, pp. 168--179.

[69] J. Boyar and R. Peralta, "A new combinational logic minimization technique with applications to cryptology," in *International Symposium on Experimental Algorithms.* Springer, 2010, pp. 178--189.

[70] A. Visconti, C. V. Schiavo, and R. Peralta, "Improved upper bounds for the expected circuit complexity of dense systems of linear equations over gf (2);' *Information Processing Letters,* vol. 137, pp. 1-5, 2018.

[71] J. Boyar, R. Peralta *et al.,* "Small low-depth circuits for cryptographic applications," *Cryptography and Communications,* vol. 11, no. 1, pp. 109-127,2019.

[72] J. Jean, A. Moradi, T. Peyrin, and P. Sasdrich, "Bit-sliding: A generic technique for bit-serial implementations of spn-based primitives," in *International Conference on Cryptographic Hardware and Embedded Systems.* Springer, 2017, pp. 687-707.

[73] S. Vaudenay, "On the need for multipermutations: Cryptanalysis of MD4 and SAFER;' in *Fast Software Encryption.* Springer Berlin Heidelberg, 1995, pp. 286-297. [Online]. Available: https://doi.org/10.1007/3-540-60590-8_22

[74] H. Mattson, Jr, "The theory of error-correcting codes (fj macwilliams and nja sloane)," *SIAM Review,* vol. 22, no. 4, pp. 513-519, 1980.

[75] K. C. Gupta and I. G. Ray, "On constructions of mds matrices from circulant-like matrices for lightweight cryptography," *Tech. Rep. ASU/20I4/l,* 2014.

[76] P. Flajolet and R. Sedgewick, *Analytic combinatorics.* cambridge University press, 2009.

[77] G. Leander, M. A. Abdelraheem, H. AlKhzairni, and E. Zenner, "A cryptanalysis of printcipher: the invariant subspace attack," in *Annual Cryptology Conference.* Springer, 2011, pp. 206-221.

[78] G. Leander, B. Minaud, and S. R!llnjom, "A generic approach to invariant subspace attacks: Cryptanalysis of robin, iscream and zorro," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 2015, pp. 254-283.

[79] J. Daemen, L. Knudsen, and V. Rijmen, "The block cipher square," in *International Workshop on Fast Software Encryption.* Springer, 1997, pp. 149-165.

[80] J. Nakahara Jr and E. Abrahao, "A new involutory mds matrix for the aes." *IJ Network Security,* vol. 9, no. 2, pp. 109-116, 2009.

[81] J. Guo, T. Peyrin, and A. Poschmann, "The photon family of lightweight hash functions," in *Annual Cryptology Conference.* Springer, 2011, pp. 222-239.

[82] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw, "The led block cipher," in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2011, pp. 326-341.

[83] K. C. Gupta and I. G. Ray, "On constructions of mds matrices from companion matrices for lightweight cryptography," in *International Coriference on Availability, Reliability, and Security.* Springer, 2013, pp. 29-43.

[84] A. Bogdanov, D. Khovratovich, and C. Rechberger, "Biclique cryptanalysis of the full aes;• in *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 2011, pp. 344-371.

[85] S. Banik, A. Bogdanov, and F. Regazzoni, "Atornic-aes v 2.0." *IACR Cryptology ePrint Archive,* vol. 2016, p. 1005, 2016.

[86] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang, "Pushing the limits: a very compact and a threshold implementation of aes," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 2011, pp. 69-88.

[87] S. Banik, A. Bogdanov, and F. Regazzoni, "Atomic-aes: A compact implementation of the aes encryption/decryption core," in *International Conference in Cryptology in India.* Springer, 2016, pp. 173-190.

[88] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen, "Aes implementation on a grain of sand," *lEE Proceedings-Information Security,* vol. 152, no. 1, pp. 13-20, 2005.

[89] S. Mathew, S. Satpathy, V. Suresh, M. Anders, H. Kaul, A. Agarwal, S. Hsu, G. Chen, and R. Krishnamurthy, "340 mv-1.1 v, 289 gbps/w, 2090-gate nanoaes hardware accelerator with area-optimized encrypt/decrypt gf (2 4) 2 polynomials in 22 nm tri-gate cmos," *IEEE Journal of Solid-State Circuits,* vol. 50, no. 4, pp. 1048-1058, 2015.

[90] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A compact rijndael hardware architecture with s-box optimization," in *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 2001, pp. 239-254.

[91] J. Choy, **H.** Yap, K. Khoo, J. Guo, T. Peyrin, A. Poschmann, and C. **H.** Tan, "Spn-hash: improving the provable resistance against differential collision attacks," in *International Conference on Cryptology in Africa.* Springer, 2012, pp. 270--286.

[92] A. Biryukov, D. Khovratovich, and **I.** Nikolic, "Distinguisher and related-key attack on the full aes-256," in *Annual International Cryptology Conference.* Springer, 2009, pp. 231-249.

[93] M. Matsui, "The first experimental cryptanalysis of the data encryption standard," in *AnnuallnternationaI Cryptology Conference.* Springer, 1994, pp. 1-11.

[94] E. Biham and A. Shamir, *Differential cryptanalysis of the data encryption standard.* Springer Science & Business Media, 2012.

[95] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo, "Gift: a small present," in *International Conference on Cryptographic Hardware and Embedded Systems.* Springer, 2017, pp. 321-345.

[96] M. Matsui, "New structure of block ciphers with provable security against differential and linear cryptanalysis," in *International Workshop on Fast Software Encryption.* Springer, 1996, pp. 205-218.

[97] D. Wagner, "The boomerang attack," in *International Workshop on Fast Software Encryption.* Springer, 1999, pp. 156-170.

[98] J. Kelsey, T. Kohno, and **B.** Schneier, "Amplified boomerang attacks against reduced-round mars and serpent," in *International Workshop on Fast Software Encryption.* Springer, 2000, pp. 75-93.

[99] E. Biham, O. Dunkelman, and N. Keller, "The rectangle attack-rectangling the serpent," in *International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 2001, pp. 340--357.

Finite Field Multiplication

A nibble(4-bit) can be represented as polynomial with bits as coefficients in GF(2).

$$b3b2b1bo\ \text{r+}\ b(x)$$

$$b(x) = \text{b3x}^3 + \text{b2x}^2 + b1x + bo$$

According to this, the polynomial representation of 2 is x and field multiplication is performed as

$$\begin{aligned} b.x &= ((\text{b}_3\text{x}^3 + \text{b}_2\text{x}^2 + b1x + b_0)\ .\ x)\ mod\ (x^4 + x + 1) \\ &= (b3x^4 + \ \times^3 + b1x^2 + box)\ mod\ (x^4 + x + 1) \\ &= \text{b2x}^3 + b1x^2 + (bo\ \text{E9}\ b3)x + \text{b3} \end{aligned}$$

The modulo operation is performed if degree of resultant polynomial is greater than 3 which is dependent upon the bit b_3. Thus multiplication with 2 is performed by a left shift of one bit and conditional xor with $x + 1$.

Multiplication Matrices over GF(2) and Bitwise Field Multiplication

Table B.l: Binary Matrices for Finite Field Multiplication over $GF(2^4)$. Cell (4,3) shows the Multiplication Matrix for element 0x07.

cell	0	1	2	3
0	0000 0000 0000 0000	1000 0100 0010 0001	0100 0010 1001 1000	1100 0110 1011 1001
4	0010 1001 1100 0100	1010 1101 1110 0101	0110 1011 0101 1100	1110 1111 0111 1101
8	1001 1100 0110 0010	0001 1000 0100 0011	1101 1110 1111 1010	0101 1010 1101 1011
12	1011 0101 1010 0110	0011 0001 1000 0111	1111 0111 0011 1110	0111 0011 0001 1111

Table B.2: Bitwise Finite Field Multiplication in $GF(2^4)$/0x13. A nibble consists of 4-bits as *bab2b1bo.*

Element	*Bit*a	*Bit2*	*Bit1*	*Bito*
0	0	0	0	0
1	*ba*		bl	*bo*
2	b2	bl	*ba* EB bo	*ba*
3	*ba* EB	b2 EB *b1*	*ba* EB *b1* EB bo	*ba* EB *bo*
4	bl	*ba* EB *bo*	*ba* EB b2	
5	*ba* EB *b1*	baEB EBbo	baEB EBb1	b2 EB bo
6	b2 EB *b1*	*ba* EB *b1* EB bo	EB bo	*ba* EB
7	*ba* EB b2 EB *b1*	*ba* EB b2 EB *b1* EB bo	b2 EB *b1* EB bo	baEB EBbo
8	*ba* EB bo	*ba* EB	EB *b1*	bl
9	*bo*	*ba*		*b1* EB *bo*
A	*ba* EB b2 EB *bo*	baEB EBb1	baEB EBb1EBbo	*ba* EB *b1*
B	b2 EB *bo*	*ba* EB *b1*	baEB EBbo	*ba* EB *b1* EB bo
C	*ba* EB *b1* EB *bo*	b2 EB *bo*	*ba* EB *b1*	b2 EB *b1*
D	*b1* EB bo	bo	*ba*	b2 EB *b1* EB bo
E	*ba* EB b2 EB *b1* EB bo	b2 EB *b1* EB bo	*b1* EB bo	baEB El1b1
F	b2 EB *b1* EB bo	*b1* EB *bo*	*bo*	*ba* EB b2 EB *b1* EB bo

Field Multiplication Xor Count

Table C.1: Number of xor Operations required for each type of Finite Field Multiplication.

Element	DM	d-Xor	s-Xor	Element	DM	d-Xor	s-Xor
0	0	0	0	8	3	3	3
1	0	0	0	9	7	1	1
2	1	1	1	A	8	8	4
3	5	5	4	B	12	6	4
4	2	2	2	C	9	5	4
5	6	6	4	D	13	3	2
6	7	5	5	E	14	8	4
7	11	9	5	F	18	6	3

SLP Heuristic

The initial baseS consists of input signals (x_3, x_2, x_1, x_0).

$$S = \{[1\,0\,0\,0],\ [0\,1\,0\,0],\ [0\,0\,1\,0],\ [0\,0\,0\,1]\}$$

The output signal $(Ya, Y2, Y1, Yo)$ to be computed is

$$\begin{vmatrix} 1\,1\,0\,0 \\ 0\,1\,1\,0 \\ 1\,0\,1\,1 \\ 1\,0\,0\,1 \end{vmatrix} \begin{vmatrix} x_3 \\ x_2 \\ x_1 \\ x_0 \end{vmatrix} \quad \begin{vmatrix} Ya \\ Y2 \\ Y1 \\ Yo \end{vmatrix}$$

Then the initial distance vector D is simply one less than the hamming weight of each row of the output matrix M i.e $D = [1\,1\,2\,1]$. The heuristic finds two bases from S such that their addition either reduces the distance or leads to a target signal. The addition of first two bases x_3 and x_2 leads to first target signal i.e $y_3 = x_3 + x_2$. Thus new distance vector is $D = [0\,1\,2\,1]$. Table D.1shows the complete run of the SLP heuristic.

Table D.1: Example running of Heuristic for finite field multiplication by 0x03.

STEP	New Base	New Distance
$Ya = xa + x2$	[1 1 0 0]	[0 1 2 1]
$Y2 = x2 + x1$	[0 1 1 0]	[0 0 2 1]
$Yo = xa + xo$	[1 0 0 1]	[0 0 1 0]
$Y1 = x1 + Yo$	[1 0 1 1]	[0 0 0 0]

Mixed Implementation Matrix for M and M^{-1}

$$Mm;=[M]=\begin{vmatrix} 2 & C & 9 & D \\ D & 2 & C & 9 \\ 9 & D & 2 & C \\ C & 9 & D & 2 \\ F & 2 & C & D \\ D & F & 2 & C \\ C & D & F & 2 \\ 2 & C & D & F \end{vmatrix}$$

$$\begin{vmatrix}
0&1&0&0&1&0&1&1&0&0&0&1&0&0&1&1\\
0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&1\\
1&0&0&1&1&0&1&0&0&1&0&0&1&0&0&0\\
1&0&0&0&0&1&1&0&0&0&1&1&0&1&1&1\\
0&0&1&1&0&1&0&0&1&0&1&1&0&0&0&1\\
0&0&0&1&0&0&1&0&0&1&0&1&1&0&0&0\\
1&0&0&0&1&0&0&1&1&0&1&0&0&1&0&0\\
0&1&1&1&1&0&0&0&0&1&1&0&0&0&1&1\\
0&0&0&1&0&0&1&1&0&1&0&0&1&0&1&1\\
1&0&0&0&0&0&0&1&0&0&1&0&0&1&0&1\\
0&1&0&0&1&0&0&0&1&0&0&1&1&0&1&0\\
0&0&1&1&0&1&1&1&1&0&0&0&0&1&1&0\\
1&0&1&1&0&0&0&1&0&0&1&1&0&1&0&0\\
0&1&0&1&1&0&0&0&0&0&0&1&0&0&1&0\\
1&0&1&0&0&1&0&0&1&0&0&0&1&0&0&1\\
0&1&1&0&0&0&1&1&0&1&1&1&1&0&0&0\\
0&1&1&1&0&1&0&0&1&0&1&1&0&0&1&1\\
0&0&1&1&0&0&1&0&0&1&0&1&0&0&0&1\\
0&0&0&1&1&0&0&1&1&0&1&0&1&0&0&0\\
1&1&1&1&1&0&0&0&0&1&1&0&0&1&1&1\\
0&0&1&1&0&1&1&1&0&1&0&0&1&0&1&1\\
0&0&0&1&0&0&1&1&0&0&1&0&0&1&0&1\\
1&0&0&0&0&0&0&1&1&0&0&1&1&0&1&0\\
0&1&1&1&1&1&1&1&1&1&0&0&0&0&1&1&0\\
1&0&1&1&0&0&1&1&0&1&1&1&0&1&0&0\\
0&1&0&1&0&0&0&1&0&0&1&1&0&0&1&0\\
1&0&1&0&1&0&0&0&0&0&0&1&1&0&0&1\\
0&1&1&0&0&1&1&1&1&1&1&1&1&0&0&0\\
0&1&0&0&1&0&1&1&0&0&1&1&0&1&1&1\\
0&0&1&0&0&1&0&1&0&0&0&1&0&0&1&1\\
1&0&0&1&1&0&1&0&1&0&0&0&0&0&0&1\\
1&0&0&0&0&1&1&0&0&1&1&1&1&1&1&1
\end{vmatrix}_{32x16}$$

Implementation Details for Matrix Multiplication with M and M^{-1}

Table F.1: Matrix Multiplication with MDS matrix M by 45 xor operations.

STEP	operation	STEP	operation	STEP	operation
1	to= X12 + X4	16	t15 = *X6* + *X3*	31	*Ys* = t2s + t29
2	t1 = *Xs* + *Xo*	17	t16 = X1 + t13	32	t31 = X2 + t10
3	t2 = X15 + *Xs*	18	*Y1* = t1s + t16	33	Y4 = t16 + t31
4	t3 = X2 + t2	19	t1B = t12 + t15	34	t33 = X14 + X2
5	*Y6* = t1 + t3	20	*Yw* = X4 +hs	35	t34 = *Y3* + t22
6	ts = X14 + xn	21	t2o=to+Y6	36	*Yo*= t33 + t34
7	*t6* = x1 + ts	22	*Y3* = ts +t2o	37	t36 = X13 + *X10*
8	*Y2=to+t6*	23	t22 = X15 + t1B	38	t37 = t16 + Y3
9	ts = x13 + *xo*	24	Y13 = xn + t22	39	Y12 = t36 + t37
10	t9 = *xw* +*xr*	25	t24 = X15 + X3	40	t39 = X14 + tg
11	*tw* = ts + tg	26	Y1 = *tw* + t24	41	t4o = Y1s + Y12
12	Y14 = *Xs* + t10	27	t26 = *Xs* +to	42	*yg* = t39 + t40
13	t12 = X12 + *Xg*	28	Yn = t10 + t26	43	t42 = *X10* + t15
14	t13 = tl + t12	29	t2B = X12 + Y2	44	t43 = Yn + *Ys*
15	Y1s = Y2 + t13	30	t29 = *X7* + X3	45	*YB* = t42 +t43

Table F.2: Matrix Multiplication with matrix M^{-1} by 50 xor operations.

STEP	operation	STEP	operation	STEP	operation
1	to = X1a + X4	18	t17 = X15 + Xo	35	YI5 = t2o + taa
2	t1 = X12 + Xg	19	hs = t6 + t11	36	ta5 = X5 + Y1
3	t2 = Xs + x5	20	Y1 = YI4 + t1s	37	Y12 = t27 + ta5
4	ta = XJ + xo	21	t2o = t4 + t15	38	ta1 = xa + t2
5	t4 = xw + ta	22	t21 = XJo + t14	39	Y1a = t25 + ta1
6	t5 = xo + t1	23	Yu = YI4 + t21	40	tag = x14 + XJ
7	t6 = X2 + t2	24	t2a = X7 + t14	41	t4o = YI5 + tag
8	Yw = t5 + t6	25	yg = X15 + t2a	42	Ys = Ya + t4o
9	ts = X14 + X12	26	t25 = x12 + tw	43	t42 = t2a + ta5
10	tg = X4 + x1	27	t26 = xn + t5	44	Yo = t40 + t42
11	tw = xu + X7	28	t21 = Y6 + t26	45	t44 = XJa + XJO
12	tu = xG + to	29	Ya = x2 + t27	46	t45 = Yu + YI5
13	YI4 = t5 + tu	30	t2g = X15 + X7	47	Y4 = t44 + t45
14	ha = Xs + tg	31	Y1 = t26 + t2g	48	t47 = xu + Xa
15	t14 = X3 + t1a	32	ta1 = Xs + to	49	t4s = to + t11
16	t15 = X5 + ts	33	Y2 = t4 + tal	50	Y5 = t41 + t4s
17	Y6 = t1a + t15	34	taa = x1 + to		

Table F.3: Mixed Implementation(MI) for multiplication with matrix M and M^{-1} by 57 xor operations. Target signals Ya1 to Y16 and Y1s to *Yo* corresponds to multiplication with Matrix M and M^{-1} respectively.

STEP	operation	STEP	operation	STEP	operation
1	*to* = X1a + *Xo*	20	*Y16* = t6 + *Y4*	39	Y21 = ts + ta7
2	t1 = X12 + *Xg*	21	t2o = Xn + *Y6*	40	Y1a = *Y6* + Y21
3	t2 = Xs + *xs*	22	Y1s = t2 + t2o	41	t4o = t1 + *t7*
4	ta = *X4* + x1	23	t22 = *X4* + *to*	42	Y29 = t34 + t40
5	t4 = X15 + X2	24	t23 = *X6* + t22	43	Y26 = *X4* + t4o
6	ts = x14 + *xu*	25	Y14 = t1 + t2a	44	Yn = Y2 + Y26
7	*t6* = *xw* + *x7*	26	t25 = *Xo* + t4	45	*Ys* = Y14 + Y29
8	*t7* = *x6* + *xa*	27	Y22 = t2 + t2s	46	t45 = *Y4* + Y12
9	ts=to+t6	28	*Y1* = Y14 + Y22	47	Y2s = Y1a + t45
10	*Yao* = Xs + ts	29	t2s = tn + t2o	48	Y2o = *Ys* + t45
11	*tw* = x12 + t2	30	Ya1 = *Xs* + t2s	49	Ys = ts + Y2o
12	tn = *xo* + t1	31	Y12 = *Y7* + Ya1	50	*Y21=Ya+Ys*
13	t12 = ta + *tw*	32	Y24 = t4 + Y12	51	*Yo=t1+Y2s*
14	*Y6* = X14 + t12	33	Ya=x2+t2s	52	Y19 = Yn + *Yo*
15	Y1s = *Yao* + *Y6*	34	*Yw* = Y1s + *Ya*	53	ts2 = Ys + *Yo*
16	t1s = x1 + tn	35	ta4 = X15 + *xu*	54	Y9 = Y24 + ts2
17	t16 = *t7* + t15	36	tas = *X7* + ta	55	*Y11* = Y2 + *yg*
18	Y2a = Xs + t16	37	Y2 = *Yao* + tas	56	Y1 = Y16 + ts2
19	*Y4* = Y15 + Y2a	38	ta7 = *xa* + tas	57	Y25 = *Yw* + Y1

Table F.4: Derived Implementation(DI) for multiplication with matrix M and M^{-1} by 69 xor operations. Target signals y15 to y_0 and z15 to z_0 corresponds to multiplication with Matrix M and M^{-1} respectively.

STEP	operation	STEP	operation	STEP	operation
1	to= X12 + X4	24	Y13 = xu + t22	47	t46 = X13 + Yw
2	t1 = Xs +xo	25	t24 = X15 + X3	48	t47 = Xg + Y6
3	t2 = X15 + X5	26	Y1 = tw + t24	49	t48 = X15 + X12
4	t3 = X2 + t2	27	t26 = X5 +to	50	zw = t47 + t4s
5	Y6 = t1 + t3	28	Yu = tw + t26	51	Z1 = Y9 + Z10
6	t5 = X14 +Xu	29	t28 = X12 + Y2	52	t51 = Xo + t46
7	t6 = X1 + t5	30	t29 = X7 + X3	53	Z14 = X3 + t51
8	Y2=to+t6	31	Ys = t2s + t29	54	zr = Y6 + Z14
9	ts = X13 + Xo	32	t31 = x2 + tw	55	Z12 = Y15 + Z7
10	t9 = X10 + X7	33	Y4 = t16 + t31	56	zs = Y13 + Z14
11	tw = ts + tg	34	t33 = X14 + X2	57	ts6 = Y4 + zs
12	Y14 = Xs + tw	35	t34 = Y3 + t22	58	Z13 = Y12 + ts6
13	t12 = X12 + Xg	36	Yo= t33 + t34	59	z6 = Ys + Z13
14	t13 = t1 + t12	37	t36 = X13 + X10	60	Z15 = Y14 + Z6
15	Y15 = Y2 + t13	38	t37 = t16 + Y3	61	Z4 = Z12 + t56
16	hs = X6 + X3	39	Y12 = t36 + t37	62	t61 = X13 + t42
17	t16 = X1 + t13	40	t39 = X14 + tg	63	Zn = Y1 + t36
18	Y1 = t1s + t16	41	t4o = Y1s + Y12	64	Z2 = Y10 + Zn
19	t1s = t12 + t1s	42	yg = t39 + t40	65	Zg=y1+z2
20	Yw = X4 + t1s	43	t42 = xw + t1s	66	zo = t37 + t61
21	t2o=to+Y6	44	t43 = Yn + Ys	67	t66 = X15 + t3
22	Y3=ts+t2o	45	Ys = t42 + t43	68	Z3 = Y15 + t66
23	t22 = X15 + t1s	46	t45 = X15 + X14	69	zs=Yn+z3