Efficient Implementation Strategies for Block Ciphers on ARMv8

Bachelorarbeit

Bastian Engel

February 27, 2023

Abstract

Lorem ipsum dolor [1] sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Declaration

I hereby declare that ...

Contents

Chapter 1

Introduction

1.1 Block ciphers

Securing communication channels between different parties has been a long-term subject of study for cryptographers and engineers which is essential to our modern world to cope with ever-increasing amounts of devices producing and sharing data. The main way to facilitate high-throughput, confidential communications nowadays is through the use of symmetric cryptography in which two parties share a common secret, called a key, which allows them to encrypt, share and subsequently decrypt messages to achieve confidentiality against third parties. Ciphers can be divided into two categories; block ciphers, which always encrypt fixed-sized messages called blocks, and stream ciphers, which continuously provide encryption for an arbitrarily long, constant stream of data.

A block cipher can be defined as a bijection between the input block (the message) and the output block (the ciphertext). For any block cipher with block size n, we denote the key-dependent encryption and decryption functions as $E_K, D_K : \mathbb{F}_2^n \to \mathbb{F}_2^n$. The simplest way to characterize this bijection is through a lookup table which yields the highest possible performance as each block can be encrypted by one simple lookup depending on the key and the message. This is not practical though due to most ciphers working with block and key sizes $n, |K| \geq 64$. For a block cipher with $n = 64, |K| = 128$, a space of $2^{64} 2^{128} 64 = 2^{198}$ is necessary. Considering modern consumer hard disks being able to store data in the order of 2^{40}, it is easy to see that a lookup table is wholly impractical. We therefore describe block ciphers al-

gorithmically which opens up possibilities for different tradeoffs and security concerns.

1.1.1 GIFT

GIFT[1], first presented in the *CHES 2017* cryptographic hardware and embedded systems conference, is a lightweight block cipher based on a previous design called |PRESENT|, developed in 2007. Its goal is to offer maximum security while being extremely light on resources. Modern battery-powered devices like RFID tags or low-latency operations like on-the-fly disc encryption present strong hardware and power constraints. GIFT aims to be a simple, low-energy cipher suited for these kinds of applications.

GIFT comes in two variants; GIFT-64 working with 64-bit blocks and GIFT-128 working with 128-bit blocks. In both cases, the key is 128 bits long. The design is a very simple, round-based substitution-permutation network (SPN). One round consists in a sequential application of the confusion layer by means of 4-bit S-boxes and subsequent diffusion through bit permutation. After the bit permutation, a round key is added to the cipher state and the single round is complete. GIFT-64 uses 28 rounds while GIFT-128 uses 40 rounds.
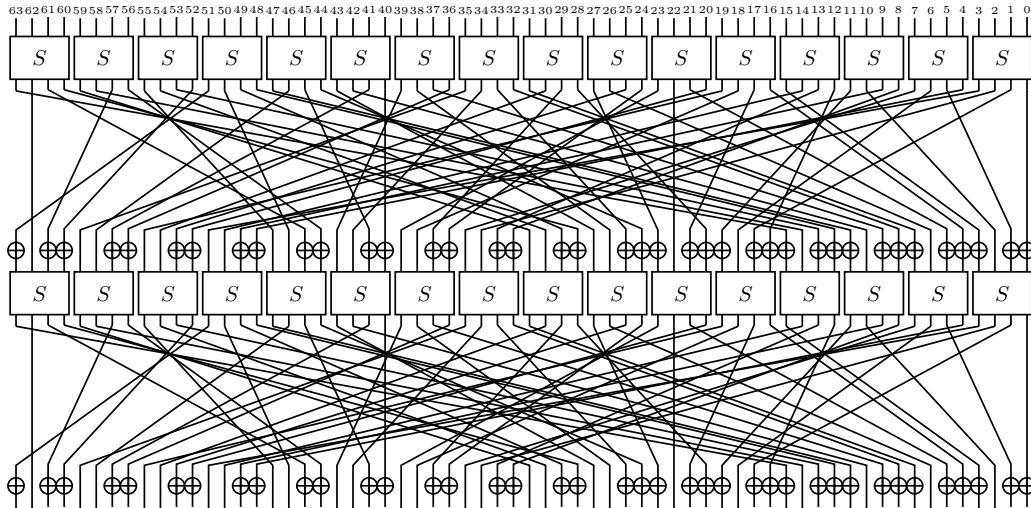

Figure 1.1: Two rounds of GIFT-64

5

Substitution layer

The input of GIFT is split into 4-bit nibbles which are then fed into 16 S-boxes for GIFT-64 and 32 S-boxes for GIFT-128. The S-box $S : \mathbb{F}_2^4 \to \mathbb{F}_2^4$ is defined as follows:

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

Permutation layer

The permutation P works on individual bits and maps bit b_i to $b_{P(i)}, \forall i \in \{0, 1, \ldots, n-1\}$. The different permutations for GIFT-64 and GIFT-128 can be expressed by:

$$P_{64}(i) = 4 \left\lfloor \frac{i}{16} \right\rfloor + 16 \left(\left(3 \left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4) \right) \bmod 4 \right) + (i \bmod 4)$$

$$P_{128}(i) = 4 \left\lfloor \frac{i}{16} \right\rfloor + 32 \left(\left(3 \left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4) \right) \bmod 4 \right) + (i \bmod 4)$$

Round key addition

The last step of each round consists in XORing a round key R_i to the cipher state. The new cipher state s_{i+1} after each full round is therefore given by

$$s_{i+1} = P(S(s_i)) \oplus R_i$$

Round key extraction and key schedule

Round key extraction differs for GIFT-64 and GIFT-128. Let $K = k7||k6|| \ldots ||k0$ denote the 128-bit key state.

GIFT-64 . We extract two 16-bit words $U||V = k_1||k_0$ from the key state. u_i and v_i are XORed to r_{4i+1} and r_{4i} of the round key R respectively.

GIFT-128 . We extract two 32-bit words $U||V = k_5||k4||k1||k_0$ from the key state. u_i and v_i are XORed to r_{4i+2} and b_{4i+1} of the round key R respectively.

In both cases, we additionally XOR a round constant $C = c_5c_4c_3c_2c_1c_0$ to bit positions $n-1, 23, 19, 15, 11, 7, 3$. The round constants are generated using a 6-bit affine linear-feedback shift register and have the following values:

Rounds	Constants
1 - 16	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E
17 - 32	1D,3A,35,2B,16,2C,18,30,21,02,05,0B,17,2E,1C,38
33 - 48	31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04

The key state is then updated by setting $k_1 \leftarrow k_1 \ggg 2$, $k_0 \leftarrow k_0 \ggg 12$ and rotating the new state 32 bits to the right:

$$k_7||k_6||\dots||k_1||k_0 \leftarrow k_1 \ggg 2||k_0 \ggg 12||k_7||k_6||\dots||k_3||k_2$$

1.1.2 Camellia

1.2 The ARMv8 platform

With small devices, embedded processors and ASICs becoming ever more ubiquitous and essential in areas like medicine or automotive design, the need for ...

Chapter 2

Implementation strategies

Due to the structural differences of SPN- and Feistel network-based ciphers, we shall analyze these two separately.

2.1 Strategies for SPN

Three implementation strategies for substitution-permutation networks are introduced by [2]:

- Table-based implementations

- `vperm` implementations

- Bitslice implementations

2.1.1 Table-based

Table-driven programming is a simple way to increase performance of operations by tabulating the results, therefore requiring only a single memory access to acquire the result. This approach is obviously limited to manageable table sizes, so while tabulating a function like the AES S-box $S_{AES} : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ requires only 2^{11} space, tabulating the `GIFT` permutation layer $P_{GIFT} : \mathbb{F}_2^{64} \to \mathbb{F}_2^{64}$ would require 2^{70} space, which is totally unfeasible.

A common approach is to tabulate the output of each S-box, including the diffusion layer, and then XORing the results together. Let n denote the internal cipher state size and s the size of a single S-box in bits. For

each S-box $S_i, i \in \{0, \ldots, \frac{n}{s}\}$, we can construct a mapping $T_i : \mathbb{F}_2^s \to \mathbb{F}_2^n$ representing substitution with subsequent permutation of that single S-box. The cipher state before round key addition is then given by $\bigoplus_{i=0}^{\frac{n}{s}-1} T_i(m_i)$ for each s-bit message chunk m_i. This approach requires space of $\frac{n}{s}|\mathbb{F}_2^s|n = \frac{n^2 2^s}{s}$ bits, which, for GIFT-64, results in a manageable size of $\frac{64^2 2^4}{4} = 2^{14}$ bits which equals 16 KiB.

Constructing the tables

For GIFT-64, table construction is relatively straightforward and can be done as follows:

Listing 2.1: Table construction algorithm

```
1    tables <- [][]
2    for sbox_index from 0 to 15 do
3        for sbox_input from 0 to 15 do
4            output <- sbox(sbox_input)
5            output <- permute(output << (4 * sbox_index))
6            tables[sbox_index][sbox_input] <- output
```

Implementing this algorithm gives us the following table representing the first and second S-box.

x	$T_0(x)$	$T_1(x)$	$\ldots$
$0x0$	$0x1$	$0x1000000000000$	$\ldots$
$0x1$	$0x8000000020000$	$0x800000002$	$\ldots$
$0x2$	$0x400000000$	$0x40000$	$\ldots$
$0x3$	$0x8000400000000$	$0x800040000$	$\ldots$
$0x4$	$0x400020000$	$0x40002$	$\ldots$
$0x5$	$0x8000400020001$	$0x1000800040002$	$\ldots$
$0x6$	$0x20001$	$0x1000000000002$	$\ldots$
$0x7$	$0x8000000000001$	$0x1000800000000$	$\ldots$
$0x8$	$0x20000$	$0x2$	$\ldots$
$0x9$	$0x8000400000001$	$0x1000800040000$	$\ldots$
$0xa$	$0x8000000020001$	$0x1000800000002$	$\ldots$
$0xb$	$0x400020001$	$0x1000000040002$	$\ldots$
$0xc$	$0x400000001$	$0x1000000040000$	$\ldots$
$0xd$	$0x0$	$0x0$	$\ldots$
$0xe$	$0x8000000000000$	$0x800000000$	$\ldots$
$0xf$	$0x8000400020000$	$0x800040002$	$\ldots$

9

The tables for `GIFT-128` can be generated in a similar way by looping through all 32 S-boxes instead of 16 on line 3.

2.1.2 Using `vperm`

Nowadays, most instructions set architectures support single-instruction, multiple-data processing. The idea of such an SIMD system is to work on multiple data stored in vectors at once to speed up calculations. For A64, two types of vector processing are available:

1. Advanced SIMD, known as NEON

2. Scalable Vector Extension (SVE)

We will take a look at NEON as this is the type of vector processing supported by the Cortex A-73 processor.

ARM Neon

The register file of the NEON unit is made up of 32 quad-word (128-bit) registers `V[0-31]`, each extending the standard 64-bit floating-point registers `D[0-31]`. These registers are divided into equally sized lanes on which the vector instructions operate. Valid ways to interpret for example the register `V0` are:
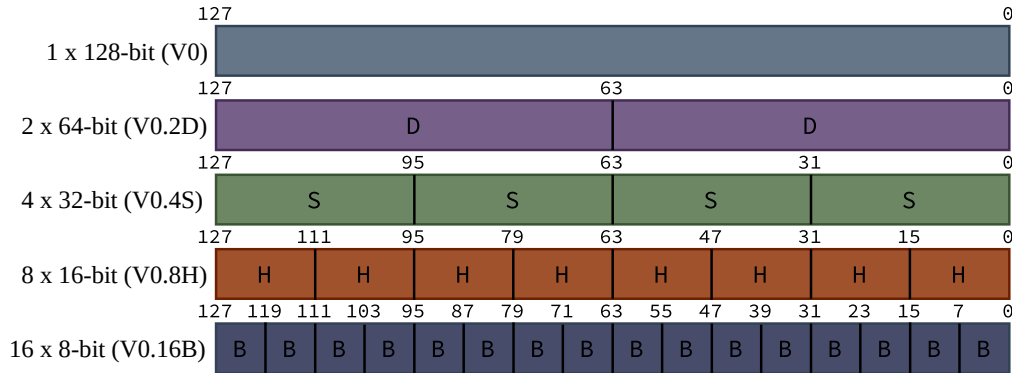

Figure 2.1: Divisions of the V register

NEON instructions interpret their operands' layouts (i.e. lane count and width) through the use of suffixes such as `.4S` or `.8H`. For instance, adding

eight 16-bit halfwords from register `V1` and `V2` together and storing the result in `V0` can be done as follows:
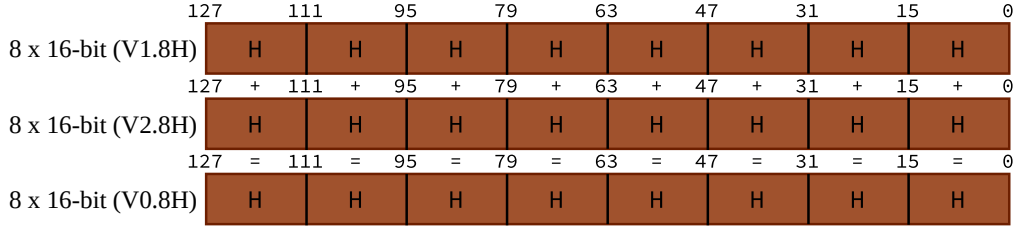
$$\text{ADD V0.8H, V1.8H, V2.8H}$$

Figure 2.2: Addition of two vector registers

The plenitude of different processing instructions allow flexible ways to further speed up algorithms having reached their optimizational limit on non-SIMD platforms. `vperm`, a general term standing for *vector permute*, is a common instruction on SIMD machines. Called `TBL` on NEON, it is used for parallel table lookups and arbitrary permutations. It takes two inputs and performs a lanewise lookup:

1. A register with lookup values

2. Two or more registers containing data

S-box lookup

This instruction can be used to implement S-box lookup of all 16 S-boxes in a single instruction. We do this by packing our 64-bit cipher state $s = s_{15}||s_{14}||\ldots||s_0$ into a vector register V_0. Because we can only operate on whole bytes, we put each 4-bit S-box into an 8-bit lane. We then put the S-box itself into register V_1 which will be used as the data register for the table lookup.

2.1.3   Bitslicing

Chapter 3

Implementation

Chapter 4

Evaluation

4.1 Limitations

4.2 Benchmarks

Acknowledgements

I want to thank ...

Bibliography

[1] Subhadeep Banik et al. "GIFT: A Small Present". In: Aug. 2017, pp. 321–345. ISBN: 978-3-319-66786-7. DOI: `10.1007/978-3-319-66787-4_16`.

[2] Ryad Benadjila et al. "Implementing Lightweight Block Ciphers on x86 Architectures". In: Selected Areas in Cryptography – SAC 2013. Ed. by Tanja Lange, Kristin Lauter, and Petr Lisoněk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 324–351. ISBN: 978-3-662-43414-7.