Efficient Implementation Strategies for Block Ciphers on ARMv8

Bachelorarbeit

Bastian Engel

April 13, 2023

Abstract

Lorem ipsum dolor 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 Notation

1.2 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 algorithmically which opens up possibilities for different tradeoffs and security concerns.

1.2.1 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.

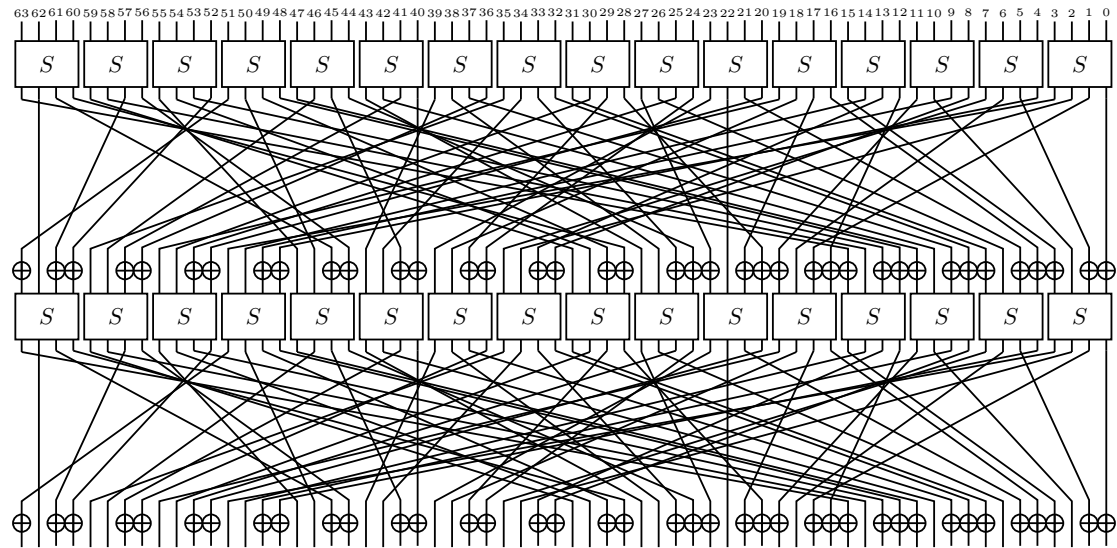Figure 1.1: Two rounds of GIFT-64

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 $b_{P(i)}$. 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 x_{i+1} after each full round is therefore given by

$$x_{i+1} = P(S(x_i)) \oplus R_i$$

Round key extraction and key schedule

Round key extraction differs for GIFT-64 and GIFT-128. Let $K_{(128)} = k_7 || k_6 || \dots || k_0$ denote the 128-bit key state.

GIFT-64 . We extract two words $U_{(16)} || V_{(16)} = k_1 || k_0$ from the key state. These are then added to round key $R_{(64)}$: $R_{4i+1} \leftarrow U_i, R_{4i} \leftarrow V_i$.

GIFT-128 . We extract two words $U_{(32)} || V_{(32)} = k_5 || k_4 || k_1 || k_0$ from the key state. These are then added to round key $R_{(128)}$: $R_{4i+2} \leftarrow U_i, R_{4i+1} \leftarrow V_i$.

In both cases, we additionally XOR a round constant $C_{(6)}$ 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 individually rotating k_1 and k_0 and rotating the new state 32 bits to the right:

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

1.2.2 Camellia

Camellia[2] is a block cipher jointly developed by NTT and Mitsubishi Electric Corporation and first published in 2001. Following AES specifications, it is able to encrypt 128-bit blocks using either 128-, 196- or 256-bit keys and claims to possess similar performance and security levels as the AES finalists.

Encryption

The encryption process has an 18-round Feistel structure for 128-bit keys and a 24-round Feistel structure for 192/256-bit key and employs key whitening to increase security. First, subkeys $kw_{t(64)}(t = 0, 1, 2, 3)$, $k_{u(64)}(u = 0, 1, \ldots, (17|23)$ and $kl_{v(64)}(v = 0, 1, 2, 3)$ are generated from the master key. Then, pre-whitening keys are applied to the plaintext $m_{(128)} = L_{(64)}||R_{(64)}$:

$$(L||R) \leftarrow (L||R) \oplus (kw_0||kw_1)$$

The next steps differ for 128-bit and 192/256-bit keys in the number of rounds:

Table 1.1: Camellia encryption

Round r	128-bit keys	192/256-bit keys
0-4	$(L||R) \leftarrow FE(L, R, k_r)$	$(L||R) \leftarrow FE(L, R, k_r)$
5	$(L||R) \leftarrow FE(L, R)$ $(L||R) \leftarrow FLL(L, R, kl_0, kl_1)$	$(L||R) \leftarrow FE(L, R)$ $(L||R) \leftarrow FLL(L, R, kl_0, kl_1)$
6-10	$(L||R) \leftarrow FE(L, R, k_r)$	$(L||R) \leftarrow FE(L, R, k_r)$
11	$(L||R) \leftarrow FE(L, R)$ $(L||R) \leftarrow FLL(L, R, kl_2, kl_3)$	$(L||R) \leftarrow FE(L, R)$ $(L||R) \leftarrow FLL(L, R, kl_2, kl_3)$
12-16	$(L||R) \leftarrow FE(L, R, k_r)$	$(L||R) \leftarrow FE(L, R, k_r)$
17		$(L||R) \leftarrow FE(L, R)$ $(L||R) \leftarrow FLL(L, R, kl_4, kl_5)$
18-23		$(L||R) \leftarrow FE(L, R, k_r)$

Finally, R and L are concatenated and XORed with the post-whitening keys to obtain the cipher text $c_{(128)}$:

$$c = (R||L) \oplus (kw_2)||kw_3)$$

Key schedule

The master key K is split into two parts $K = K_{L(128)}||K_{R(128)}$ with $K_R = 0$ for 128-bit keys. Then, two variables $K_{A(128)}, K_{B(128)}$ are generated by repeated application of the round function with key constants $\Sigma_i, i = (0, 1, \ldots, 5)$:

$K_A \leftarrow K_L \oplus K_R$
$K_A \leftarrow FE(FE(K_A, \Sigma_0 = 0xa09e667f3bcc908b), \Sigma_1 = 0xb67ae8584caa73b2)$
$K_A \leftarrow K_A \oplus K_L$
$K_A \leftarrow FE(FE(K_A, \Sigma_2 = 0xc6ef372fe94f82be), \Sigma_3 = 0x54ff53a5f1d36f1c)$
$K_B \leftarrow FE(FE(K_A, \Sigma_4 = 0x10e527fade682d1d), \Sigma_5 = 0xb05688c2b3e6c1fd)$

Subkeys are then created by rotating K_L, K_R, K_A, K_B:

Table 1.2: Subkeys for 128-bit key

Usage	Subkey	Value	Usage	Subkey	Value
Prewhitening	$kw_{0(64)}$	$(K_L \lll 0)_{L(64)}$	F(Round 9)	$k_{9(64)}$	$(K_L \lll 60)_{R(64)}$
	$kw_{1(64)}$	$(K_L \lll 0)_{R(64)}$	F(Round 10)	$k_{10(64)}$	$(K_A \lll 60)_{L(64)}$
F(Round 0)	$k_{0(64)}$	$(K_A \lll 0)_{L(64)}$	F(Round 11)	$k_{11(64)}$	$(K_A \lll 60)_{R(64)}$
F(Round 1)	$k_{1(64)}$	$(K_A \lll 0)_{R(64)}$	FL	$kl_{2(64)}$	$(K_L \lll 77)_{L(64)}$
F(Round 2)	$k_{2(64)}$	$(K_L \lll 15)_{L(64)}$	FL^{-1}	$kl_{3(64)}$	$(K_L \lll 77)_{R(64)}$
F(Round 3)	$k_{3(64)}$	$(K_L \lll 15)_{R(64)}$	F(Round 12)	$k_{12(64)}$	$(K_L \lll 94)_{L(64)}$
F(Round 4)	$k_{4(64)}$	$(K_A \lll 15)_{L(64)}$	F(Round 13)	$k_{13(64)}$	$(K_L \lll 94)_{R(64)}$
F(Round 5)	$k_{5(64)}$	$(K_A \lll 15)_{R(64)}$	F(Round 14)	$k_{14(64)}$	$(K_A \lll 94)_{L(64)}$
FL	$kl_{0(64)}$	$(K_A \lll 30)_{L(64)}$	F(Round 15)	$k_{15(64)}$	$(K_L \lll 94)_{R(64)}$
FL^{-1}	$kl_{1(64)}$	$(K_A \lll 30)_{R(64)}$	F(Round 16)	$k_{16(64)}$	$(K_A \lll 111)_{L(64)}$
F(Round 6)	$k_{6(64)}$	$(K_L \lll 45)_{L(64)}$	F(Round 17)	$k_{17(64)}$	$(K_A \lll 111)_{R(64)}$
F(Round 7)	$k_{7(64)}$	$(K_L \lll 45)_{R(64)}$	Postwhitening	$kw_{2(64)}$	$(K_A \lll 111)_{L(64)}$
F(Round 8)	$k_{8(64)}$	$(K_A \lll 45)_{L(64)}$		$kw_{3(64)}$	$(K_A \lll 111)_{R(64)}$

Subkey generation for 192/256-bit keys are generated in a similar way.

Components

We will give an overview of the main functional components of Camellia.

Feistel round function FE:

$$FE : (\mathbb{F}_2^{64})^3 \rightarrow (\mathbb{F}_2^{64})^2$$
$$(L_{(64)}, R_{(64)}, k_{(64)}) \mapsto (R \oplus F(L, k), L)$$

SP-function F:

$$F : (\mathbb{F}_2^{64})^2 \to \mathbb{F}_2^{64}$$
$$(X_{(64)}, k_{(64)}) \mapsto P(S(X \oplus k))$$

Substitution function S:

$$S : \mathbb{F}_2^{64} \to \mathbb{F}_2^{64}$$
$$l_{0(8)}||l_{1(8)}||l_{2(8)}||l_{3(8)}|| \atop l_{4(8)}||l_{5(8)}||l_{6(8)}||l_{7(8)} \mapsto {s_0(l_0)||s_1(l_1)||s_2(l_2)||s_3(l_3)|| \atop s_1(l_4)||s_2(l_5)||s_3(l_6)||s_0(l_7)} ,$$

with 8-bit S-boxes $s_0, s_1, s_2, s_3 : \mathbb{F}_2^8 \to \mathbb{F}_2^8$.

Permutation function P:

$$P : \mathbb{F}_2^{64} \to \mathbb{F}_2^{64}$$

$$\begin{pmatrix} z_7 \\ z_6 \\ z_5 \\ z_4 \\ z_3 \\ z_2 \\ z_1 \\ z_0 \end{pmatrix} \mapsto \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} z_7 \\ z_6 \\ z_5 \\ z_4 \\ z_3 \\ z_2 \\ z_1 \\ z_0 \end{pmatrix}$$

FL layer function FLL:

$$FLL : (\mathbb{F}_2^{64})^4 \to (\mathbb{F}_2^{64})^2$$
$$(X_{L(64)}, X_{R(64)}, k_{0(64)}, k_{1(64)}) \mapsto (FL(X_L, k_0), FL^{-1}(X_R, k_1))$$

FL:

$$FL : (\mathbb{F}_2^{64})^2 \to \mathbb{F}_2^{64}$$
$$(X_{L(32)}||X_{R(32)}, k_{L(32)}||k_{R(32)}) \mapsto (Y_{L(32)}||Y_{R(32)}),$$

where

$$Y_{R(32)} = ((X_L \cap k_L) \lll_1) \oplus X_R$$
$$Y_{L(32)} = (Y_R \cup k_R) \oplus X_L$$

FL^{-1}**:**

$$FL^{-1} : (\mathbb{F}_2^{64})^2 \to \mathbb{F}_2^{64}$$
$$(Y_{L(32)}||Y_{R(32)}, k_{L(32)}||k_{R(32)}) \mapsto (X_{L(32)}||X_{R(32)}),$$

where

$$X_{L(32)} = (Y_R \cup k_R) \oplus Y_L$$
$$X_{R(32)} = ((X_L \cap k_L) \lll_1) \oplus Y_R$$

1.3   The ARMv8 platform

With small devices and embedded processors becoming ever more ubiquitous and essential in areas like consumer electronics or industrial and IoT applications, the need for low-power, high-performance microprocessors has increased steadily. With more than 250 billion chips shipped, semiconductors designed by ARM power 95% of mobile devices and have found a great many applications due to their high performance and low power consumption[3]. The ODROID-N2+[4] development board we are using is based on the big.LITTLE architecture and is powered by a quad-core ARM Cortex-A73 processor and a weaker dual-core ARM Cortex-A53 for power efficiency. Both these processors are part of the eight generation of ARM designs known as ARMv8[5].

ARMv8 defines three architecture profiles for different use cases as well as dynamic execution states with corresponding instruction sets. This work will focus on the A profile running in the AArch64 state utilizing the A64 instruction set with NEON and crypto extensions.

Table 1.3: ARMv8 profiles

Profile	Description
Application (A)	Traditional use with virtual memory and privilege level support
Real-time (R)	Real-time, low-latency, deterministic embedded systems
Microcontroller (M)	Very low-power, fast-interrupt embedded systems

Table 1.4: ARMv8 execution states

Execution state	Usage	Instruction sets
AArch32	32-bit compatibility	A32/T32
AArch64	64-bit	A64

1.3.1   General architecture

ARMv8 is a RISC architecture employing simple data processing instructions operating only on registers as well as dedicated load/store instructions to transfer data from register to memory and back. This enables faster execution of individual instructions, a simpler pipeline design, predictable instruction timings and fewer addressing modes.

The A64 instruction set defines 31 64-bit general-purpose registers `X0-X30` which can also be accessed as 32-bit registers `W0-W30`. Values are loaded from and stored to memory using `LDR`/`STR`. Data processing instructions generally use explicit output registers instead of overwriting the first input register.

Table 1.5: A64 addressing modes

Addressing mode	Example	Description
Base register	`LDR W0, [X1]`	`W0 = *(X1);`
Offset	`LDR W0, [X1, #12]`	`W0 = *(X1 + 12);`
Pre-indexing	`LDR W0, [X1, #12]!`	`X1 += 12; W0 = *(X1);`
Post-indexing	`LDR W0, [X1], #12`	`W0 = *(X1); X1 += 12;`

1.3.2   NEON

ARMv8 supports single-instruction, multiple-data (SIMD) processing. These systems allow the programmer to store multiple pieces of data in a vector and work on them in parallel to speed up calculations. The A64 instruction set defines two possible SIMD implementations:

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-A73 processor.

The register file of the NEON unit is made up of 32 quad-word (128-bit) registers `V0-V31`, each extending the standard 64-bit floating-point registers `D0-31`. These registers are divided into equally sized lanes on which vector instructions operate. Figure 1.2 shows valid ways to interpret the register `V0`.
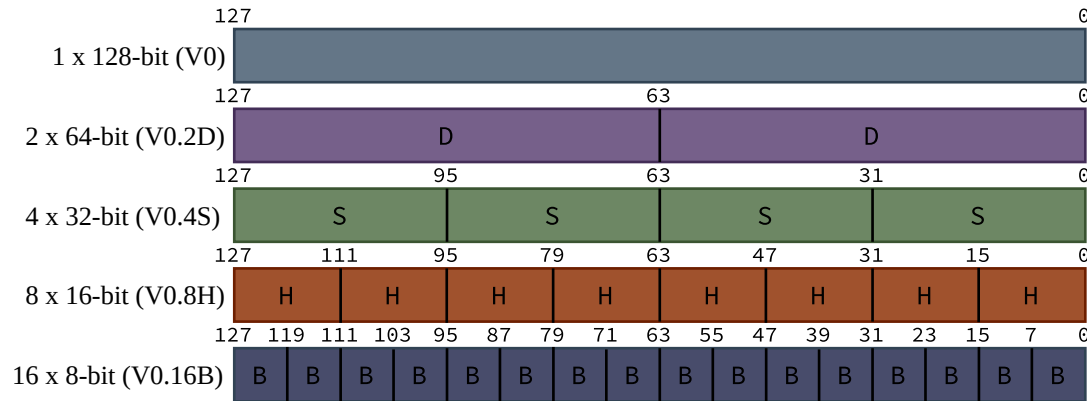
Figure 1.2: Divisions of the V0 register

NEON instructions interpret their operands' layouts (i.e. lane count and width) through the use of suffixes such as `.4S` or `.8H`. Adding eight 16-bit halfwords stored in `V1` and `V2` can be done as follows:

```
ADD V0.8H, V1.8H, V2.8H
```
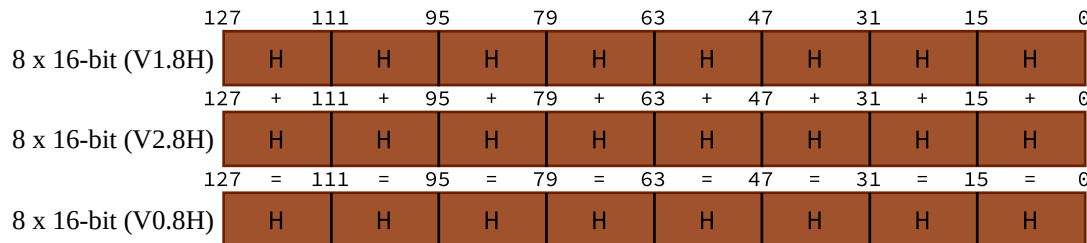

Figure 1.3: Addition of two vector registers

1.3.3 NEON Intrinsics

The header file `<arm_neon.h>` provides ARM-specific data and function definitions including vector data types and C functions for working with these vectors. These functions are known as NEON intrinsics [6] and give the programmer a

high-level interface to most NEON instructions. Major advantages of this approach include the ease of development as the compiler takes over register allocation and load/store operations as well as performance benefits through compiler optimizations.

Standard vector data types have the format `uintnxm_t` with lane width n in bits and and lane count m. Array types of the format `uintnxmxc_t`, $c \in \{2, 3, 4\}$ are also defined which are used in operations requiring multiple parameters like `TBL` or pairwise load/stores. Intrinsics include the operation name and lane data format as well as an optional `q` suffix to indicate operation on a 128-bit register. Multiplying eight pairs of 16-bit numbers `a,b` for example can be done via the following:

$$\texttt{uint16x8\_t result = vmulq\_u16(a, b);}$$

In this case, the compiler allocates vector registers for `a`, `b` and `result` and assembles the intrinsic to `MUL Vr.8H, Va.8H, Vb.8H`. Necessary loads and stores for the result and parameters are also handled automatically. Of special interest to us are the following intrinsics, each existing in different variants with different lane widths and also array types:

Table 1.6: Common NEON intrinsics

Intrinsic		Summary
uint64_t	vgetq_lane_u64(void)	Extract a single lane
void	vsetq_lane_u64(uint64_t)	Insert a single lane
uint64x2_t	vdupq_n_u64(uint64_t)	Initialize all lanes to same value
void	vst1q_u64(uint64_t*, uint64x2_t)	Store from register to memory
uint64x2_t	vld1q_u64(uint64_t*, uint64x2_t)	Load from memory to register
uint8x16_t	veorq_u8(uint8x16_t, uint8x16_t)	bitwise XOR
uint8x16_t	vandq_u8(uint8x16_t, uint8x16_t)	bitwise AND
uint8x16_t	vorrq_u8(uint8x16_t, uint8x16_t)	bitwise OR
uint8x16_t	vmvnq_u8(uint8x16_t)	bitwise NOT
uint8x16_t	vqtbl2q_u8(uint8x16_t, uint8x16_t)	permutation (TBL)

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 **GIFT**

Three implementation strategies for substitution-permutation networks are introduced by [7]:

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

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`

The plenitude of different processing instructions introduced by NEON1.3.2 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 arbitary permutations. It takes two inputs to perform a lanewise lookup:

1. A register with lookup values

2. One 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 which neatly fits into the 128-bit registers. We then put the S-box itself into register V_1 which will be used as the data register for the table lookup.

The confusion layer can now be performed through one `TBL` instruction:

<div align="center">

`TBL V0.16B, V1.16B, V0.16B`

</div>
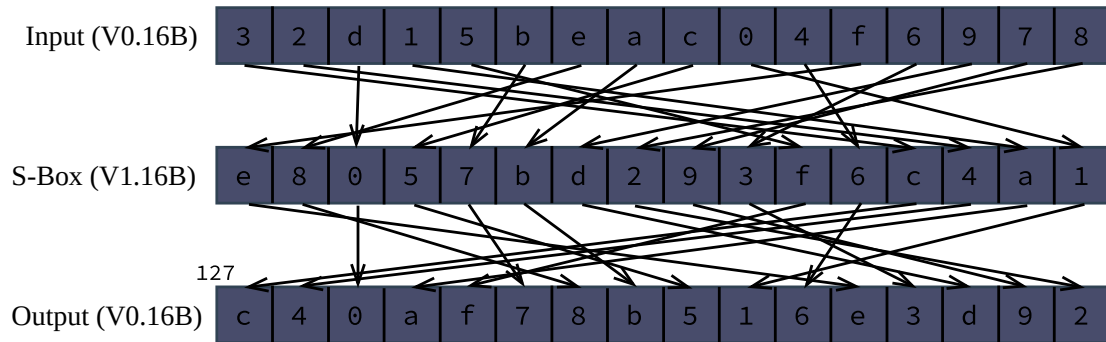
Figure 2.1: Performing the S-Box lookup in parallel

2.1.3   Bitslicing

Bitslicing refers to the technique of splitting up n bits into m slices to achieve a more efficient representation to operate on. The structure of GIFT naturally offers possibilities for bitslicing. We split the cipher state bits $b_{63}b_{62}\ldots b_0$ into four slices $S_i, i \in \{0, 1, 2, 3\}$ such that the i-th slice contains all i-th bits of the individual S-boxes. This is equivalent to transposing the bit matrix.

$$S = \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} b_{60}b_{56}b_{52}\ldots b_0 \\ b_{61}b_{57}b_{53}\ldots b_1 \\ b_{62}b_{58}b_{54}\ldots b_2 \\ b_{63}b_{59}b_{55}\ldots b_3 \end{bmatrix}$$

Parallel S-Boxes

This representation offers multiple advantages. We first note that computation of the S-box can be executed in parallel, similar to the `vperm` technique above. This can be done by finding an algorithmic way to apply the S-box which has already been proposed by the original `GIFT` authors:

$$S_1 \leftarrow S_1 \oplus (S_0 \wedge S_2)$$
$$t \leftarrow S_0 \oplus (S_1 \wedge S_3)$$
$$S_2 \leftarrow S_2 \oplus (t \vee S_1)$$
$$S_0 \leftarrow S_3 \oplus S_2$$
$$S_1 \leftarrow S_1 \oplus S_0$$
$$S_0 \leftarrow \neg S_0$$
$$S_2 \leftarrow S_2 \oplus (t \wedge S_1)$$
$$S_3 \leftarrow t$$

This is very efficient as it only requires six XOR-, three AND and one OR operation.

An important property of the permutation is the fact that bits always stay in their slice. This means we can decompose the permutation P into four permutations $P_i, i \in \{0, 1, 2, 3\}$ and apply these permutations separately to each slice. One possible way to implement a permutation P_i in software is to mask off all bits individually, shift them to their correct position and OR them together:

$$P_i(S_i) = \bigvee_{k=0}^{15} (S_i \wedge m_i) \lll s_i$$

This approach requires 47 operations, meaning all four permutations require over 150 operations which would present a major bottleneck to the round function. We can improve on this by working on multiple message blocks at once and using the aforementioned `vperm` instruction to implement the bit shuffling. We then need only four instructions for the complete diffusion layer.

Using `vperm` for slice permutation

We cannot use the `TBL` instruction directly as we need to shuffle individual bits, but the smallest data we can operate on are bytes. We therefore encrypt $8n$ messages at once which allows us to create bytewise groupings. These messages are put into $4m$ registers with register R_{4i} containing S_0, register R_{4i+1} containing S_1 and so forth. With block size BS and register size RS, the following must hold:

$$8n \cdot BS = 4m \cdot RS$$

In the case of `GIFT-64` with $BS = 64$ and ARM NEON with $RS = 128$, we get

$$8n \cdot 64 = 4m \cdot 128 \Leftrightarrow n = m$$

$n = m = 1$ would be a valid choice which yields eight messages divided into four registers. We choose $n = m = 2$ so we can directly utilize the algorithm for bit packing presented by the original GIFT authors, although it is simple to adapt this algorithm to only four registers and eight messages by adjusting the `SWAPMOVE` shift and mask values.

Packing the data into bitslice format

Let $a, b, \ldots, p$ be sixteen messages of length 64 with subscripts denoting individual bits. We first put these messages into eight SIMD registers $V_0, V_1, \ldots, V_7$:

$$
\begin{aligned}
V_0 &= b||a & V_4 &= j||i \\
V_1 &= d||c & V_5 &= l||k \\
V_2 &= f||e & V_6 &= n||m \\
V_3 &= h||g & V_7 &= p||o
\end{aligned}
$$

We then use the `SWAPMOVE` technique to bring the data into bitslice format. This operation operates on two registers A, B using mask M and shift value N. It swaps bits in A masked by $(M \ll N)$ with bits in B masked by M in using only three XOR-, one AND- and two shift operations.

$$
\begin{aligned}
&\text{SWAPMOVE}(A, B, M, N): \\
&\quad T = ((A \gg N) \oplus B) \wedge M \\
&\quad B = B \oplus T \\
&\quad A = A \oplus (T \ll N)
\end{aligned}
$$

One caveat of this approach is the fact that NEON registers cannot be shifted in their entirety due to the fact bits are not able to cross lanes. This leads to the problem of being able to shift at most two lanes of 64 bits at once. We thus need to implement the `shr(V,n)` and `shl(V,n)` operations on our own. This can be done by first extracting the 64-bit lanes a, b out of $V = b||a$, shifting the lanes individually and finally shifting and ORing the crossing bits back into the other lane.

$$\begin{aligned}
\text{shl}(V, n) : \\
a, b \quad &= V[0], V[1] \\
c \quad &= (a \gg (64 - n)) \\
V[0] \quad &= (a \ll n) \\
V[1] \quad &= (b \ll n) \vee c
\end{aligned}$$

The following operations group all i-th bits of the messages $a, c, \ldots, o$ into bytes and puts these into the lower half of the registers $V_{i \bmod 8}$. The same is done for messages $b, d, \ldots, p$, only differing in that the bytes are put into the upper half of the registers.

$$\text{SWAPMOVE}(V_0, V_1, 0x5555\ldots55, 1) \quad \text{SWAPMOVE}(V_4, V_5, 0x5555\ldots55, 1)$$
$$\text{SWAPMOVE}(V_2, V_3, 0x5555\ldots55, 1) \quad \text{SWAPMOVE}(V_6, V_7, 0x5555\ldots55, 1)$$
$$\text{SWAPMOVE}(V_0, V_2, 0x3333\ldots33, 2) \quad \text{SWAPMOVE}(V_4, V_6, 0x3333\ldots33, 2)$$
$$\text{SWAPMOVE}(V_1, V_3, 0x3333\ldots33, 2) \quad \text{SWAPMOVE}(V_5, V_7, 0x3333\ldots33, 2)$$
$$\text{SWAPMOVE}(V_0, V_4, 0x0f0f\ldots0f, 4) \quad \text{SWAPMOVE}(V_1, V_5, 0x0f0f\ldots0f, 4)$$
$$\text{SWAPMOVE}(V_2, V_6, 0x0f0f\ldots0f, 4) \quad \text{SWAPMOVE}(V_3, V_7, 0x0f0f\ldots0f, 4)$$

With $Ax = o_x m_x k_x j_x g_x e_x c_x a_x$ and $Bx = p_x n_x l_x i_x h_x f_x d_x b_x$ denoting byte groups, our data now has the following permutation-friendly format:

n	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
V_0	$B56$	$B48$	$B40$	$B32$	$B24$	$B16$	$B8$	$B0$	$A56$	$A48$	$A40$	$A32$	$A24$	$A16$	$A8$	$A0$
V_1	$B57$	$B49$	$B41$	$B33$	$B25$	$B17$	$B9$	$B1$	$A57$	$A49$	$A41$	$A33$	$A25$	$A17$	$A9$	$A1$
V_2	$B58$	$B50$	$B42$	$B34$	$B26$	$B18$	$B10$	$B2$	$A58$	$A50$	$A42$	$A34$	$A26$	$A18$	$A10$	$A2$
V_3	$B59$	$B51$	$B43$	$B35$	$B27$	$B19$	$B11$	$B3$	$A59$	$A51$	$A43$	$A35$	$A27$	$A19$	$A11$	$A3$
V_4	$B60$	$B52$	$B44$	$B36$	$B28$	$B20$	$B12$	$B4$	$A60$	$A52$	$A44$	$A36$	$A28$	$A20$	$A12$	$A4$
V_5	$B61$	$B53$	$B45$	$B37$	$B29$	$B21$	$B13$	$B5$	$A61$	$A53$	$A45$	$A37$	$A29$	$A21$	$A13$	$A5$
V_6	$B62$	$B54$	$B46$	$B38$	$B30$	$B22$	$B14$	$B6$	$A62$	$A54$	$A46$	$A38$	$A30$	$A22$	$A14$	$A6$
V_7	$B63$	$B55$	$B47$	$B39$	$B31$	$B23$	$B15$	$B7$	$A63$	$A55$	$A47$	$A39$	$A31$	$A23$	$A15$	$A7$

Although this would already work, we prefer to have only bits of the same messages in each register - otherwise the permutation would need to operate on two

source registers with the added requirement of storing the pre-permutation values for the first four registers, slowing down the round function through superfluous load/stores. This transformation is trivial by use of `TBL` with two data source operands. The final data format we operate on is as follows:

n	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
V_0	$A60$	$A56$	$A52$	$A48$	$A44$	$A40$	$A36$	$A32$	$A28$	$A24$	$A20$	$A16$	$A12$	$A8$	$A4$	$A0$
V_1	$A61$	$A57$	$A53$	$A49$	$A45$	$A41$	$A37$	$A33$	$A29$	$A25$	$A21$	$A17$	$A13$	$A9$	$A5$	$A1$
V_2	$A62$	$A58$	$A54$	$A50$	$A46$	$A42$	$A38$	$A34$	$A30$	$A26$	$A22$	$A18$	$A14$	$A10$	$A6$	$A2$
V_3	$A63$	$A59$	$A55$	$A51$	$A47$	$A43$	$A39$	$A35$	$A31$	$A27$	$A23$	$A19$	$A15$	$A11$	$A7$	$A3$
V_4	$B60$	$B56$	$B52$	$B48$	$B44$	$B40$	$B36$	$B32$	$B28$	$B24$	$B20$	$B16$	$B12$	$B8$	$B4$	$B0$
V_5	$B61$	$B57$	$B53$	$B49$	$B45$	$B41$	$B37$	$B33$	$B29$	$B25$	$B21$	$B17$	$B13$	$B9$	$B5$	$B1$
V_6	$B62$	$B58$	$B54$	$B50$	$B46$	$B42$	$B38$	$B34$	$B30$	$B26$	$B22$	$B18$	$B14$	$B10$	$B6$	$B2$
V_7	$B63$	$B59$	$B55$	$B51$	$B47$	$B43$	$B39$	$B35$	$B31$	$B27$	$B23$	$B19$	$B15$	$B11$	$B7$	$B3$

We can now create permutation tables using the specification of the individual slice permutations P_i which are then applied to V_i and V_{i+4} respectively:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P_0(j)$	0	12	8	4	1	13	9	5	2	14	10	6	3	15	11	7
$P_1(j)$	4	0	12	8	5	1	13	9	6	2	14	10	7	3	15	11
$P_2(j)$	8	4	0	12	9	5	1	13	10	6	2	14	11	7	3	15
$P_3(j)$	12	8	4	0	13	9	5	1	14	10	6	2	15	11	7	3

One thing to take note of is the original permutation values only show where a given byte should land, not which byte belongs to a certain position - i.e. for P_0, byte 1 should land in position 12, but the byte belonging to position 1 is byte 4. Because `TBL` works in the latter way, we have to do some trivial rearrangements.

Assuming the correct permutation values are put into registers V_8, V_9, V_{10}, V_{11}, this now allows us to compute the permutation layer for all 16 blocks in only eight permutation instructions.

```
TBL V0, V0, V8      TBL V1, V1, V9
TBL V4, V4, V8      TBL V5, V5, V9
TBL V2, V2, V10     TBL V3, V3, V11
TBL V6, V6, V10     TBL V7, V7, V11
```

Round key function

In contrast to packing and unpacking of data which is only done once in the beginning and end, a round key is derived for every round, so the round key derivation function needs to be as fast as possible. A simple but naive approach

for one round would be to generate a single round key, copy it 15 times and pack the resulting registers similar to how we proceed with the messages. Due to the cost of packing the messages, this is prohibitivly expensive. Because we know where each byte group ends up after packing, we can directly XOR the round key bits to the correct position. Extending these bits to bytes can then be done simply by repeatedly shifting and ORing the registers together.

2.2 Strategies for Camellia

2.2.1 Platform-independent techniques

The original paper proposes various platform-independent ways to implement Camellia efficiently. These can be summarized as follows:

Subkey ordering: Compute all K_L-dependent subkeys, then all K_A-dependent subkeys etc. to save registers

XOR cancellation property in key schedule:

$$K_A \leftarrow K_L \oplus K_R$$
$$K_A \leftarrow FE(FE(K_A, \Sigma_0 = 0xa09e667f3bcc908b), \Sigma_1 = 0xb67ae8584caa73b2)$$
$$K_A \leftarrow K_A \oplus K_L$$

causes cancellations, allowing us to eliminate some operations by replacing it with the following:

$$K_{A_L} \leftarrow F(K_{L_R} \oplus F(K_{L_L}, \Sigma_0))$$
$$K_{A_R} \leftarrow F(K_{L_L} \oplus \Sigma_1)$$

kl_5, kl_6 **generation from** k_{11}, k_{12}: since $(kl_5, kl_6) = (k_{11}, k_{12} \lll_{32})$, using word-oriented rotation saves a few instructions

Chapter 3

Implementation

Implementations in the C programming language for the presented strategies can be found in Appendix A. Although directly writing Assembler code could result in a small performance benefit, this generally increases the work necessary by an order of magnitude for only limited results. Instruction-level optimization and in particular register allocation is left to the compiler. Relying on the compiler mandates a closer study of the generated, optimized assembler. All source files were compiled using clang version 15.0.7 and optimization level O3.

3.1 Pipelining

Understanding certain choices requires an understanding of the Cortex-A73 instruction pipeline[8]. Being a superscalar processor, it is able to execute more than one instruction per clock cycle by dispatching instructions to different execution units working in parallel.
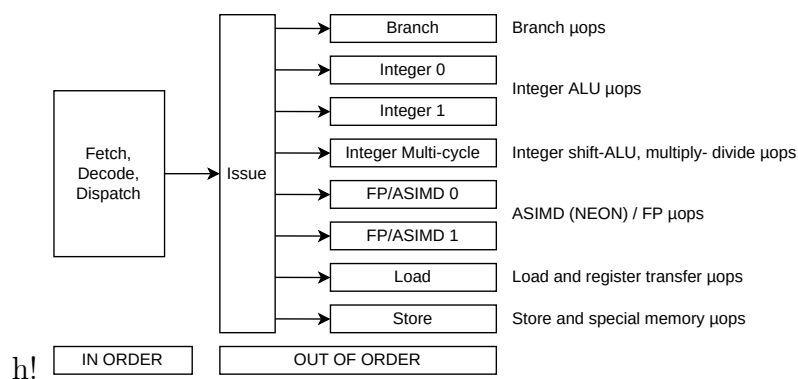

Figure 3.1: High-level overview of the Cortex-A72 instruction pipeline

The processor might for example store a calculation result, load a necessary value from memory and execute two SIMD operations at once, all in the same clock cycle. Modern compilers take advantage of this fact by reordering instructions such that all pipeline execution units stay as busy as possible and do not stall while having to wait for new instructions to be dispatched. A more thorough analysis will be presented for the bitsliced strategy of `GIFT`, but all implementations are heavily reliant on function inlining, instruction reordering and loop unrolling.

3.2 GIFT

3.2.1 Table-based

This is the simplest strategy to implement. Indeed, its biggest advantage lies in its portability to other platforms without relying on specific features or extensions. The cipher state is stored as a 64-bit word and one round consists in extracting the 4-bit S-boxes, looking up table values, collecting these in an accumulator and finally adding the round key.

```
uint64_t gift_64_table_subperm(const uint64_t cipher_state)
{
        uint64_t new_cipher_state = 0;

        for (size_t i = 0; i < 16; i++) {
                int nibble = (cipher_state >> (i * 4)) & 0xf;
                new_cipher_state ^= tables[i][nibble];
        }

        return new_cipher_state;
}
```

```
uint64_t gift_64_table_encrypt(const uint64_t m,
                               const uint64_t rks[restrict ROUNDS_GIFT_64])
{
        uint64_t c = m;

        // round loop
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
                c = gift_64_table_subperm(c);
                c ^= rks[round];
        }

        return c;
}
```

3.2.2 Using `vperm`

Implementation of the substitution layer requires the use of a single vector intrinsic. This mandates the packing of data into a vector register which in turn is

disadvantageous to the permutation layer as we need to extract single bits. Packing and unpacking is nothing more than filling 8-bit vector lanes with 4-bit S-boxes and vice versa.

```
uint8x16_t gift_64_vec_sbox_subcells(const uint8x16_t cipher_state)
{
        return vqtbl1q_u8(sbox_vec, cipher_state);
}
```

```
uint8x16_t gift_64_vec_sbox_permute(const uint8x16_t cipher_state)
{
        // collect individual bits into 64-bit register
        uint64_t new_cipher_state = 0UL;
        uint64_t boxes[2];
        vst1q_u64(boxes, cipher_state);

        for (size_t box = 0; box < 16; box++) {
                for (size_t i = 0; i < 4; i++) {
                        const int bit = (boxes[box / 8] >> ((box % 8) * 8 + i)) & 0x1;
                        new_cipher_state |= (uint64_t)bit << perm_64[box * 4 + i];
                }
        }

        return gift_64_vec_sbox_bits_pack(new_cipher_state);
}
```

```
uint64_t gift_64_vec_sbox_encrypt(const uint64_t m,
                                  const uint8x16_t rks[restrict ROUNDS_GIFT_64])
{
        // pack into vector register
        uint8x16_t c = gift_64_vec_sbox_bits_pack(m);

        // round loop
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
                c = gift_64_vec_sbox_subcells(c);
                c = gift_64_vec_sbox_permute(c);
                c = veorq_u8(c, rks[round]);
        }

        // unpack
        return gift_64_vec_sbox_bits_unpack(c);
}
```

3.2.3 Bitslicing

We will examine the round function in closer detail and compare the source code with the generated assembly.

Listing 3.1: Round function src

```
1  for (int round = 0; round < ROUNDS_GIFT_64; round++) {
2      gift_64_vec_sliced_subcells(s);
3      gift_64_vec_sliced_permute(s);
4
5      // round key addition
6      s[0].val[0] = veorq_u8(s[0].val[0], rks[round][0].val[0]);
7      s[0].val[1] = veorq_u8(s[0].val[1], rks[round][0].val[1]);
8      s[0].val[2] = veorq_u8(s[0].val[2], rks[round][0].val[2]);
9      s[0].val[3] = veorq_u8(s[0].val[3], rks[round][0].val[3]);
10     s[1].val[0] = veorq_u8(s[1].val[0], rks[round][1].val[0]);
11     s[1].val[1] = veorq_u8(s[1].val[1], rks[round][1].val[1]);
12     s[1].val[2] = veorq_u8(s[1].val[2], rks[round][1].val[2]);
13     s[1].val[3] = veorq_u8(s[1].val[3], rks[round][1].val[3]);
14 }
```

Listing 3.2: Round function asm

```
1   and   v20.16b, v17.16b, v6.16b       24  tbl   v18.16b, {v6.16b}, v2.16b
2   add   x9, x20, x8                    25  tbl   v19.16b, {v21.16b}, v3.16b
3   eor   v20.16b, v20.16b, v16.16b      26  tbl   v21.16b, {v4.16b}, v2.16b
4   add   x8, x8, #0x80                  27  ldp   q6, q4, [x9, #-112]
5   and   v21.16b, v20.16b, v19.16b      28  mvn   v16.16b, v16.16b
6   cmp   x8, #0xe70                     29  tbl   v16.16b, {v16.16b}, v0.16b
7   eor   v21.16b, v21.16b, v6.16b       30  tbl   v17.16b, {v17.16b}, v1.16b
8   orr   v6.16b, v6.16b, v16.16b        31  mvn   v5.16b, v5.16b
9   eor   v6.16b, v6.16b, v17.16b        32  tbl   v5.16b, {v5.16b}, v0.16b
10  eor   v16.16b, v19.16b, v6.16b       33  ldp   q22, q23, [x9, #-80]
11  eor   v17.16b, v16.16b, v20.16b      34  eor   v6.16b, v6.16b, v16.16b
12  and   v19.16b, v21.16b, v17.16b      35  eor   v16.16b, v4.16b, v17.16b
13  eor   v6.16b, v19.16b, v6.16b        36  tbl   v7.16b, {v7.16b}, v1.16b
14  and   v19.16b, v7.16b, v4.16b        37  eor   v17.16b, v22.16b, v18.16b
15  eor   v19.16b, v19.16b, v5.16b       38  tbl   v20.16b, {v20.16b}, v3.16b
16  and   v20.16b, v19.16b, v18.16b      39  ldp   q4, q18, [x9, #-48]
17  eor   v20.16b, v20.16b, v4.16b       40  eor   v19.16b, v23.16b, v19.16b
18  orr   v4.16b, v4.16b, v5.16b         41  eor   v4.16b, v4.16b, v5.16b
19  eor   v4.16b, v4.16b, v7.16b         42  ldp   q22, q23, [x9, #-16]
20  eor   v5.16b, v18.16b, v4.16b        43  eor   v5.16b, v18.16b, v7.16b
21  eor   v7.16b, v5.16b, v19.16b        44  eor   v7.16b, v22.16b, v21.16b
22  and   v18.16b, v20.16b, v7.16b       45  eor   v18.16b, v23.16b, v20.16b
23  eor   v4.16b, v18.16b, v4.16b        46  b.ne  197e0
```

Without needing to dive too deep into details, it is obvious to see the compiler has inlined the two function calls to subcells and permute with lines 1 to 23 originating from the subcells and lines 24-46 from the permute and round key addition functions. It has chosen not to unroll the loop, but has moved the loop counter increment as well as the condition check in between the subcells instructions to line 4 and 6. In addition, the loop counter serves a second purpose as an offset register and is therefore incremented by 0x80=128 instead of just 1.

Permutation (tbl) and round key addition (eor) instructions are interleaved. The compiler recognizes data dependencies and can therefore proceed with round key addition immediately after a slice has been permuted without needing to wait for all permutations to finish. This is only logical considering the inner workings of the instruction pipeline: by interleaving NEON with regular logic and load instruc-

tions, the execution units are filled more evenly and pipeline stalls are prevented which speeds up computation.

Round keys are loaded from memory a few instructions before they are needed; assuming all round keys are stored in the L1 cache, loading a floating-point/vector register takes 5 cycles. After the load has been dispatched to the load execution unit in line 27, the processor happily continues chugging away at the instruction stream by issuing `tbl` and `mvn` μops to other execution units.

These kinds of optimizations are pervasive when programming using higher level languages like C and modern-day compilers more often than not outperform handcrafted assembly.

Chapter 4

Evaluation

In this chapter, we will evaluate the strategies through performance measurements and discuss advantages, disadvantages and possible use cases.

4.1 Benchmarks

Performance measurements were taken for each strategy as well as for naive reference implementations and are presented through latency lat in cycles per byte (c/B) as well as constant throughput thr in MiB/s of the entire encryption strategy. Round key derivation is measured separately. Measurements of all individual components like packing or permuting have to be viewed as upper bounds due to the aforementioned inlining, instruction reordering and pipelining (TODO REF HERE) taking place in the actual encryption function.

The AArch64 defines system registers in addition to general-purpose registers which are used for system configuration and monitoring. One of these registers is the performance monitor cycle count register `PMCCNTR` which counts processor clock cycles. Access from userspace is disabled by default and can be activated through a custom Linux kernel module by setting `PMUSERENR.EN` to 1. To minimize interference and because the cycle count register is core-local, we isolate and utilize one Cortex-A53 and Cortex-A73 core from the rest of the system for exclusive benchmarking purposes respectively by use of the `isolcpus` kernel command line parameter and `taskset` command utility.

4.1.1 GIFT

4.1.2 Camellia

Table 4.1: Benchmarks for GIFT

Strategy	Component	Cortex-A53		Cortex-A73	
		lat (c/B)	*thr* (MiB/s)	*lat* (c/B)	*thr* (MiB/s)
Naive GIFT-64	`round_keys` `encrypt`	223.54 1367.66	1.22	190.34 830.49	2.33
	`subcells` `permute`	4.64 36.68		3.13 21.09	
Naive GIFT-128	`round_keys` `encrypt`	182.91 3532.52	0.51	167.16 2615.89	0.79
	`subcells` `permute`	6.91 82.68		2.73 61.27	
Table-driven	`round_keys` `encrypt`	223.81 122.11	13.59	190.11 119.62	16.09
	`subperm`	5.15		4.47	
`vperm` S-box	`round_keys` `encrypt`	308.12 658.95	2.53	270.80 492.29	3.93
	`subcells` `permute` `pack` `unpack`	1.63 22.71 7.51 8.76		1.13 15.24 2.94 5.89	
Bitsliced	`round_keys` `encrypt`	10.13 16.22	111.89	7.66 13.51	154.21
	`subcells` `permute` `pack` `unpack`	0.41 0.39 1.48 1.49		0.18 0.09 1.01 1.01	

Table 4.2: Benchmarks for Camellia

Strategy	Component	Cortex-A53		Cortex-A73	
		lat (c/B)	*thr* (MiB/s)	*lat* (c/B)	*thr* (MiB/s)
Naive GIFT-64	`round_keys` `encrypt`	223.54 1367.66	1.22	190.34 830.49	2.33
	`subcells` `permute`	4.64 36.68		3.13 21.09	
Naive GIFT-128	`round_keys` `encrypt`	182.91 3532.52	0.51	167.16 2615.89	0.79
	`subcells` `permute`	6.91 82.68		2.73 61.27	
Table-driven	`round_keys` `encrypt`	223.81 122.11	13.59	190.11 119.62	16.09
	`subperm`	5.15		4.47	
`vperm` S-box	`round_keys` `encrypt`	308.12 658.95	2.53	270.80 492.29	3.93
	`subcells` `permute` `pack` `unpack`	1.63 22.71 7.51 8.76		1.13 15.24 2.94 5.89	
Bitsliced	`round_keys` `encrypt`	10.13 16.22	111.89	7.66 13.51	154.21
	`subcells` `permute` `pack` `unpack`	0.41 0.39 1.48 1.49		0.18 0.09 1.01 1.01	

Acknowledgements

I want to thank ...

Appendix A

C implementations

A.1 Implementations for SPN

A.1.1 Table-based

Listing A.1: `gift_table.h`

```
1  #ifndef GIFT_TABLE_H
2  #define GIFT_TABLE_H
3
4  #include <stdint.h>
5
6  #define ROUNDS_GIFT_64 28
7
8  void gift_64_table_generate_round_keys(uint64_t rks[restrict ROUNDS_GIFT_64],
9                                         const uint64_t key[restrict 2]);
10
11 uint64_t gift_64_table_subperm(const uint64_t cipher_state);
12
13 // can only encrypt using table technique!
14 uint64_t gift_64_table_encrypt(const uint64_t m,
15                                const uint64_t rks[restrict ROUNDS_GIFT_64]);
16
17 #endif
```

Listing A.2: `gift_table.c`

```
1  #include "gift_table.h"
2
3  #include <stdlib.h>
4  #include <string.h>
5
6  static const int round_const[] = {
7          // rounds 0-15
8          0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B,
9          0x37, 0x2F, 0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E,
10         // rounds 16-31
11         0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30,
12         0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E, 0x1C, 0x38,
13         // rounds 32-47
```

```
14          0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A,
15          0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04
16 };
17
18 static const uint64_t tables[16][16] = {
19          { 0x0000000000000001UL, 0x0008000000020000UL, 0x0000000400000000UL, 0
               x0008000400000000UL, 0x0000000400020000UL, 0x0008000400020001UL, 0
               x0000000000020001UL, 0x0008000000000001UL, 0x0000000000020000UL, 0
               x0008000400000001UL, 0x0008000000020001UL, 0x0000000400020001UL, 0
               x0000000400000001UL, 0x0000000000000000UL, 0x0008000000000000UL, 0
               x0008000400020000UL },
20          { 0x0001000000000000UL, 0x0000000800000002UL, 0x0000000000040000UL, 0
               x0000000800040000UL, 0x0000000000040002UL, 0x0001000800040002UL, 0
               x0001000000000002UL, 0x0001000800000000UL, 0x0000000000000002UL, 0
               x0001000800040000UL, 0x0001000800000002UL, 0x0001000000040002UL, 0
               x0001000000040000UL, 0x0000000000000000UL, 0x0000000800000000UL, 0
               x0000000800040002UL },
21          { 0x0000000100000000UL, 0x0002000000080000UL, 0x0000000000000004UL, 0
               x0000000000080004UL, 0x0002000000000004UL, 0x0002000100080004UL, 0
               x0002000100000000UL, 0x0000000100080000UL, 0x0002000000000000UL, 0
               x0000000100080004UL, 0x0002000100080000UL, 0x0002000100000004UL, 0
               x0000000100000004UL, 0x0000000000000000UL, 0x0000000000080000UL, 0
               x0002000000080004UL },
22          { 0x0000000000010000UL, 0x0000000200000008UL, 0x0004000000000000UL, 0
               x0004000000000008UL, 0x0004000200000000UL, 0x0004000200010008UL, 0
               x0000000200010000UL, 0x0000000000010008UL, 0x0000000200000000UL, 0
               x0004000000010008UL, 0x0000000200010008UL, 0x0004000200010000UL, 0
               x0004000000010000UL, 0x0000000000000000UL, 0x0000000000000008UL, 0
               x0004000200000008UL },
23          { 0x0000000000000010UL, 0x0080000000200000UL, 0x0000000400000000UL, 0
               x0080000400000000UL, 0x0000000400020000UL, 0x0080000400020010UL, 0
               x0000000000200010UL, 0x0080000000000010UL, 0x0000000000200000UL, 0
               x0080000400000010UL, 0x0080000000200010UL, 0x0000000400200010UL, 0
               x0000000400000010UL, 0x0000000000000000UL, 0x0080000000000000UL, 0
               x0080000400200000UL },
24          { 0x0010000000000000UL, 0x0000000800000020UL, 0x0000000000400000UL, 0
               x0000000800400000UL, 0x0000000000400020UL, 0x0010000800400020UL, 0
               x0010000000000020UL, 0x0010000800000000UL, 0x0000000000000020UL, 0
               x0010000800400000UL, 0x0010000800000020UL, 0x0010000000400020UL, 0
               x0010000000400000UL, 0x0000000000000000UL, 0x0000000800000000UL, 0
               x0000000800400020UL },
25          { 0x0000000100000000UL, 0x0020000000800000UL, 0x0000000000000040UL, 0
               x0000000000800040UL, 0x0020000000000040UL, 0x0020000100080040UL, 0
               x0020000100000000UL, 0x0000000100080000UL, 0x0020000000000000UL, 0
               x0000000100080040UL, 0x0020000100080000UL, 0x0020000100000040UL, 0
               x0000000100000040UL, 0x0000000000000000UL, 0x0000000000800000UL, 0
               x0020000000800040UL },
26          { 0x0000000000100000UL, 0x0000000200000080UL, 0x0040000000000000UL, 0
               x0040000000000080UL, 0x0040000200000000UL, 0x0040000200100080UL, 0
               x0000000200100000UL, 0x0000000000100080UL, 0x0000000200000000UL, 0
               x0040000000100080UL, 0x0000000200100080UL, 0x0040000200100000UL, 0
               x0040000000100000UL, 0x0000000000000000UL, 0x0000000000000080UL, 0
               x0040000200000080UL },
27          { 0x0000000000000100UL, 0x0800000002000000UL, 0x0000004000000000UL, 0
               x0800040000000000UL, 0x0000004002000000UL, 0x0800040002000100UL, 0
               x0000000002000100UL, 0x0800000000000100UL, 0x0000000002000000UL, 0
               x0800040000000100UL, 0x0800000002000100UL, 0x0000004002000100UL, 0
               x0000040000000100UL, 0x0000000000000000UL, 0x0800000000000000UL, 0
               x0800040002000000UL },
28          { 0x0100000000000000UL, 0x0000080000000200UL, 0x0000000004000000UL, 0
               x0000080004000000UL, 0x0000000004000200UL, 0x0100080004000200UL, 0
               x0100000000000200UL, 0x0100080000000000UL, 0x0000000000000200UL, 0
```

```
               x0100080004000000UL, 0x0100080000000200UL, 0x0100000004000200UL, 0
               x0100000004000000UL, 0x0000000000000000UL, 0x0000080000000000UL, 0
               x0000080004000200UL },
29        { 0x0000010000000000UL, 0x0200000008000000UL, 0x0000000000000400UL, 0
               x0000000008000400UL, 0x0200000000000400UL, 0x0200010008000400UL, 0
               x0200010000000000UL, 0x0000010008000000UL, 0x0200000000000000UL, 0
               x0000010008000400UL, 0x0200010008000000UL, 0x0200010000000400UL, 0
               x0000010000000400UL, 0x0000000000000000UL, 0x0000000008000000UL, 0
               x0200000008000400UL },
30        { 0x0000000001000000UL, 0x0000020000000800UL, 0x0400000000000000UL, 0
               x0400000000000800UL, 0x0400020000000000UL, 0x0400020001000800UL, 0
               x0000020001000000UL, 0x0000000001000800UL, 0x0000020000000000UL, 0
               x0400000001000800UL, 0x0000020001000800UL, 0x0400020001000000UL, 0
               x0400000001000000UL, 0x0000000000000000UL, 0x0000000000000800UL, 0
               x0400020000000800UL },
31        { 0x0000000000001000UL, 0x8000000020000000UL, 0x0000400000000000UL, 0
               x8000400000000000UL, 0x0000400020000000UL, 0x8000400020001000UL, 0
               x0000000020001000UL, 0x8000000000001000UL, 0x0000000020000000UL, 0
               x8000400000001000UL, 0x8000000020001000UL, 0x0000400020001000UL, 0
               x0000400000001000UL, 0x0000000000000000UL, 0x8000000000000000UL, 0
               x8000400020000000UL },
32        { 0x1000000000000000UL, 0x0000800000002000UL, 0x0000000040000000UL, 0
               x0000800040000000UL, 0x0000000040002000UL, 0x1000800040002000UL, 0
               x1000000000002000UL, 0x1000800000000000UL, 0x0000000000002000UL, 0
               x1000800040000000UL, 0x1000800000002000UL, 0x1000000040002000UL, 0
               x1000000040000000UL, 0x0000000000000000UL, 0x0000800000000000UL, 0
               x0000800040002000UL },
33        { 0x0000100000000000UL, 0x2000000080000000UL, 0x0000000000004000UL, 0
               x0000000080004000UL, 0x2000000000004000UL, 0x2000100080004000UL, 0
               x2000100000000000UL, 0x0000100080000000UL, 0x2000000000000000UL, 0
               x0000100080004000UL, 0x2000100080000000UL, 0x2000100000004000UL, 0
               x0000100000004000UL, 0x0000000000000000UL, 0x0000000080000000UL, 0
               x2000000080004000UL },
34        { 0x0000000010000000UL, 0x0000200000008000UL, 0x4000000000000000UL, 0
               x4000000000008000UL, 0x4000200000000000UL, 0x4000200010008000UL, 0
               x0000200010000000UL, 0x0000000010008000UL, 0x0000200000000000UL, 0
               x4000000010008000UL, 0x0000200010008000UL, 0x4000200010000000UL, 0
               x4000000010000000UL, 0x0000000000000000UL, 0x0000000000008000UL, 0
               x4000200000008000UL }
35 };
36
37 void gift_64_table_generate_round_keys(uint64_t rks[restrict ROUNDS_GIFT_64],
38                                         const uint64_t key[restrict 2])
39 {
40        uint64_t key_state[] = {key[0], key[1]};
41        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
42                int v = (key_state[0] >> 0 ) & 0xffff;
43                int u = (key_state[0] >> 16) & 0xffff;
44
45                // add round key (RK=U||V)
46                rks[round] = 0UL;
47                for (size_t i = 0; i < 16; i++) {
48                        int key_bit_v  = (v >> i)  & 0x1;
49                        int key_bit_u  = (u >> i)  & 0x1;
50                        rks[round] ^= (uint64_t)key_bit_v << (i * 4 + 0);
51                        rks[round] ^= (uint64_t)key_bit_u << (i * 4 + 1);
52                }
53
54                // add single bit
55                rks[round] ^= 1UL << 63;
56
57                // add round constants
```

```
58                       rks[round] ^= ((round_const[round] >> 0) & 0x1) << 3;
59                       rks[round] ^= ((round_const[round] >> 1) & 0x1) << 7;
60                       rks[round] ^= ((round_const[round] >> 2) & 0x1) << 11;
61                       rks[round] ^= ((round_const[round] >> 3) & 0x1) << 15;
62                       rks[round] ^= ((round_const[round] >> 4) & 0x1) << 19;
63                       rks[round] ^= ((round_const[round] >> 5) & 0x1) << 23;
64
65                       // update key state
66                       int k0 = (key_state[0] >> 0 ) & 0xffffUL;
67                       int k1 = (key_state[0] >> 16) & 0xffffUL;
68                       k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
69                       k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
70                       key_state[0] >>= 32;
71                       key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
72                       key_state[1] >>= 32;
73                       key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
74               }
75       }
76
77       uint64_t gift_64_table_subperm(const uint64_t cipher_state)
78       {
79               uint64_t new_cipher_state = 0;
80
81               for (size_t i = 0; i < 16; i++) {
82                       int nibble = (cipher_state >> (i * 4)) & 0xf;
83                       new_cipher_state ^= tables[i][nibble];
84               }
85
86               return new_cipher_state;
87       }
88
89       uint64_t gift_64_table_encrypt(const uint64_t m,
90                                       const uint64_t rks[restrict ROUNDS_GIFT_64])
91       {
92               uint64_t c = m;
93
94               // round loop
95               for (int round = 0; round < ROUNDS_GIFT_64; round++) {
96                       c = gift_64_table_subperm(c);
97                       c ^= rks[round];
98               }
99
100              return c;
101      }
```

A.1.2 Using `vperm`

Listing A.3: `gift_vec_sbox.h`

```
1    #ifndef GIFT_VEC_SBOX_H
2    #define GIFT_VEC_SBOX_H
3
4    #include <stdint.h>
5    #include <arm_neon.h>
6
7    #define ROUNDS_GIFT_64 28
8
9    // expose for benchmarking
```

```
10  uint8x16_t gift_64_vec_sbox_bits_pack(const uint64_t a);
11  uint64_t   gift_64_vec_sbox_bits_unpack(const uint8x16_t a);
12  uint8x16_t gift_64_vec_sbox_subcells(const uint8x16_t cipher_state);
13  uint8x16_t gift_64_vec_sbox_subcells_inv(const uint8x16_t cipher_state);
14  uint8x16_t gift_64_vec_sbox_permute(const uint8x16_t cipher_state);
15  uint8x16_t gift_64_vec_sbox_permute_inv(const uint8x16_t cipher_state);
16  void       gift_64_vec_sbox_generate_round_keys(uint8x16_t rks[ROUNDS_GIFT_64],
17                                          const uint64_t key[restrict 2]);
18
19  // construct tables
20  void gift_64_vec_sbox_init(void);
21
22  uint64_t gift_64_vec_sbox_encrypt(const uint64_t m,
23                                    const uint8x16_t rks[restrict ROUNDS_GIFT_64]);
24  uint64_t gift_64_vec_sbox_decrypt(const uint64_t c,
25                                    const uint8x16_t rks[restrict ROUNDS_GIFT_64]);
26
27  #endif
```

Listing A.4: `gift_vec_sbox.c`

```
1   #include "gift_vec_sbox.h"
2
3   #include <stdint.h>
4   #include <arm_neon.h>
5   #include <string.h>
6
7   static uint64_t sbox_vec_u64[2] = {
8           0x09030f060c040a01UL, 0x0e080005070b0d02UL
9   };
10
11  static uint64_t sbox_vec_inv_u64[2] = {
12          0x0b040c020608000dUL, 0x050f09030a01070eUL
13  };
14
15  static uint8x16_t sbox_vec;
16  static uint8x16_t sbox_vec_inv;
17
18  // split S-box bits into vector lanes
19  uint8x16_t gift_64_vec_sbox_bits_pack(const uint64_t a)
20  {
21          uint8x16_t v;
22          v = vsetq_lane_u64(
23          (uint64_t)((a >> 4 * 0) & 0xf) << 8 * 0 |
24          (uint64_t)((a >> 4 * 1) & 0xf) << 8 * 1 |
25          (uint64_t)((a >> 4 * 2) & 0xf) << 8 * 2 |
26          (uint64_t)((a >> 4 * 3) & 0xf) << 8 * 3 |
27          (uint64_t)((a >> 4 * 4) & 0xf) << 8 * 4 |
28          (uint64_t)((a >> 4 * 5) & 0xf) << 8 * 5 |
29          (uint64_t)((a >> 4 * 6) & 0xf) << 8 * 6 |
30          (uint64_t)((a >> 4 * 7) & 0xf) << 8 * 7, v, 0);
31
32          v = vsetq_lane_u64(
33          (uint64_t)((a >> 4 * 8)  & 0xf) << 8 * 0 |
34          (uint64_t)((a >> 4 * 9)  & 0xf) << 8 * 1 |
35          (uint64_t)((a >> 4 * 10) & 0xf) << 8 * 2 |
36          (uint64_t)((a >> 4 * 11) & 0xf) << 8 * 3 |
37          (uint64_t)((a >> 4 * 12) & 0xf) << 8 * 4 |
38          (uint64_t)((a >> 4 * 13) & 0xf) << 8 * 5 |
39          (uint64_t)((a >> 4 * 14) & 0xf) << 8 * 6 |
40          (uint64_t)((a >> 4 * 15) & 0xf) << 8 * 7, v, 1);
```

```
41
42          return v;
43  }
44
45  // merge S-box bits into single uint64_t
46  uint64_t gift_64_vec_sbox_bits_unpack(const uint8x16_t v)
47  {
48          uint64_t a = 0UL;
49          a |= (uint64_t)vgetq_lane_u8(v, 0) << 4 * 0;
50          a |= (uint64_t)vgetq_lane_u8(v, 1) << 4 * 1;
51          a |= (uint64_t)vgetq_lane_u8(v, 2) << 4 * 2;
52          a |= (uint64_t)vgetq_lane_u8(v, 3) << 4 * 3;
53          a |= (uint64_t)vgetq_lane_u8(v, 4) << 4 * 4;
54          a |= (uint64_t)vgetq_lane_u8(v, 5) << 4 * 5;
55          a |= (uint64_t)vgetq_lane_u8(v, 6) << 4 * 6;
56          a |= (uint64_t)vgetq_lane_u8(v, 7) << 4 * 7;
57          a |= (uint64_t)vgetq_lane_u8(v, 8) << 4 * 8;
58          a |= (uint64_t)vgetq_lane_u8(v, 9) << 4 * 9;
59          a |= (uint64_t)vgetq_lane_u8(v, 10) << 4 * 10;
60          a |= (uint64_t)vgetq_lane_u8(v, 11) << 4 * 11;
61          a |= (uint64_t)vgetq_lane_u8(v, 12) << 4 * 12;
62          a |= (uint64_t)vgetq_lane_u8(v, 13) << 4 * 13;
63          a |= (uint64_t)vgetq_lane_u8(v, 14) << 4 * 14;
64          a |= (uint64_t)vgetq_lane_u8(v, 15) << 4 * 15;
65
66          return a;
67  }
68
69  static const size_t perm_64[] = {
70          0, 17, 34, 51, 48, 1, 18, 35, 32, 49, 2, 19, 16, 33, 50, 3,
71          4, 21, 38, 55, 52, 5, 22, 39, 36, 53, 6, 23, 20, 37, 54, 7,
72          8, 25, 42, 59, 56, 9, 26, 43, 40, 57, 10, 27, 24, 41, 58, 11,
73          12, 29, 46, 63, 60, 13, 30, 47, 44, 61, 14, 31, 28, 45, 62, 15
74  };
75
76  static const size_t perm_64_inv[] = {
77          0, 5, 10, 15, 16, 21, 26, 31, 32, 37, 42, 47, 48, 53, 58, 63,
78          12, 1, 6, 11, 28, 17, 22, 27, 44, 33, 38, 43, 60, 49, 54, 59,
79          8, 13, 2, 7, 24, 29, 18, 23, 40, 45, 34, 39, 56, 61, 50, 55,
80          4, 9, 14, 3, 20, 25, 30, 19, 36, 41, 46, 35, 52, 57, 62, 51
81  };
82
83  static const int round_const[] = {
84          // rounds 0-15
85          0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F, 0x1E, 0x3C, 0x39, 0
                  x33, 0x27, 0x0E,
86          // rounds 16-31
87          0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0
                  x2E, 0x1C, 0x38,
88          // rounds 32-47
89          0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A, 0x34, 0x29, 0x12, 0x24, 0x08, 0
                  x11, 0x22, 0x04
90  };
91
92  uint8x16_t gift_64_vec_sbox_subcells(const uint8x16_t cipher_state)
93  {
94          return vqtbl1q_u8(sbox_vec, cipher_state);
95  }
96
97  uint8x16_t gift_64_vec_sbox_subcells_inv(const uint8x16_t cipher_state)
98  {
99          return vqtbl1q_u8(sbox_vec_inv, cipher_state);
```

```
100  }
101
102  uint8x16_t gift_64_vec_sbox_permute(const uint8x16_t cipher_state)
103  {
104          // collect individual bits into 64-bit register
105          uint64_t new_cipher_state = 0UL;
106          uint64_t boxes[2];
107          vst1q_u64(boxes, cipher_state);
108
109          for (size_t box = 0; box < 16; box++) {
110                  for (size_t i = 0; i < 4; i++) {
111                          const int bit = (boxes[box / 8] >> ((box % 8) * 8 + i)) & 0x1;
112                          new_cipher_state |= (uint64_t)bit << perm_64[box * 4 + i];
113                  }
114          }
115
116          return gift_64_vec_sbox_bits_pack(new_cipher_state);
117  }
118
119  uint8x16_t gift_64_vec_sbox_permute_inv(const uint8x16_t cipher_state)
120  {
121          // collect into 64-bit register (faster)
122          uint64_t new_cipher_state = 0;
123          uint64_t boxes[2];
124          vst1q_u64(boxes, cipher_state);
125
126          // S-box 0-7
127          for (size_t box = 0; box < 8; box++) {
128                  for (size_t i = 0; i < 4; i++) {
129                          const int bit = (boxes[0] >> (box * 8 + i)) & 0x1;
130                          new_cipher_state |= (uint64_t)bit << perm_64_inv[box * 4 + i];
131                  }
132          }
133
134          // S-box 8-15
135          for (size_t box = 0; box < 8; box++) {
136                  for (size_t i = 0; i < 4; i++) {
137                          const int bit = (boxes[1] >> (box * 8 + i)) & 0x1;
138                          new_cipher_state |= (uint64_t)bit << perm_64_inv[(box + 8) * 4
139                                  + i];
140                  }
141          }
142
143          return gift_64_vec_sbox_bits_pack(new_cipher_state);
144  }
145
146  void gift_64_vec_sbox_generate_round_keys(uint8x16_t rks[ROUNDS_GIFT_64],
147                                            const uint64_t key[2])
148  {
149          uint64_t key_state[] = {key[0], key[1]};
150          for (int round = 0; round < ROUNDS_GIFT_64; round++) {
151                  const int v = (key_state[0] >> 0 ) & 0xffff;
152                  const int u = (key_state[0] >> 16) & 0xffff;
153
154                  // add round key (RK=U||V)
155                  uint64_t round_key = 0UL;
156                  for (size_t i = 0; i < 16; i++) {
157                          const int key_bit_v  = (v >> i)  & 0x1;
158                          const int key_bit_u  = (u >> i)  & 0x1;
159                          round_key ^= (uint64_t)key_bit_v << (i * 4 + 0);
160                          round_key ^= (uint64_t)key_bit_u << (i * 4 + 1);
161                  }
```

```
161
162                    // add single bit
163                    round_key ^= 1UL << 63;
164
165                    // add round constants
166                    round_key ^= ((round_const[round] >> 0) & 0x1) << 3;
167                    round_key ^= ((round_const[round] >> 1) & 0x1) << 7;
168                    round_key ^= ((round_const[round] >> 2) & 0x1) << 11;
169                    round_key ^= ((round_const[round] >> 3) & 0x1) << 15;
170                    round_key ^= ((round_const[round] >> 4) & 0x1) << 19;
171                    round_key ^= ((round_const[round] >> 5) & 0x1) << 23;
172
173                    // pack into vector register
174                    rks[round] = gift_64_vec_sbox_bits_pack(round_key);
175
176                    // update key state
177                    int k0 = (key_state[0] >> 0 ) & 0xffffUL;
178                    int k1 = (key_state[0] >> 16) & 0xffffUL;
179                    k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
180                    k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
181                    key_state[0] >>= 32;
182                    key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
183                    key_state[1] >>= 32;
184                    key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
185            }
186  }
187
188  void gift_64_vec_sbox_init(void)
189  {
190            // construct sbox_vec
191            sbox_vec = vld1q_u64(sbox_vec_u64);
192
193            // construct sbox_vec_inv
194            sbox_vec_inv = vld1q_u64(sbox_vec_inv_u64);
195  }
196
197  uint64_t gift_64_vec_sbox_encrypt(const uint64_t m,
198                                    const uint8x16_t rks[restrict ROUNDS_GIFT_64])
199  {
200            // pack into vector register
201            uint8x16_t c = gift_64_vec_sbox_bits_pack(m);
202
203            // round loop
204            for (int round = 0; round < ROUNDS_GIFT_64; round++) {
205                    c = gift_64_vec_sbox_subcells(c);
206                    c = gift_64_vec_sbox_permute(c);
207                    c = veorq_u8(c, rks[round]);
208            }
209
210            // unpack
211            return gift_64_vec_sbox_bits_unpack(c);
212  }
213
214  uint64_t gift_64_vec_sbox_decrypt(const uint64_t c,
215                                    const uint8x16_t rks[restrict ROUNDS_GIFT_64])
216  {
217            // pack into vector register
218            uint8x16_t m = gift_64_vec_sbox_bits_pack(c);
219
220            // round loop (in reverse)
221            for (int round = ROUNDS_GIFT_64 - 1; round >= 0; round--) {
222                    m = veorq_u8(m, rks[round]);
```

```
223                    m = gift_64_vec_sbox_permute_inv(m);
224                    m = gift_64_vec_sbox_subcells_inv(m);
225            }
226
227            // unpack
228            return gift_64_vec_sbox_bits_unpack(m);
229  }
```

A.1.3   Bitslicing

Listing A.5: `gift_vec_sliced.h`

```
1   #ifndef GIFT_VEC_SLICED_H
2   #define GIFT_VEC_SLICED_H
3
4   #include <stdint.h>
5   #include <arm_neon.h>
6
7   #define ROUNDS_GIFT_64 28
8
9   // expose for benchmarking
10  uint8x16_t shl(const uint8x16_t v, const int n);
11  uint8x16_t shr(const uint8x16_t v, const int n);
12  void gift_64_vec_sliced_swapmove(uint8x16_t *restrict a, uint8x16_t *restrict b,
13                                   const uint8x16_t m, const int n);
14  void gift_64_vec_sliced_bits_pack(uint8x16x4_t m[restrict 2]);
15  void gift_64_vec_sliced_bits_unpack(uint8x16x4_t m[restrict 2]);
16
17  void gift_64_vec_sliced_subcells(uint8x16x4_t cipher_state[restrict 2]);
18  void gift_64_vec_sliced_subcells_inv(uint8x16x4_t cipher_state[restrict 2]);
19  void gift_64_vec_sliced_permute(uint8x16x4_t cipher_state[restrict 2]);
20  void gift_64_vec_sliced_permute_inv(uint8x16x4_t cipher_state[2]);
21  void gift_64_vec_sliced_generate_round_keys(uint8x16x4_t rks[restrict ROUNDS_GIFT_64
        ][2],
22                                              const uint64_t key[restrict 2]);
23
24  void gift_64_vec_sliced_init(void);
25
26  void gift_64_vec_sliced_encrypt(uint64_t c[restrict 16],
27                                  const uint64_t m[restrict 16],
28                                  const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2]);
29  void gift_64_vec_sliced_decrypt(uint64_t m[restrict 16],
30                                  const uint64_t c[restrict 16],
31                                  const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2]);
32
33  #endif
```

Listing A.6: `gift_vec_sliced.c`

```
1   #include "gift_vec_sliced.h"
2
3   #include <stddef.h>
4   #include <stdio.h>
5   #include <string.h>
6
7   static uint64_t pack_shf_u64[4] = {
8           0x1303120211011000UL, 0x1707160615051404UL, // S0/S1/S2/S3
9           0x1b0b1a0a19091808UL, 0x1f0f1e0e1d0d1c0cUL, // S4/S5/S6/S7
```

```
10 };
11
12 static uint64_t pack_shf_inv_u64[4] = {
13         0x0e0c0a0806040200UL, 0x1e1c1a1816141210UL, // S0/S1/S2/S3
14         0x0f0d0b0907050301UL, 0x1f1d1b1917151311UL, // S4/S5/S6/S7
15 };
16
17 static uint64_t perm_u64[8] = {
18         0x0f0b07030c080400UL, 0x0d0905010e0a0602UL, // S0/S4
19         0x0c0804000d090501UL, 0x0e0a06020f0b0703UL, // S1/S5
20         0x0d0905010e0a0602UL, 0x0f0b07030c080400UL, // S2/S6
21         0x0e0a06020f0b0703UL, 0x0c0804000d090501UL  // S3/S7
22 };
23
24
25 static uint64_t perm_inv_u64[8] = {
26         0x05090d0104080c00UL, 0x070b0f03060a0e02UL, // S0/S4
27         0x090d0105080c0004UL, 0x0b0f03070a0e0206UL, // S1/S5
28         0x0d0105090c000408UL, 0x0f03070b0e02060aUL, // S2/S6
29         0x0105090d0004080cUL, 0x03070b0f02060a0eUL  // S3/S7
30 };
31
32 static uint8x16x2_t pack_shf;
33 static uint8x16x2_t pack_shf_inv;
34 static uint8x16x4_t perm;
35 static uint8x16x4_t perm_inv;
36
37 static uint8x16_t pack_mask_0;
38 static uint8x16_t pack_mask_1;
39 static uint8x16_t pack_mask_2;
40
41 static const int round_const[] = {
42         // rounds 0-15
43         0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F, 0x1E, 0x3C, 0x39, 0
                x33, 0x27, 0x0E,
44         // rounds 16-31
45         0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0
                x2E, 0x1C, 0x38,
46         // rounds 32-47
47         0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A, 0x34, 0x29, 0x12, 0x24, 0x08, 0
                x11, 0x22, 0x04
48 };
49
50 uint8x16_t shl(const uint8x16_t v, const int n)
51 {
52         uint64_t l[2];
53         vst1q_u64(l, v);
54         l[1] = (l[1] << n) | (l[0] >> (64 - n));
55         l[0] <<= n;
56         return vld1q_u64(l);
57 }
58
59 uint8x16_t shr(const uint8x16_t v, const int n)
60 {
61         uint64_t l[2];
62         vst1q_u64(l, v);
63         l[0] = l[0] >> n | (l[1] << (64 - n));
64         l[1] >>= n;
65         return vld1q_u64(l);
66 }
67
68 void gift_64_vec_sliced_swapmove(uint8x16_t *restrict a, uint8x16_t *restrict b,
```

```
69                                     const uint8x16_t m, const int n)
70  {
71
72          const uint8x16_t t = vandq_u8(veorq_u8(shr(*a, n), *b), m);
73          *b = veorq_u8(*b, t);
74          *a = veorq_u8(*a, shl(t, n));
75  }
76
77  void gift_64_vec_sliced_bits_pack(uint8x16x4_t m[restrict 2])
78  {
79          // take care not to shift mask bits out of the register
80          gift_64_vec_sliced_swapmove(&m[0].val[0], &m[0].val[1], pack_mask_0, 1);
81          gift_64_vec_sliced_swapmove(&m[0].val[2], &m[0].val[3], pack_mask_0, 1);
82          gift_64_vec_sliced_swapmove(&m[1].val[0], &m[1].val[1], pack_mask_0, 1);
83          gift_64_vec_sliced_swapmove(&m[1].val[2], &m[1].val[3], pack_mask_0, 1);
84
85          gift_64_vec_sliced_swapmove(&m[0].val[0], &m[0].val[2], pack_mask_1, 2);
86          gift_64_vec_sliced_swapmove(&m[0].val[1], &m[0].val[3], pack_mask_1, 2);
87          gift_64_vec_sliced_swapmove(&m[1].val[0], &m[1].val[2], pack_mask_1, 2);
88          gift_64_vec_sliced_swapmove(&m[1].val[1], &m[1].val[3], pack_mask_1, 2);
89
90          // make bytes (a0 b0 c0 d0 a4 b4 c4 d4 -> a0 b0 c0 d0 e0 f0 g0 h0)
91          gift_64_vec_sliced_swapmove(&m[0].val[0], &m[1].val[0], pack_mask_2, 4);
92          gift_64_vec_sliced_swapmove(&m[0].val[2], &m[1].val[2], pack_mask_2, 4);
93          gift_64_vec_sliced_swapmove(&m[0].val[1], &m[1].val[1], pack_mask_2, 4);
94          gift_64_vec_sliced_swapmove(&m[0].val[3], &m[1].val[3], pack_mask_2, 4);
95
96          // same plaintext slice bits into same register (so we only have to do
97          // what we are doing here once instead of every round)
98          const uint8x16x2_t pairs[4] = {
99                  { .val = { m[0].val[0], m[1].val[0] }},
100                 { .val = { m[0].val[1], m[1].val[1] }},
101                 { .val = { m[0].val[2], m[1].val[2] }},
102                 { .val = { m[0].val[3], m[1].val[3] }},
103         };
104
105         m[0].val[0] = vqtbl2q_u8(pairs[0], pack_shf.val[0]);
106         m[0].val[1] = vqtbl2q_u8(pairs[1], pack_shf.val[0]);
107         m[0].val[2] = vqtbl2q_u8(pairs[2], pack_shf.val[0]);
108         m[0].val[3] = vqtbl2q_u8(pairs[3], pack_shf.val[0]);
109
110         m[1].val[0] = vqtbl2q_u8(pairs[0], pack_shf.val[1]);
111         m[1].val[1] = vqtbl2q_u8(pairs[1], pack_shf.val[1]);
112         m[1].val[2] = vqtbl2q_u8(pairs[2], pack_shf.val[1]);
113         m[1].val[3] = vqtbl2q_u8(pairs[3], pack_shf.val[1]);
114 }
115
116 void gift_64_vec_sliced_bits_unpack(uint8x16x4_t m[restrict 2])
117 {
118         const uint8x16x2_t pairs[4] = {
119                 { .val = { m[0].val[0], m[1].val[0] }},
120                 { .val = { m[0].val[1], m[1].val[1] }},
121                 { .val = { m[0].val[2], m[1].val[2] }},
122                 { .val = { m[0].val[3], m[1].val[3] }},
123         };
124
125         m[0].val[0] = vqtbl2q_u8(pairs[0], pack_shf_inv.val[0]);
126         m[0].val[1] = vqtbl2q_u8(pairs[1], pack_shf_inv.val[0]);
127         m[0].val[2] = vqtbl2q_u8(pairs[2], pack_shf_inv.val[0]);
128         m[0].val[3] = vqtbl2q_u8(pairs[3], pack_shf_inv.val[0]);
129
130         m[1].val[0] = vqtbl2q_u8(pairs[0], pack_shf_inv.val[1]);
```

```
131         m[1].val[1] = vqtbl2q_u8(pairs[1], pack_shf_inv.val[1]);
132         m[1].val[2] = vqtbl2q_u8(pairs[2], pack_shf_inv.val[1]);
133         m[1].val[3] = vqtbl2q_u8(pairs[3], pack_shf_inv.val[1]);
134
135         // take care not to shift mask bits out of the register
136         gift_64_vec_sliced_swapmove(&m[0].val[0], &m[0].val[1], pack_mask_0, 1);
137         gift_64_vec_sliced_swapmove(&m[0].val[2], &m[0].val[3], pack_mask_0, 1);
138         gift_64_vec_sliced_swapmove(&m[1].val[0], &m[1].val[1], pack_mask_0, 1);
139         gift_64_vec_sliced_swapmove(&m[1].val[2], &m[1].val[3], pack_mask_0, 1);
140
141         gift_64_vec_sliced_swapmove(&m[0].val[0], &m[0].val[2], pack_mask_1, 2);
142         gift_64_vec_sliced_swapmove(&m[0].val[1], &m[0].val[3], pack_mask_1, 2);
143         gift_64_vec_sliced_swapmove(&m[1].val[0], &m[1].val[2], pack_mask_1, 2);
144         gift_64_vec_sliced_swapmove(&m[1].val[1], &m[1].val[3], pack_mask_1, 2);
145
146         // make bytes (a0 b0 c0 d0 a4 b4 c4 d4 -> a0 b0 c0 d0 e0 f0 g0 h0)
147         gift_64_vec_sliced_swapmove(&m[0].val[0], &m[1].val[0], pack_mask_2, 4);
148         gift_64_vec_sliced_swapmove(&m[0].val[2], &m[1].val[2], pack_mask_2, 4);
149         gift_64_vec_sliced_swapmove(&m[0].val[1], &m[1].val[1], pack_mask_2, 4);
150         gift_64_vec_sliced_swapmove(&m[0].val[3], &m[1].val[3], pack_mask_2, 4);
151 }
152
153 void gift_64_vec_sliced_subcells(uint8x16x4_t cs[restrict 2])
154 {
155         cs[0].val[1] = veorq_u8(cs[0].val[1],
156                             vandq_u8(cs[0].val[0], cs[0].val[2]));
157         uint8x16_t t = veorq_u8(cs[0].val[0],
158                             vandq_u8(cs[0].val[1], cs[0].val[3]));
159         cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
160         cs[0].val[0] = veorq_u8(cs[0].val[3], cs[0].val[2]);
161         cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
162         cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
163         cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
164         cs[0].val[3] = t;
165
166         cs[1].val[1] = veorq_u8(cs[1].val[1],
167                             vandq_u8(cs[1].val[0], cs[1].val[2]));
168         t            = veorq_u8(cs[1].val[0],
169                             vandq_u8(cs[1].val[1], cs[1].val[3]));
170         cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
171         cs[1].val[0] = veorq_u8(cs[1].val[3], cs[1].val[2]);
172         cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
173         cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
174         cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
175         cs[1].val[3] = t;
176 }
177
178 void gift_64_vec_sliced_subcells_inv(uint8x16x4_t cs[restrict 2])
179 {
180         uint8x16_t t = cs[0].val[3];
181         cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
182         cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
183         cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
184         cs[0].val[3] = veorq_u8(cs[0].val[0], cs[0].val[2]);
185         cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
186         cs[0].val[0] = veorq_u8(t, vandq_u8(cs[0].val[1], cs[0].val[3]));
187         cs[0].val[1] = veorq_u8(cs[0].val[1],
188                             vandq_u8(cs[0].val[0], cs[0].val[2]));
189
190         t            = cs[1].val[3];
191         cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
192         cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
```

```
193            cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
194            cs[1].val[3] = veorq_u8(cs[1].val[0], cs[1].val[2]);
195            cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
196            cs[1].val[0] = veorq_u8(t, vandq_u8(cs[1].val[1], cs[1].val[3]));
197            cs[1].val[1] = veorq_u8(cs[1].val[1],
198                              vandq_u8(cs[1].val[0], cs[1].val[2]));
199 }
200
201 void gift_64_vec_sliced_permute(uint8x16x4_t cs[restrict 2])
202 {
203            cs[0].val[0] = vqtbl1q_u8(cs[0].val[0], perm.val[0]);
204            cs[0].val[1] = vqtbl1q_u8(cs[0].val[1], perm.val[1]);
205            cs[0].val[2] = vqtbl1q_u8(cs[0].val[2], perm.val[2]);
206            cs[0].val[3] = vqtbl1q_u8(cs[0].val[3], perm.val[3]);
207
208            cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm.val[0]);
209            cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm.val[1]);
210            cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm.val[2]);
211            cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm.val[3]);
212 }
213
214 void gift_64_vec_sliced_permute_inv(uint8x16x4_t cs[restrict 2])
215 {
216            cs[0].val[0] = vqtbl1q_u8(cs[0].val[0], perm_inv.val[0]);
217            cs[0].val[1] = vqtbl1q_u8(cs[0].val[1], perm_inv.val[1]);
218            cs[0].val[2] = vqtbl1q_u8(cs[0].val[2], perm_inv.val[2]);
219            cs[0].val[3] = vqtbl1q_u8(cs[0].val[3], perm_inv.val[3]);
220
221            cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm_inv.val[0]);
222            cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm_inv.val[1]);
223            cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm_inv.val[2]);
224            cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm_inv.val[3]);
225 }
226
227 void gift_64_vec_sliced_generate_round_keys(uint8x16x4_t rks[restrict ROUNDS_GIFT_64
         ][2],
228                                               const uint64_t key[restrict 2])
229 {
230            uint64_t key_state[] = {key[0], key[1]};
231            for (int round = 0; round < ROUNDS_GIFT_64; round++) {
232                    const int v = (key_state[0] >> 0 ) & 0xffff;
233                    const int u = (key_state[0] >> 16) & 0xffff;
234
235                    // add round key (RK=U||V)
236                    // (slice 2 stays unused)
237                    uint64_t rk[6] = { 0x0UL };
238                    for (size_t i = 0; i < 8; i++) {
239                            int key_bit_v  = (v >> (i + 0)) & 0x1;
240                            int key_bit_u  = (u >> (i + 0)) & 0x1;
241                            rk[0]          ^= (uint64_t)key_bit_v << (i * 8);
242                            rk[2]          ^= (uint64_t)key_bit_u << (i * 8);
243
244                            key_bit_v      = (v >> (i + 8)) & 0x1;
245                            key_bit_u      = (u >> (i + 8)) & 0x1;
246                            rk[1]          ^= (uint64_t)key_bit_v << (i * 8);
247                            rk[3]          ^= (uint64_t)key_bit_u << (i * 8);
248                    }
249
250                    // add single bit
251                    rk[5] ^= 1UL << (7 * 8);
252
253                    // add round constants
```

```
254                    rk[4] ^= ((uint64_t)(round_const[round] >> 0) & 0x1) << (0 * 8);
255                    rk[4] ^= ((uint64_t)(round_const[round] >> 1) & 0x1) << (1 * 8);
256                    rk[4] ^= ((uint64_t)(round_const[round] >> 2) & 0x1) << (2 * 8);
257                    rk[4] ^= ((uint64_t)(round_const[round] >> 3) & 0x1) << (3 * 8);
258                    rk[4] ^= ((uint64_t)(round_const[round] >> 4) & 0x1) << (4 * 8);
259                    rk[4] ^= ((uint64_t)(round_const[round] >> 5) & 0x1) << (5 * 8);
260
261                    // extend bits to bytes
262                    for (size_t i = 0; i < 6; i++) {
263                            rk[i] |= rk[i] << 1;
264                            rk[i] |= rk[i] << 2;
265                            rk[i] |= rk[i] << 4;
266                    }
267
268                    rks[round][0].val[0] = vsetq_lane_u64(rk[0], rks[round][0].val[0], 0);
269                    rks[round][0].val[0] = vsetq_lane_u64(rk[1], rks[round][0].val[0], 1);
270                    rks[round][0].val[1] = vsetq_lane_u64(rk[2], rks[round][0].val[1], 0);
271                    rks[round][0].val[1] = vsetq_lane_u64(rk[3], rks[round][0].val[1], 1);
272                    rks[round][0].val[2] = vdupq_n_u8(0);
273                    rks[round][0].val[3] = vsetq_lane_u64(rk[4], rks[round][0].val[3], 0);
274                    rks[round][0].val[3] = vsetq_lane_u64(rk[5], rks[round][0].val[3], 1);
275                    rks[round][1]        = rks[round][0];
276
277                    // update key state
278                    int k0 = (key_state[0] >> 0 ) & 0xffffUL;
279                    int k1 = (key_state[0] >> 16) & 0xffffUL;
280                    k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
281                    k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
282                    key_state[0] >>= 32;
283                    key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
284                    key_state[1] >>= 32;
285                    key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
286            }
287 }
288
289 void gift_64_vec_sliced_init(void)
290 {
291         // bit packing shuffle
292         pack_shf = vld1q_u8_x2((uint8_t*)&pack_shf_u64[0]);
293
294         // inverse bit packing shuffle
295         pack_shf_inv = vld1q_u8_x2((uint8_t*)&pack_shf_inv_u64[0]);
296
297         // permutations
298         perm = vld1q_u8_x4((uint8_t*)&perm_u64[0]);
299
300         // inverse permutations
301         perm_inv = vld1q_u8_x4((uint8_t*)&perm_inv_u64[0]);
302
303         // packing masks
304         pack_mask_0 = vdupq_n_u8(0x55);
305         pack_mask_1 = vdupq_n_u8(0x33);
306         pack_mask_2 = vdupq_n_u8(0x0f);
307 }
308
309 void gift_64_vec_sliced_encrypt(uint64_t c[restrict 16],
310                                 const uint64_t m[restrict 16],
311                                 const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2])
312 {
313         uint8x16x4_t s[2];
314         s[0] = vld1q_u8_x4((uint8_t*)&m[0]);
315         s[1] = vld1q_u8_x4((uint8_t*)&m[8]);
```

```
316            gift_64_vec_sliced_bits_pack(s);
317
318            for (int round = 0; round < ROUNDS_GIFT_64; round++) {
319                    gift_64_vec_sliced_subcells(s);
320                    gift_64_vec_sliced_permute(s);
321
322                    // round key addition
323                    s[0].val[0] = veorq_u8(s[0].val[0], rks[round][0].val[0]);
324                    s[0].val[1] = veorq_u8(s[0].val[1], rks[round][0].val[1]);
325                    s[0].val[2] = veorq_u8(s[0].val[2], rks[round][0].val[2]);
326                    s[0].val[3] = veorq_u8(s[0].val[3], rks[round][0].val[3]);
327                    s[1].val[0] = veorq_u8(s[1].val[0], rks[round][1].val[0]);
328                    s[1].val[1] = veorq_u8(s[1].val[1], rks[round][1].val[1]);
329                    s[1].val[2] = veorq_u8(s[1].val[2], rks[round][1].val[2]);
330                    s[1].val[3] = veorq_u8(s[1].val[3], rks[round][1].val[3]);
331            }
332
333            gift_64_vec_sliced_bits_unpack(s);
334            vst1q_u8_x4((uint8_t*)&c[0], s[0]);
335            vst1q_u8_x4((uint8_t*)&c[8], s[1]);
336    }
337
338    void gift_64_vec_sliced_decrypt(uint64_t m[restrict 16],
339                                    const uint64_t c[restrict 16],
340                                    const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2])
341    {
342            uint8x16x4_t s[2];
343            s[0] = vld1q_u8_x4((uint8_t*)&c[0]);
344            s[1] = vld1q_u8_x4((uint8_t*)&c[8]);
345            gift_64_vec_sliced_bits_pack(s);
346
347            for (int round = ROUNDS_GIFT_64 - 1; round >= 0; round--) {
348                    // round key addition
349                    s[0].val[0] = veorq_u8(s[0].val[0], rks[round][0].val[0]);
350                    s[0].val[1] = veorq_u8(s[0].val[1], rks[round][0].val[1]);
351                    s[0].val[2] = veorq_u8(s[0].val[2], rks[round][0].val[2]);
352                    s[0].val[3] = veorq_u8(s[0].val[3], rks[round][0].val[3]);
353                    s[1].val[0] = veorq_u8(s[1].val[0], rks[round][1].val[0]);
354                    s[1].val[1] = veorq_u8(s[1].val[1], rks[round][1].val[1]);
355                    s[1].val[2] = veorq_u8(s[1].val[2], rks[round][1].val[2]);
356                    s[1].val[3] = veorq_u8(s[1].val[3], rks[round][1].val[3]);
357
358                    gift_64_vec_sliced_permute_inv(s);
359                    gift_64_vec_sliced_subcells_inv(s);
360            }
361
362            gift_64_vec_sliced_bits_unpack(s);
363            vst1q_u8_x4((uint8_t*)&m[0], s[0]);
364            vst1q_u8_x4((uint8_t*)&m[8], s[1]);
365    }
```

Appendix B

Lorem dolor

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] Kazumaro Aoki et al. "Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms — Design andAnalysis". In: *Selected Areas in Cryptography*. Ed. by Douglas R. Stinson and Stafford Tavares. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 39–56. ISBN: 978-3-540-44983-6.

[3] ARM Limited. "The Future is Built on Arm". URL: https://www.arm.com/company (visited on 04/07/2023).

[4] "ODROID-N2+ with 4GByte RAM". URL: https://www.hardkernel.com/shop/odroid-n2-with-4gbyte-ram-2/#tab-description (visited on 04/03/2023).

[5] ARM Limited. "ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile". 2013. URL: https://developer.arm.com/documentation/ddi0487/latest/ (visited on 02/02/2023).

[6] ARM Limited. "Arm Neon Intrinsics Reference". 2022. URL: https://arm-software.github.io/acle/neon_intrinsics/advsimd.html (visited on 02/07/2023).

[7] 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.

[8] ARM Limited. "Cortex-A72 Software Optimization Guide". 2015. URL: https://developer.arm.com/documentation/uan0016/a/ (visited on 04/06/2023).