Efficient Implementation Strategies for Block Ciphers on ARMv8

Bachelorarbeit

Bastian Engel

May 30, 2023

Abstract

Processor designs licensed by ARM are ubiquitous in many areas due to their low power consumption and heat generation. Being used in nearly all modern smartphones and tablets as well as in many desktop computers and IoT devices, the need for securing communications and confidentiality through encryption arises. Designing cryptographic algorithms for such low-power environments is an active area of research and various designs have been proposed. We examine in detail the two block ciphers GIFT and Camellia. For that, we first give an introduction of the designs as well as the ARMv8 platform we develop for. We then present multiple implementation strategies which leverage ARMv8 SIMD processing facilities and realize these in the C programming language using NEON intrinsics. In the end, we present performance benchmarks and a brief comparison to conclude this work. We find bit- and bytesliced implementations employing vector instructions to be the fastest.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Contents

Chapter 1

Introduction

1.1 Motivation

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 [1]. Following the proliferation of these systems, the need for new and efficient cryptographic algorithms to encrypt and decrypt data arises.

Two promising ciphers are GIFT and Camellia. Both of these have not yet been implemented for 64-bit ARMv8 - implementations only exist for 32-bit in the case of GIFT [2] and Intel AVX-2 and AES-NI in the case of Camellia [3]. Our contribution is to fill this gap and produce efficient implementations of GIFT and Camellia for 64-bit ARMv8.

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` [4], 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.

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

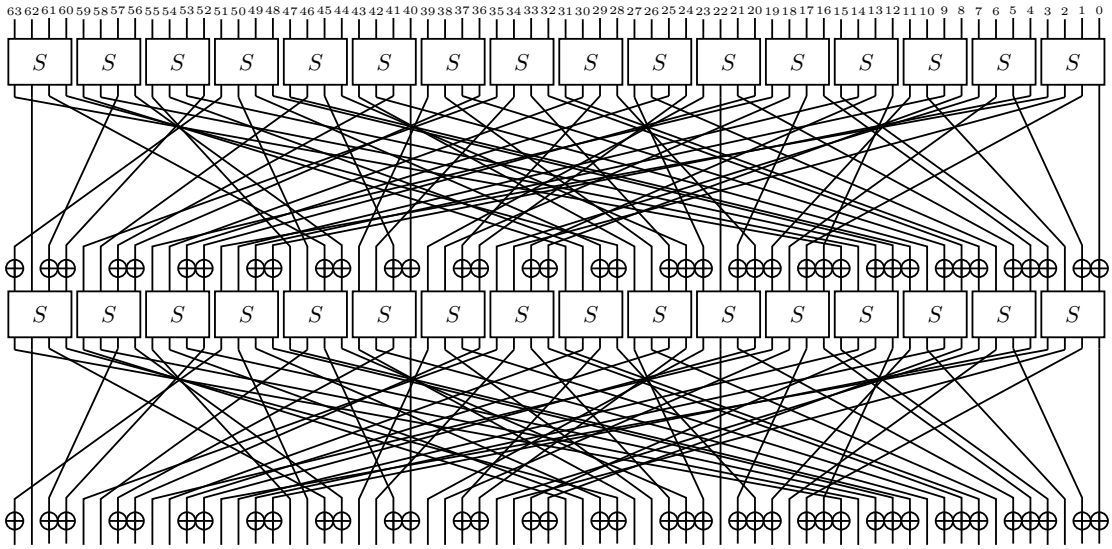

Figure 1.1: Two rounds of GIFT-64

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 $K_{(128)} = k_7||k_6||\ldots||k_0$ 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:

$$k_7||k_6||\dots||k_1||k_0 \leftarrow k_1 \ggg_2 ||k_0 \ggg_{12} ||k_7||k_6||\dots||k_3||k_2$$

1.2.2 Camellia

Camellia [5] 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 $kw_{t(64)}(t = 0, 1, 2, 3)$, $k_{u(64)}(u = 0, 1, \dots, (17|23)$ 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:

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

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

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		$(L\|\|R) \leftarrow FE(L, R, k_r)$

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 $\Sigma_i, i = (0, 1, \ldots, 5)$:

$$
\begin{aligned}
&K_A \leftarrow K_L \oplus K_R \\
&K_A \leftarrow FE(FE(K_A, \Sigma_0 0\text{xa09e667f3bcc908b}), \Sigma_1 0\text{xb67ae8584caa73b2}) \\
&K_A \leftarrow K_A \oplus K_L \\
&K_A \leftarrow FE(FE(K_A, \Sigma_2 0\text{xc6ef372fe94f82be}), \Sigma_3 0\text{x54ff53a5f1d36f1c}) \\
&K_B \leftarrow FE(FE(K_A, \Sigma_4 0\text{x10e527fade682d1d}), \Sigma_5 0\text{xb05688c2b3e6c1fd})
\end{aligned}
$$

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

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

Table 1.2: Subkey creation for 128-bit keys

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)	$k_{8(64)}$	$(K_A \lll_{45})_{L(64)}$		$kw_{3(64)}$	$(K_A \lll_{111})_{R(64)}$

SP-function F:

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

Substitution function S:

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

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:

$$\begin{aligned} P : \mathbb{F}_2^{64} &\to \mathbb{F}_2^{64} \\ \begin{pmatrix} z_7 \\ z_6 \\ z_5 \\ z_4 \\ z_3 \\ z_2 \\ z_1 \\ z_0 \end{pmatrix} &\mapsto \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} z_7 \\ z_6 \\ z_5 \\ z_4 \\ z_3 \\ z_2 \\ z_1 \\ z_0 \end{pmatrix} \end{aligned}$$

FL layer function FLL:

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

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

The ODROID-N2+ [6] development board we are using for testing is powered by an Amlogic S922X processor with four ARM Cortex-A73 and two weaker ARM Cortex-A53 cores based on the big.LITTLE architecture combining slower, more power efficient cores with faster, more power hungry ones. Both these cores are part of the eight generation of ARM designs known as ARMv8 [7] released in 2011 and succeeding the older 32-bit ARMv7.

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 64-bit A64 instruction set with NEON and crypto extensions.

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

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

data from register to memory and back. This enables faster execution of individual instructions, a simplier 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`.

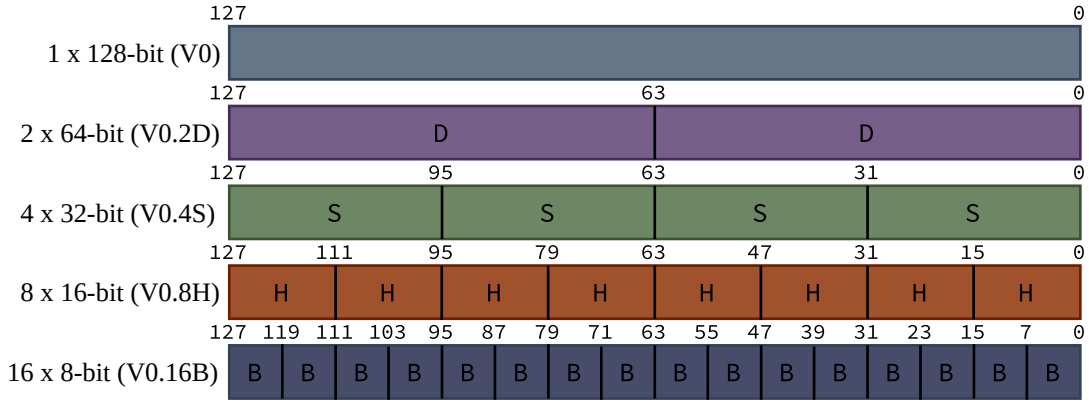

Figure 1.2: Divisions of the V0 register

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
```

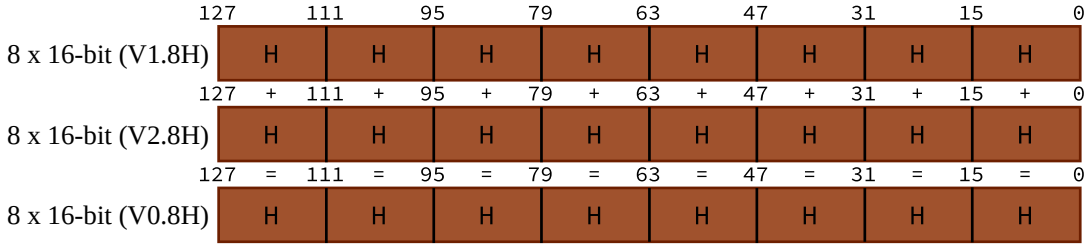

Figure 1.3: Addition of two vector registers

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 [8] 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:

```
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
`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`)
`uint64x2_t`	`vextq_u64(uint64x2_t, uint64x2_t, int)`	extract from pair of vectors

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 [9]:

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

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 arbitrary 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 $s = s_{15}||s_{14}||\dots||s_0$ 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:

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

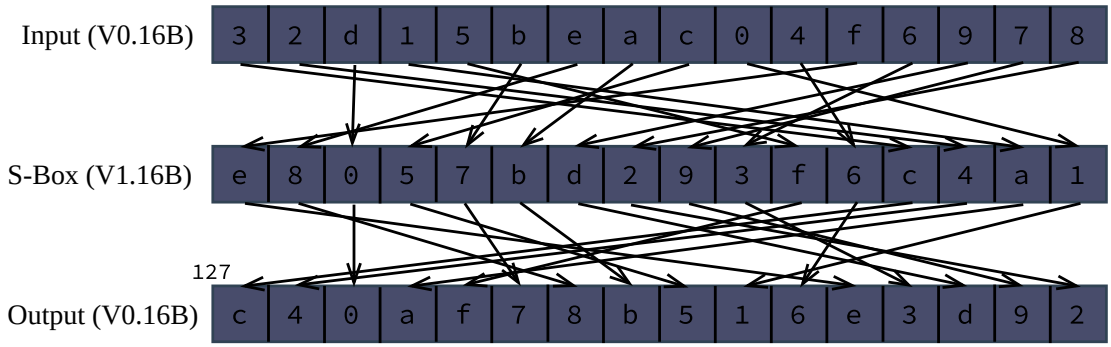

Figure 2.1: Performing the S-Box lookup in parallel

2.1.3 Bitslicing

Bitsliced implementation techniques were first introduced to improve the performance of DES in 1997 and work by viewing a processor with n-bit registers as a machine capable of executing n bitwise operations at once [10]. Bitslicing offers a performance advantage by splitting up n bits into m slices to achieve a more efficient representation which can exploit this bitwise parallelism. The structure

of GIFT naturally offers possibilities for bitslicing. We split the cipher state bits $b_{63}b_{62}\dots b_0$ 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.

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

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 a bitwise instruction sequence to apply the S-box which has already been proposed by the original GIFT authors:

$$\begin{aligned} S_1 &\leftarrow S_1 \oplus (S_0 \wedge S_2) \\ t &\leftarrow S_0 \oplus (S_1 \wedge S_3) \\ S_2 &\leftarrow S_2 \oplus (t \vee S_1) \\ S_0 &\leftarrow S_3 \oplus S_2 \\ S_1 &\leftarrow S_1 \oplus S_0 \\ S_0 &\leftarrow \neg S_0 \\ S_2 &\leftarrow S_2 \oplus (t \wedge S_1) \\ S_3 &\leftarrow t \end{aligned}$$

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:

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

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. Encrypting multiple blocks in parallel is a realistic use case since usually a lot of blocks are encrypted under the same key.

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 extracting and shifting the overflow ov out of $V = V[1]||V[0]$, shifting the lanes individually and finally ORing the overflow bits to the corresponding vector element.

$$\begin{aligned}&\text{shl}(V, n):\\ &\quad ov \;\; = V[0] \gg_{64-n}\\ &\quad V[0] \;= V[0] \ll\\ &\quad V[1] \;= (V[1] \ll) \vee ov\end{aligned}$$

The following operations group all i-th bits of the messages $a, c, \ldots, o$ into bytes and put 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.

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

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 Optimized non-SIMD

The original paper proposes various platform-independent ways to implement Camellia efficiently [5]. Only some of these apply to the ARMv8 architecture since features like the inline barrel shifter and bitfield manipulation instructions generally offer better performance.

XOR cancellation property in key schedule: While deriving K_A in the key schedule, the instruction sequence

$$\begin{aligned} K_A &\leftarrow K_L \oplus K_R \\ K_A &\leftarrow FE(FE(K_A, \Sigma_0), \Sigma_1) \\ K_A &\leftarrow K_A \oplus K_L \end{aligned}$$

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

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

Absorption of whitening keys: Whitening keys kw_1, kw_3 can be absorbed into other subkeys to save two XOR operations

Efficiently computing $(P \circ S)$**:** This technique applies to 64-bit processors. By preparing tables

$$\begin{aligned}
SP_0(y_{0(8)}) &= (s_0(y_0), s_0(y_0), s_0(y_0), 0, s_0(y_0), 0, 0, s_0(y_0)) \\
SP_1(y_{1(8)}) &= (0, s_1(y_1), s_1(y_1), s_1(y_1), s_1(y_1), s_1(y_1), 0, 0) \\
SP_2(y_{2(8)}) &= (s_2(y_2), 0, s_2(y_2), s_2(y_2), 0, s_2(y_2), s_2(y_2), 0) \\
SP_3(y_{3(8)}) &= (s_3(y_3), s_3(y_3), 0, s_3(y_3), 0, 0, s_3(y_3), s_3(y_3)) \\
SP_4(y_{4(8)}) &= (0, s_2(y_4), s_2(y_4), s_2(y_4), 0, s_2(y_4), s_2(y_4), s_2(y_4)) \\
SP_5(y_{5(8)}) &= (s_3(y_5), 0, s_3(y_5), s_3(y_5), s_3(y_5), 0, s_3(y_5), s_3(y_5)) \\
SP_6(y_{6(8)}) &= (s_4(y_6), s_4(y_6), 0, s_4(y_6), s_4(y_6), s_4(y_6), 0, s_4(y_6)) \\
SP_7(y_{7(8)}) &= (s_1(y_7), s_1(y_7), s_1(y_7), 0, s_1(y_7), s_1(y_7), s_1(y_7), 0)
\end{aligned}$$

,

we can compute $(P \circ S)(X_{(64)})$ using only 8 table lookups and 7 XORs in the following way:

$$(z'_1, z'_2, z'_3, z'_4, z'_5, z'_6, z'_7, z'_8) \leftarrow \bigoplus_{i=0}^{7} SP_i(y_i)$$

2.2.2 Byteslicing

Because Camellia is a byte-oriented block cipher, we can pursue a similar strategy as for GIFT: find a convenient data packing format and a way to apply necessary operations in parallel.

We choose a bytesliced representation by encrypting 16 plaintext blocks at once so we can fill 16 vector registers with register V_i containing all i-th bytes. This lends itself well to a NEON implementation since S-boxes as well as the FL layer, Feistel round and packing/unpacking functions can be implemented efficiently.

Packing and unpacking

Packing and unpacking of data can be done efficiently by use of 32 `TBL` instructions and is summarized in Figure 2.2. After packing, every i-th register will contain all i-th bytes of the 16 input blocks. Unpacking is done in a similar way with different permutation tables.

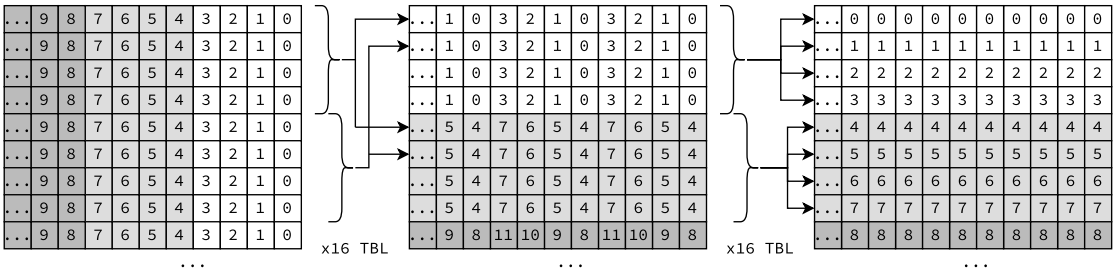

Figure 2.2: Camellia bytesliced packing function with 32 `TBL` operations. Curly brackets represent a 4-element vector array being used as source registers.

Hardware-accelerated Camellia S-box

A bytesliced implementation strategy for the S-box on ARMv8 can be derived from already existing x86-optimized implementations utilizing the AES-NI advanced encryption standard instruction set [3]. NEON itself possesses cryptographic extensions for finite field arithmetic and AES as well as SHA calculations. These can be used to produce an accelerated Camellia implementation due to the algebraic similarity of the AES- and Camellia S-boxes.

The AES S-box works by multiplicatively inverting $x \in \mathbb{F}_2^8$ over $\mathrm{GF}(2^8)$ and then applying an affine transformation A:

$$s(x) : \mathbb{F}_2^8 \to \mathbb{F}_2^8, x \mapsto A(x^{-1}_{\mathrm{GF}(2^8)})$$

The Camellia S-box s_0 is defined as follows:

$$s_0(x) : \mathbb{F}_2^8 \to \mathbb{F}_2^8, x \mapsto h(g(f(x \oplus 0xc5))) \oplus 0x6e$$

with affine transformations h, f and the multiplicative inversion function g in the composite field $\mathrm{GF}((2^4)^2)$. Because Galois fields of equal size are isomorphic, there exist affine isomorphisms δ, δ^{-1} between $\mathrm{GF}(2^8)$ and $\mathrm{GF}((2^4)^2)$ respectively [11]:

$$\begin{aligned} \delta &: \mathrm{GF}(2^8) \quad \to \mathrm{GF}((2^4)^2) \\ \delta^{-1} &: \mathrm{GF}((2^4)^2) \to \mathrm{GF}(2^8) \end{aligned}$$

NEON provides the AESE instruction for one round of AES encryption which works on a 128-bit vector register. By applying the inverse of ShiftRow to x, we can apply the AES S-box to 16 bytes at once.

$$\mathsf{AESE}(x) = \text{SubBytes}(\text{ShiftRow(x)}) \Leftrightarrow \mathsf{AESE}(\text{ShiftRow}^{-1}(x)) = \text{SubBytes}(x)$$

We can then reverse the affine transformation A due to bijectivity and extract the multiplicative inverse of all 16 vector bytes in $\text{GF}(2^8)$.

$$x^{-1}_{\text{GF}(2^8)} = A^{-1}(A(x^{-1}_{\text{GF}(2^8)})) = A^{-1}(\text{SubBytes}(x)) = A^{-1}(\mathsf{AESE}(\text{ShiftRow}^{-1}(x)))$$

Using the affine isomorphism, we transform this inverse into the inverse in $\text{GF}((2^4)^2)$ which is equal to $g(x)$:

$$g(x) = x^{-1}_{\text{GF}((2^4)^2)} = \delta(x^{-1}_{\text{GF}(2^8)}) = \delta(A^{-1}(\mathsf{AESE}(\text{ShiftRow}^{-1}(x))))$$

We now redefine h, f as H, F such that they include addition of constants $0xc5$ and $0x6e$. We can now use the inverse in conjunction with H and F and an additional input transformation to calculate the Camellia S-box:

$$\begin{aligned} s_0(x) &= h(g(f(x \oplus 0xc5))) \oplus 0x6e \\ &= H(g(F(x))) \\ &= H(\delta(A^{-1}(\mathsf{AESE}(\text{ShiftRow}^{-1}(\delta^{-1}(F(x))))))) \end{aligned}$$

Because of different bit endianness of the matrices representing δ, δ^{-1}, we define an additional bit-swapping function:

$$S = S^{-1} : \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \mapsto \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}$$

The final equation serving as the basis for our implementation then becomes

$$s_0(x) = S^{-1}(H(\delta(S(A^{-1}(\mathsf{AESE}(\text{ShiftRow}^{-1}(S^{-1}(\delta^{-1}(F(S(x)))))))))))$$

While appearing to be computationally too intensive for any performance gains, we shall notice that all of the transformations are affine and can therefore be combined into a single operation by simple matrix multiplication. We combine them into a pre-filter function θ_0 and a post-filter function ψ_0:

$$\theta_0(x) = S^{-1}(\delta^{-1}(F(S(x))))$$
$$\psi_0(x) = S^{-1}(H(\delta(S(A^{-1}(x)))))$$

This simplifies our final equation:

$$s_0(x) = \psi_0(\mathsf{AESE}(\text{ShiftRow}^{-1}(\theta_0(x))))$$

The other S-boxes s_1, s_2, s_3 are defined in terms of s_0 and rotations. We can include these rotations by constructing additional S-box-specific filters θ_3, ψ_1, ψ_2:

$$s_1(x) = s_0(x) \lll_1 = \psi_1(\mathsf{AESE}(\text{ShiftRow}^{-1}(\theta_0(x))))$$
$$s_2(x) = s_0(x) \ggg_1 = \psi_2(\mathsf{AESE}(\text{ShiftRow}^{-1}(\theta_0(x))))$$
$$s_3(x) = s_0(x \lll_1) = \psi_0(\mathsf{AESE}(\text{ShiftRow}^{-1}(\theta_3(x))))$$

Fast matrix multiplication

Efficient application of the filter functions is essential to performance and can be implemented in parallel by use of the `TBL` instruction. First note that matrix-vector multiplication $\mathbb{F}_2^{8\times 8} \cdot \mathbb{F}_2^8 \to \mathbb{F}_2^8$ can be decomposed into two multiplications $\mathbb{F}_2^{8\times 4} \cdot \mathbb{F}_2^4 \to \mathbb{F}_2^8$ with subsequent addition of the results:

$$\begin{pmatrix} 1&1&1&0&1&1&0&1\\ 0&0&1&1&0&0&1&0\\ 1&1&0&1&0&0&0&0\\ 1&0&1&1&0&0&1&1\\ 0&0&0&0&1&1&0&0\\ 1&0&1&0&0&1&0&0\\ 0&0&1&0&1&1&0&0\\ 1&0&0&0&0&1&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7 \end{pmatrix} = \begin{pmatrix} 1&1&0&1\\ 0&0&1&0\\ 0&0&0&0\\ 0&0&1&1\\ 1&1&0&0\\ 0&1&0&0\\ 1&1&0&0\\ 0&1&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_0\\ x_1\\ x_2\\ x_3 \end{pmatrix} \oplus \begin{pmatrix} 1&1&1&0\\ 0&0&1&1\\ 1&1&0&1\\ 1&0&1&1\\ 0&0&0&0\\ 1&0&1&0\\ 0&0&1&0\\ 1&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_4\\ x_5\\ x_6\\ x_7 \end{pmatrix}$$

We can then tabulate the decomposed multiplications since each input now consists of only four bits which allows us to fit the 16 possible 8-bit results into a single vector register. A single matrix multiplication is then executed for 16 bytes in parallel by use of two `TBL` and one XOR instruction with an additional AND and SHR operation for masking off the lower/upper 4 bits.

Permutation

Since all bytes of the same position are collected in the same register, applying the permutation can be realized in 16 XOR operations as described in the Camellia specification with the difference that we choose to not swap the lower and higher 4 bytes, but rather compensate for this by choosing the correct bytes when XORing with the other half later in the Feistel round.

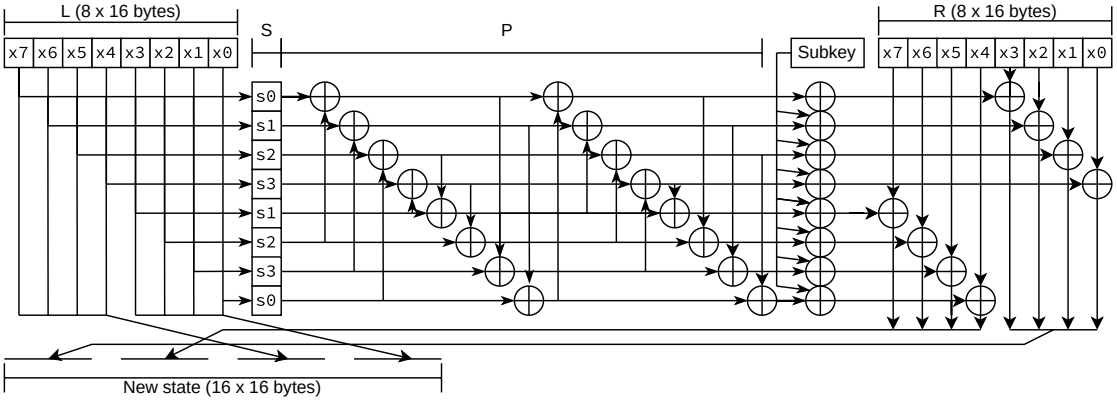

Figure 2.3: Camellia bytesliced Feistel round function

FL layer

The two functions FL, FL^{-1} can be implemented in a straightforward manner once 1-bit left rotation has been defined for the bytesliced representation. This can be facilitated by extracting the highest bit of each byte, shifting it all the way to the right and storing it as overflow ov_i. ov_i is then added to the left-shifted value of byte $(i + 4)(\mathrm{mod}4)$ for the final result.

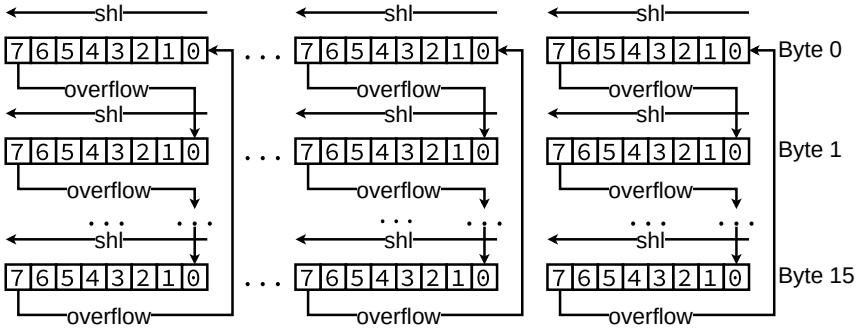

Figure 2.4: Camellia bytesliced 1-bit left rotate

Chapter 3

Implementation

Implementations in the C programming language for the presented strategies can be found in Appendix C. 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 code. All source files were compiled using clang version 15.0.7 at optimization level -O3.

3.1 Pipelining

Understanding certain implementation choices requires an understanding of the Cortex-A73 instruction pipeline [12]. 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.

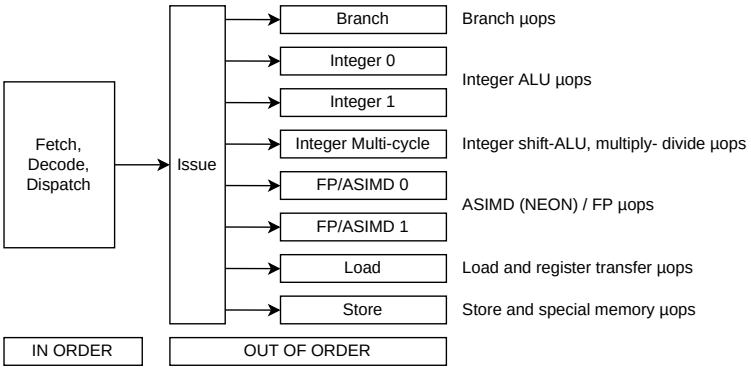

Figure 3.1: High-level overview of the Cortex-A72 instruction pipeline

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.

Listing 3.1: GIFT-64 table subperm

```
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;
}
```

Listing 3.2: GIFT-64 table encrypt

```
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;
}
```

Lots of operations require extraction of a certain number of consecutive bits, usually referred to as a bitfield. Indices for table lookups are generally attained by right-shifting a larger value stored in a register, then applying an AND operation to get the lowest n bits and finally writing the result into the beginning of another

register. Due to its usefulness, this operation is implemented as `UBFX` for an unsigned bitfield extract and can be used to implement S-box extraction for subperm lookups which would otherwise take two or three instructions. Interestingly, the A64 instruction set makes heavy use of instruction aliasing. The logical shift left instruction `lsl` for example is an alias of `UBFX` which itself is an alias for `UBFM`.

Another keyword aiding in optimization is `restrict` which can be used for pointer and array function parameters; the programmer can add this keyword to parameters to tell the compiler they are never aliased by any other pointers which allows the compiler to rearrange instructions and eliminate loads.

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.

Listing 3.3: `vperm` S-box

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

Listing 3.4: `vperm` permutation

```
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);
}
```

Listing 3.5: `vperm` encrypt function

```
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);

```

```
7         // round loop
8         for (int round = 0; round < ROUNDS_GIFT_64; round++) {
9                 c = gift_64_vec_sbox_subcells(c);
10                c = gift_64_vec_sbox_permute(c);
11                c = veorq_u8(c, rks[round]);
12        }
13
14        // unpack
15        return gift_64_vec_sbox_bits_unpack(c);
16 }
```

3.2.3 Bitslicing

A note on data storage

NEON only provides 32 vector registers. While this is more than the 16 YMM registers offered by Intel's AVX-256 vector extension, it is not enough to accompany all 28 round keys for GIFT-64 plus the 16 registers representing cipher state at once. Because loading and storing single registers is inefficient, data is represented using the vector array type `uint8x16x4_t`. Loads and stores are then assembled such that the whole array is loaded or stored in a single instruction. Loading a single vector from memory for example has a latency of 5 cycles while loading four vectors into a vector array can be done in 8 cycles which results in an amortized cost of 2 cycles per vector. Vectors are grouped into vector arrays as often as possible to reduce the number of necessary loads and stores.

Shifts for swapmove

Implementing shift functions for 128-bit NEON registers by extracting the overflow and adding it back in later can be realized using 5 instructions. It would be most useful for the function to take a shift amount parameter, but most NEON intrinsics encode their parameters into the final machine code such that parameters need to be compile-time constants. We will therefore implement `shl`, `shr` and `swapmove` as C macros utilizing the `vextq_u64` intrinsic to swap the two 64-bit vector elements:

Listing 3.6: Bitsliced GIFT swapmove and shift macros

```
1  /*
2  uint8x16_t shl(const uint8x16_t v, const int n) */
3  #define shl(_a, v, n)                                                    \
4  {                                                                        \
5          uint64x2_t _overflow = vshrq_n_u64(v, 64 - n);                   \
6          _overflow = vextq_u64(vdupq_n_u64(0x0), _overflow, 1);           \
7          _a = vorrq_u8(vshlq_n_u64(v, n), _overflow);                     \
8  }
9
10 /*
11 uint8x16_t shr(const uint8x16_t v, const int n) */
```

```
12 #define shr(_a, v, n)                                                    \
13 {                                                                        \
14         uint64x2_t _overflow = vshlq_n_u64(v, 64 - n);                  \
15         _overflow = vextq_u64(_overflow, vdupq_n_u64(0x0), 1);          \
16         _a = vorrq_u8(vshrq_n_u64(v, n), _overflow);                    \
17 }
18
19 /* implemented as a macro so we can use vshlq_n_u8 with variable n
20 void gift_64_vec_sliced_swapmove(uint8x16_t *restrict a, uint8x16_t *restrict b,
21                                  const uint8x16_t m, const int n) */
22 #define gift_64_vec_sliced_swapmove(a, b, m, n)                          \
23 {                                                                        \
24         uint8x16_t _a;                                                  \
25         shr(_a, a, n);                                                  \
26         const uint8x16_t _t = vandq_u8(veorq_u8(_a, b), m);             \
27         b = veorq_u8(b, _t);                                            \
28         shl(_a, _t, n);                                                 \
29         a = veorq_u8(a, _a);                                            \
30 }
```

Round function

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

Listing 3.7: Bitsliced GIFT S-box

```
1  void gift_64_vec_sliced_subcells(uint8x16x4_t cs[restrict 2])
2  {
3          cs[0].val[1] = veorq_u8(cs[0].val[1],
4                                  vandq_u8(cs[0].val[0], cs[0].val[2]));
5          uint8x16_t t = veorq_u8(cs[0].val[0],
6                                  vandq_u8(cs[0].val[1], cs[0].val[3]));
7          cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
8          cs[0].val[0] = veorq_u8(cs[0].val[3], cs[0].val[2]);
9          cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
10         cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
11         cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
12         cs[0].val[3] = t;
13
14         cs[1].val[1] = veorq_u8(cs[1].val[1],
15                                 vandq_u8(cs[1].val[0], cs[1].val[2]));
16         t            = veorq_u8(cs[1].val[0],
17                                 vandq_u8(cs[1].val[1], cs[1].val[3]));
18         cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
19         cs[1].val[0] = veorq_u8(cs[1].val[3], cs[1].val[2]);
20         cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
21         cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
22         cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
23         cs[1].val[3] = t;
24 }
```

Listing 3.8: Bitsliced GIFT permutation

```
1  void gift_64_vec_sliced_permute(uint8x16x4_t cs[restrict 2])
2  {
3          cs[0].val[0] = vqtbl1q_u8(cs[0].val[0], perm.val[0]);
4          cs[0].val[1] = vqtbl1q_u8(cs[0].val[1], perm.val[1]);
```

```
5         cs[0].val[2] = vqtbl1q_u8(cs[0].val[2], perm.val[2]);
6         cs[0].val[3] = vqtbl1q_u8(cs[0].val[3], perm.val[3]);
7
8         cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm.val[0]);
9         cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm.val[1]);
10        cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm.val[2]);
11        cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm.val[3]);
12  }
```

Listing 3.9: Round function

```
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 3.10: Round function assembly

```
1  and     v20.16b, v17.16b, v6.16b
2  add     x9, x20, x8
3  eor     v20.16b, v20.16b, v16.16b
4  add     x8, x8, #0x80
5  and     v21.16b, v20.16b, v19.16b
6  cmp     x8, #0xe70
7  eor     v21.16b, v21.16b, v6.16b
8  orr     v6.16b, v6.16b, v16.16b
9  eor     v6.16b, v6.16b, v17.16b
10 eor     v16.16b, v19.16b, v6.16b
11 eor     v17.16b, v16.16b, v20.16b
12 and     v19.16b, v21.16b, v17.16b
13 eor     v6.16b, v19.16b, v6.16b
14 and     v19.16b, v7.16b, v4.16b
15 eor     v19.16b, v19.16b, v5.16b
16 and     v20.16b, v19.16b, v18.16b
17 eor     v20.16b, v20.16b, v4.16b
18 orr     v4.16b, v4.16b, v5.16b
19 eor     v4.16b, v4.16b, v7.16b
20 eor     v5.16b, v18.16b, v4.16b
21 eor     v7.16b, v5.16b, v19.16b
22 and     v18.16b, v20.16b, v7.16b
23 eor     v4.16b, v18.16b, v4.16b
```

```
24 tbl     v18.16b, {v6.16b}, v2.16b
25 tbl     v19.16b, {v21.16b}, v3.16b
26 tbl     v21.16b, {v4.16b}, v2.16b
27 ldp     q6, q4, [x9, #-112]
28 mvn     v16.16b, v16.16b
29 tbl     v16.16b, {v16.16b}, v0.16b
30 tbl     v17.16b, {v17.16b}, v1.16b
31 mvn     v5.16b, v5.16b
32 tbl     v5.16b, {v5.16b}, v0.16b
33 ldp     q22, q23, [x9, #-80]
34 eor     v6.16b, v6.16b, v16.16b
35 eor     v16.16b, v4.16b, v17.16b
36 tbl     v7.16b, {v7.16b}, v1.16b
37 eor     v17.16b, v22.16b, v18.16b
38 tbl     v20.16b, {v20.16b}, v3.16b
39 ldp     q4, q18, [x9, #-48]
40 eor     v19.16b, v23.16b, v19.16b
41 eor     v4.16b, v4.16b, v5.16b
42 ldp     q22, q23, [x9, #-16]
43 eor     v5.16b, v18.16b, v7.16b
44 eor     v7.16b, v22.16b, v21.16b
45 eor     v18.16b, v23.16b, v20.16b
46 b.ne    197e0
```

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 instructions, 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 continues processing 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 assembler code.

3.3 Camellia

3.3.1 Optimized non-SIMD implementation

An optimized non-SIMD implementation based on the platform-independent techniques presented in the original paper [5] is relatively easy to achieve since all functional components are already well defined. One thing to take note of however is the fact that the specification is based on a big endian representation while ARMv8 is a little endian machine. The problem of endianness manifests itself whenever a memory-register interaction takes place, i.e. for loads and stores. Arithmetic on a register is unaffected since a register is always treated as one large number with no conception of addresses.

We will store input data and arrays in Camellia byte-order such that a lower array element actually represents a higher numerical value, i.e. the numerical value of

```
{ 0x0123456789abcdefUL, 0xfedcba9876543210UL }
```

is equal to

```
0x0123456789abcdeffedcba9876543210
```

which is equal to the byte string

```
01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
```

This allows us to change the endianness of the implementation while keeping input in the original Camellia byte order. A change in implementation endianness manifest itself for example in table lookups where, according to Camellia specification, the first byte is expected to be the most significant byte:

Listing 3.11: Camellia optimized SP function with reverse byte order

```
uint64_t camellia_spec_opt_F(uint64_t X, const uint64_t k)
{
        X ^= k;

        // compute P(S(X)) through large 64-bit lookup table
        uint64_t result = 0UL;
        result ^= SP0[(X >> 56) & 0xff];
        result ^= SP1[(X >> 48) & 0xff];
        result ^= SP2[(X >> 40) & 0xff];
        result ^= SP3[(X >> 32) & 0xff];
        result ^= SP4[(X >> 24) & 0xff];
        result ^= SP5[(X >> 16) & 0xff];
        result ^= SP6[(X >> 8 ) & 0xff];
        result ^= SP7[(X >> 0 ) & 0xff];

        return result;
}
```

3.3.2 Bytesliced implementation

Key generation

A possible approach to storing keys is storing them in bytesliced format so they can later be loaded from memory and used directly since we only have 32 vector registers in total and half of them are already occupied by the current cipher state. This can be implemented by first generating 64 bit round keys and then filling register V_i with the i-th byte by means of `vdupq_n_u8` for each key.

Listing 3.12: Bytesliced Camellia round key generation

```
void camellia_sliced_generate_round_keys_128(struct camellia_rks_sliced_128 *restrict
    rks,
                                             const uint64_t key[restrict 2])
{
        // use standard key derivation
        struct camellia_rks_128 rks_128;
        camellia_spec_opt_generate_round_keys_128(&rks_128, key);

        // now pack round keys by use of vdupq_n_u8 since all bytes are the same
        // in a bytesliced representation

        // whitening and FL layer keys
        for (size_t i = 0; i < 4; i++) {
                for (size_t byte = 0; byte < 8; byte++) {
                        uint8x16_t *reg_kw = &rks->kw[i][byte / 4].val[byte % 4];
```

```
15                          uint8x16_t *reg_kl = &rks->kl[i][byte / 4].val[byte % 4];
16
17                          *reg_kw = vdupq_n_u8((rks_128.kw[i] >> (8 * byte)) & 0xff);
18                          *reg_kl = vdupq_n_u8((rks_128.kl[i] >> (8 * byte)) & 0xff);
19                  }
20          }
21
22          // F function keys
23          for (size_t i = 0; i < 18; i++) {
24                  for (size_t byte = 0; byte < 8; byte++) {
25                          uint8x16_t *reg_ku = &rks->ku[i][byte / 4].val[byte % 4];
26
27                          *reg_ku = vdupq_n_u8((rks_128.ku[i] >> (8 * byte)) & 0xff);
28                  }
29          }
30  }
```

Another way to implement key generation is not to store the keys bytesliced, but rather to store the 64 bit values and create necessary vector registers on the fly. This saves 3120 bytes of memory and could show some usefulness for memory-constrained environments, but benchmarking shows a performance drop of about 3.6%. This is due to the fact that `vdupq_n_u8` has a latency of 8 cycles and a throughput of only 1, making it a great deal slower than simple vector array loads.

F-function

The s-function implements matrix multiplication with a given pre- and postfilter using the aforementioned strategy.

Listing 3.13: Bytesliced Camellia S-box

```
1   static uint8x16_t s(const uint8x16_t X,
2                       const uint8x16x2_t prefilter,
3                       const uint8x16x2_t postfilter)
4   {
5           // prefilter
6           uint8x16_t pre_low  = vqtbl1q_u8(prefilter.val[0],
7                                            vandq_u8(X, lower_4_bits_mask));
8           uint8x16_t pre_high = vqtbl1q_u8(prefilter.val[1],
9                                            vshrq_n_u8(X, 4));
10          uint8x16_t pre = veorq_u8(pre_low, pre_high);
11
12          // inverse ShiftRows
13          pre = vqtbl1q_u8(pre, shiftrows_inv);
14
15          // AES single round encryption (x <- AESSubBytes(AESShiftRows(x)))
16          uint8x16_t aesd = vaeseq_u8(pre, vdupq_n_u8(0x0));
17
18          // postfilter
19          uint8x16_t post_low  = vqtbl1q_u8(postfilter.val[0],
20                                            vandq_u8(aesd, lower_4_bits_mask));
21          uint8x16_t post_high = vqtbl1q_u8(postfilter.val[1],
22                                            vshrq_n_u8(aesd, 4));
23          uint8x16_t post = veorq_u8(post_low, post_high);
24
25          return post;
26  }
```

Just as before we need to invert the byte order since the first vector `X[0].val[0]` contains all least significant bytes, but the Camellia specification places the most significant bytes at the beginning.

Listing 3.14: Bytesliced Camellia F-function

```
void camellia_sliced_F(uint8x16x4_t X[restrict 2],
                       const uint8x16x4_t k[restrict 2])
{
        // key additions
        for (size_t byte = 0; byte < 8; byte++) {
                uint8x16_t *reg = &X[byte / 4].val[byte % 4];

                *reg = veorq_u8(*reg, k[byte / 4].val[byte % 4]);
        }

        // S-boxes (beware of endianness)
        X[1].val[3] = s(X[1].val[3], prefilter_0, postfilter_0); // s0
        X[1].val[2] = s(X[1].val[2], prefilter_0, postfilter_1); // s1
        X[1].val[1] = s(X[1].val[1], prefilter_0, postfilter_2); // s2
        X[1].val[0] = s(X[1].val[0], prefilter_3, postfilter_0); // s3
        X[0].val[3] = s(X[0].val[3], prefilter_0, postfilter_1); // s1
        X[0].val[2] = s(X[0].val[2], prefilter_0, postfilter_2); // s2
        X[0].val[1] = s(X[0].val[1], prefilter_3, postfilter_0); // s3
        X[0].val[0] = s(X[0].val[0], prefilter_0, postfilter_0); // s0

        // permutation
        X[1].val[3] = veorq_u8(X[1].val[3], X[0].val[2]);
        X[1].val[2] = veorq_u8(X[1].val[2], X[0].val[1]);
        X[1].val[1] = veorq_u8(X[1].val[1], X[0].val[0]);
        X[1].val[0] = veorq_u8(X[1].val[0], X[0].val[3]);
        X[0].val[3] = veorq_u8(X[0].val[3], X[1].val[1]);
        X[0].val[2] = veorq_u8(X[0].val[2], X[1].val[0]);
        X[0].val[1] = veorq_u8(X[0].val[1], X[1].val[3]);
        X[0].val[0] = veorq_u8(X[0].val[0], X[1].val[2]);

        X[1].val[3] = veorq_u8(X[1].val[3], X[0].val[0]);
        X[1].val[2] = veorq_u8(X[1].val[2], X[0].val[3]);
        X[1].val[1] = veorq_u8(X[1].val[1], X[0].val[2]);
        X[1].val[0] = veorq_u8(X[1].val[0], X[0].val[1]);
        X[0].val[3] = veorq_u8(X[0].val[3], X[1].val[0]);
        X[0].val[2] = veorq_u8(X[0].val[2], X[1].val[3]);
        X[0].val[1] = veorq_u8(X[0].val[1], X[1].val[2]);
        X[0].val[0] = veorq_u8(X[0].val[0], X[1].val[1]);


        // X[0] and X[1] are swapped now; this is
        // taken into account in the feistel round
}
```

Chapter 4

Evaluation

In this chapter, we will evaluate the strategies through performance measurements and compare them with one another.

4.1 Performance evaluation

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 taking place in the actual encryption function.

ARMv8-A 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. Results can be found in full in Appendix A and are summarized in Figure 4.1.

Bit- and bytesliced implementations leveraging SIMD instructions show the highest throughput values. Camellia implementations tend to be faster than GIFT due to the higher number of rounds as well as the bit permutation slowing down GIFT software implementations, differing from Camellia which is byte-oriented. Bitsliced GIFT manages to be the fastest due to the bit permutation being imple-

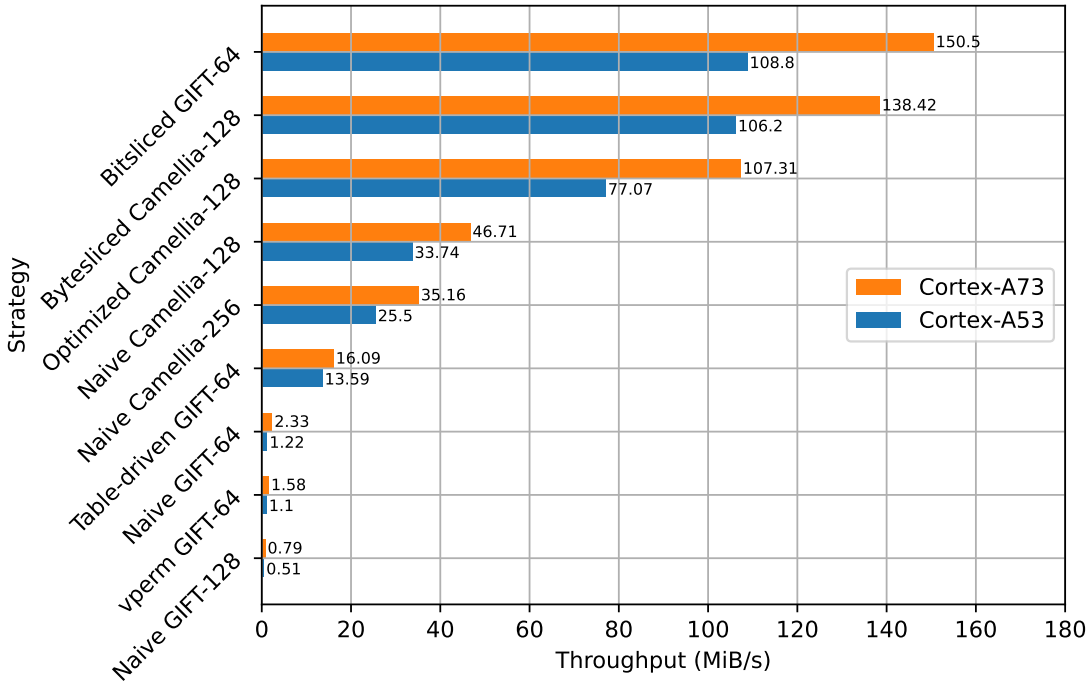

Figure 4.1: Throughput in MiB/s for each strategy and processor type

mented efficiently through bytewise table lookups. Bytesliced Camellia achieves an only slightly lower performance than bitsliced GIFT in spite of increased complexity due to the higher number of bytes being encrypted in parallel.

The claimed performance of GIFT by the original authors of 2.10 cycles per byte utilizing Intel AVX2 is superior to our implementation which only manages to achieve 13.98 cycles per byte [4].

Our Camellia implementation for ARMv8 achieves a latency of around 15.19 cycles per byte, putting it in a similar position as the byte-sliced implementation for x86 utilizing AVX and AES-NI instructions which claims a latency of 5.32 cycles per byte [3] and surpassing the figure of 20.38 cycles per byte of the original Camellia paper [5].

The differences in speed are grounded in varying vector register sizes, instruction timings and implementation details as well as available vector instructions.

Table-driven implementations show a 691% and 230% improvement for GIFT and Camellia respectively compared to their naive implementations. GIFT especially benefits from this approach due to the elimination of the expensive bit permutation layer. While S-box lookup is accelerated for `vperm` GIFT-64, the whole implementation only achieves a throughput of 1.58MiB/s due to extremely inefficient packing and unpacking operations every round.

4.2 Conclusion

We have examined various strategies for efficient implementation of GIFT and Camellia and have realized these in the C programming language using NEON intrinsics for the ARMv8 platform. Bit- and bytesliced implementations have proven themselves especially suited for high-throughput applications on SIMD platforms with the added security benefit of being constant-time, thus eliminating possible side-channels through e.g. cache timings. Table implementations also show good performance characteristics compared to non-table approaches and are extremely easy to port to other platforms since they don't rely on any specific machine instructions, but are vulnerable to cache timing attacks since they access data-dependent memory locations.

GIFT as well as Camellia are suited for low-power hardware as well as bitsliced software implementation with GIFT appearing to be more suited towards hardware implementation due to the low area requirement of the S-box and bit wiring permutation layer.

Vector length determines the number of blocks that can be processed in parallel and therefore is the major contributor to performance since a doubling of register size with the same instruction throughput generally implies a doubling in performance. NEON provides 128 bit registers which is smaller than the 512 bit registers offered by Intel's AVX-512, but new extensions for ARM like SVE2 offer flexible vector lengths up to 2048 bits which will result not only in further acceleration of traditional compute domains like machine learning or multimedia, but also of massively parallelized cryptographic algorithms for low-power and embedded systems.

4.3 Notes

Working on this project has allowed me to learn a lot about ARMv8 and its A64 instruction set as well as SIMD extensions and the use of intrinsics. I have written a lot of code for x86-64, but working with a RISC machine with comparatively simple instructions and addressing modes felt somewhat simpler and more straightforward. The toolchain was surprisingly simple to set up: after flashing a custom odroid Linux image to the SD card, setting up SSH, copying the sysroot (i.e. `/lib` and `/usr`) to my local development machine and feeding clang with the correct parameters for compilation, I could just copy the binary to the target system and run it there. Building, deploying and executing was simple to automate using a Makefile.

Aside from tooling, I also learned a lot about efficient cryptography implementations and how bitslicing can make a seemingly inefficient cipher like GIFT very fast. Most of the time debugging was spent on fixing values for permutation tables since it was easy to make a mistake there, but once they were corrected the rest was usually pretty simple.

Trying to optimize performance without writing assembler code was also very educational since constantly making small adjustments to the C program and then analyzing the disassembled binaries taught me a lot about what the compiler actually does and how it optimizes programs through e.g. loop unrolling and inlining as well as combining logical instructions.

Appendix A

Detailed benchmarking results

A.1 GIFT

Table A.1: Benchmarks for GIFT

		Cortex-A53		Cortex-A73	
Strategy	Component	*lat* (c/B)	*thr* (MiB/s)	*lat* (c/B)	*thr* (MiB/s)
Naive GIFT-64	`round_keys`	223.54	1.22	190.34	2.33
	`encrypt`	1367.66		830.49	
	`subcells`	4.64		3.13	
	`permute`	36.68		21.09	
Naive GIFT-128	`round_keys`	182.91	0.51	167.16	0.79
	`encrypt`	3532.52		2615.89	
	`subcells`	6.91		2.73	
	`permute`	82.68		61.27	
Table-driven	`round_keys`	223.81	13.59	190.11	16.09
	`encrypt`	122.11		119.62	
	`subperm`	5.15		4.47	
`vperm` S-box	`round_keys`	308.12	1.10	270.69	1.58
	`encrypt`	1514.28		1218.18	
	`subcells`	1.63		1.13	
	`permute`	53.18		42.40	
	`pack`	7.51		1.24	
	`unpack`	7.39		4.67	
Bitsliced	`round_keys`	19.93	108.80	16.12	150.50
	`encrypt`	16.69		13.98	
	`subcells`	0.41		0.06	
	`permute`	0.39		0.18	
	`pack`	1.81		1.34	
	`unpack`	1.68		1.31	

A.2 Camellia

Table A.2: Benchmarks for Camellia

		Cortex-A53		Cortex-A73	
Strategy	Component	*lat* (c/B)	*thr* (MiB/s)	*lat* (c/B)	*thr* (MiB/s)
Naive Camellia-128	round_keys	15.26	33.74	11.08	46.71
	encrypt	53.44		43.51	
	feistel_round	4.02		2.66	
	S	2.01		0.94	
	F	3.13		2.25	
	P	2.03		1.63	
	FL	1.06		0.72	
Naive Camellia-256	round_keys	22.01	25.50	16.99	35.16
	encrypt	70.55		58.27	
Optimized Camellia-128	round_keys	8.18	77.07	5.93	107.31
	encrypt	23.63		19.83	
	feistel_round	2.32		1.56	
	F	2.21		1.43	
	FL	1.06		0.69	
Bytesliced Camellia-128	round_keys	2.59	106.20	2.19	138.42
	encrypt	17.14		15.19	
	feistel_round	1.00		0.69	
	F	0.79		0.58	
	FL	0.36		0.12	
	pack	1.04		0.96	
	unpack	0.91		0.83	

Appendix B

List of symbols

$\ll_n$ / $\gg_n$	left/right shift by n bits
$\lll_n$ / $\ggg_n$	left/right rotate by n bits
$\wedge$	AND
$\vee$	OR
$\neg$	NOT
$\oplus$	XOR
$A_{(32)}$	A is 32 bits long
$A_{21} \leftarrow 1$	set bit 21 of A to 1
$\|\|$	concatenation

Appendix C

C source code

C.1 GIFT

C.1.1 Table-based

Listing C.1: `gift/table.h`

```c
#pragma once

#include <stdint.h>

#define ROUNDS_GIFT_64 28

void gift_64_table_generate_round_keys(uint64_t rks[restrict ROUNDS_GIFT_64],
                                       const uint64_t key[restrict 2]);

uint64_t gift_64_table_subperm(const uint64_t cipher_state);

// can only encrypt using table technique!
uint64_t gift_64_table_encrypt(const uint64_t m,
                               const uint64_t rks[restrict ROUNDS_GIFT_64]);
```

Listing C.2: `gift/table.c`

```c
#include <stdint.h>
#include <stddef.h>

#include "table.h"

static const int round_const[] = {
        // rounds 0-15
        0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B,
        0x37, 0x2F, 0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E,
        // rounds 16-31
        0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30,
        0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E, 0x1C, 0x38,
        // rounds 32-47
        0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A,
        0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04
};
```

```
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, 0x0000004000000000UL, 0
               x0080004000000000UL, 0x0000004000200000UL, 0x0080004000200010UL, 0
               x0000000000200010UL, 0x0080000000000010UL, 0x0000000000200000UL, 0
               x0080004000000010UL, 0x0080000000200010UL, 0x0000004000200010UL, 0
               x0000004000000010UL, 0x0000000000000000UL, 0x0080000000000000UL, 0
               x0080004000200000UL },
24         { 0x0010000000000000UL, 0x0000008000000020UL, 0x0000000000400000UL, 0
               x0000008000400000UL, 0x0000000000400020UL, 0x0010008000400020UL, 0
               x0010000000000020UL, 0x0010008000000000UL, 0x0000000000000020UL, 0
               x0010008000400000UL, 0x0010008000000020UL, 0x0010000000400020UL, 0
               x0010000000400000UL, 0x0000000000000000UL, 0x0000008000000000UL, 0
               x0000008000400020UL },
25         { 0x0000001000000000UL, 0x0020000000800000UL, 0x0000000000000040UL, 0
               x0000000000800040UL, 0x0020000000000040UL, 0x0020001000800040UL, 0
               x0020001000000000UL, 0x0000001000800000UL, 0x0020000000000000UL, 0
               x0000001000800040UL, 0x0020001000800000UL, 0x0020001000000040UL, 0
               x0000001000000040UL, 0x0000000000000000UL, 0x0000000000800000UL, 0
               x0020000000800040UL },
26         { 0x0000000001000000UL, 0x0000002000000080UL, 0x0040000000000000UL, 0
               x0040000000000080UL, 0x0040002000000000UL, 0x0040002000100080UL, 0
               x0000002000100000UL, 0x0000000000100080UL, 0x0000002000000000UL, 0
               x0040000000100080UL, 0x0000002000100080UL, 0x0040002000100000UL, 0
               x0040000000100000UL, 0x0000000000000000UL, 0x0000000000000080UL, 0
               x0040002000000080UL },
27         { 0x0000000000000100UL, 0x0800000002000000UL, 0x0000040000000000UL, 0
               x0800040000000000UL, 0x0000040002000000UL, 0x0800040002000100UL, 0
               x0000000002000100UL, 0x0800000000000100UL, 0x0000000002000000UL, 0
               x0800040000000100UL, 0x0800000002000100UL, 0x0000040002000100UL, 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
                x0000100080000400UL, 0x0200100080000000UL, 0x0200100000000400UL, 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          { 0x0001000000000000UL, 0x2000000080000000UL, 0x0000000000004000UL, 0
                x0000000080004000UL, 0x2000000000004000UL, 0x2001000080004000UL, 0
                x2001000000000000UL, 0x0001000080000000UL, 0x2000000000000000UL, 0
                x0001000080004000UL, 0x2001000080000000UL, 0x2001000000004000UL, 0
                x0001000000004000UL, 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 }
```

C.1.2 Using `vperm`

Listing C.3: `gift/vec_sbox.h`

```
1  #pragma once
2
3  #include <arm_neon.h>
4  #include <stdint.h>
5
6  #define ROUNDS_GIFT_64 28
7
8  // expose for benchmarking
9  uint8x16_t gift_64_vec_sbox_bits_pack(const uint64_t a);
10 uint64_t   gift_64_vec_sbox_bits_unpack(const uint8x16_t a);
11 uint8x16_t gift_64_vec_sbox_subcells(const uint8x16_t cipher_state);
12 uint8x16_t gift_64_vec_sbox_subcells_inv(const uint8x16_t cipher_state);
```

```
13 uint8x16_t gift_64_vec_sbox_permute(const uint8x16_t cipher_state);
14 uint8x16_t gift_64_vec_sbox_permute_inv(const uint8x16_t cipher_state);
15 void       gift_64_vec_sbox_generate_round_keys(uint8x16_t rks[ROUNDS_GIFT_64],
16                                                 const uint64_t key[restrict 2]);
17
18 // construct tables
19 void gift_64_vec_sbox_init(void);
20
21 uint64_t gift_64_vec_sbox_encrypt(const uint64_t m,
22                                   const uint8x16_t rks[restrict ROUNDS_GIFT_64]);
23 uint64_t gift_64_vec_sbox_decrypt(const uint64_t c,
24                                   const uint8x16_t rks[restrict ROUNDS_GIFT_64]);
```

Listing C.4: `gift/vec_sbox.c`

```
1  #include <arm_neon.h>
2  #include <stdint.h>
3  #include <stddef.h>
4
5  #include "vec_sbox.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         uint64_t lane = vgetq_lane_u64(v, 0);
50         a = (uint64_t)((lane >> 8 * 0) & 0xf) << 4 * 0 |
51             (uint64_t)((lane >> 8 * 1) & 0xf) << 4 * 1 |
52             (uint64_t)((lane >> 8 * 2) & 0xf) << 4 * 2 |
53             (uint64_t)((lane >> 8 * 3) & 0xf) << 4 * 3 |
54             (uint64_t)((lane >> 8 * 4) & 0xf) << 4 * 4 |
55             (uint64_t)((lane >> 8 * 5) & 0xf) << 4 * 5 |
56             (uint64_t)((lane >> 8 * 6) & 0xf) << 4 * 6 |
57             (uint64_t)((lane >> 8 * 7) & 0xf) << 4 * 7;
58
59         lane = vgetq_lane_u64(v, 1);
60         a |= (uint64_t)((lane >> 8 * 0) & 0xf) << 4 * 8  |
61              (uint64_t)((lane >> 8 * 1) & 0xf) << 4 * 9  |
62              (uint64_t)((lane >> 8 * 2) & 0xf) << 4 * 10 |
63              (uint64_t)((lane >> 8 * 3) & 0xf) << 4 * 11 |
64              (uint64_t)((lane >> 8 * 4) & 0xf) << 4 * 12 |
65              (uint64_t)((lane >> 8 * 5) & 0xf) << 4 * 13 |
66              (uint64_t)((lane >> 8 * 6) & 0xf) << 4 * 14 |
67              (uint64_t)((lane >> 8 * 7) & 0xf) << 4 * 15;
68
69         return a;
70 }
71
72 static const size_t perm_64[] = {
73         0, 17, 34, 51, 48, 1, 18, 35, 32, 49, 2, 19, 16, 33, 50, 3,
74         4, 21, 38, 55, 52, 5, 22, 39, 36, 53, 6, 23, 20, 37, 54, 7,
75         8, 25, 42, 59, 56, 9, 26, 43, 40, 57, 10, 27, 24, 41, 58, 11,
76         12, 29, 46, 63, 60, 13, 30, 47, 44, 61, 14, 31, 28, 45, 62, 15
77 };
78
79 static const size_t perm_64_inv[] = {
80         0, 5, 10, 15, 16, 21, 26, 31, 32, 37, 42, 47, 48, 53, 58, 63,
81         12, 1, 6, 11, 28, 17, 22, 27, 44, 33, 38, 43, 60, 49, 54, 59,
82         8, 13, 2, 7, 24, 29, 18, 23, 40, 45, 34, 39, 56, 61, 50, 55,
83         4, 9, 14, 3, 20, 25, 30, 19, 36, 41, 46, 35, 52, 57, 62, 51
84 };
85
86 static const int round_const[] = {
87         // rounds 0-15
88         0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F, 0x1E, 0x3C, 0x39, 0
               x33, 0x27, 0x0E,
89         // rounds 16-31
90         0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0
               x2E, 0x1C, 0x38,
91         // rounds 32-47
92         0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A, 0x34, 0x29, 0x12, 0x24, 0x08, 0
               x11, 0x22, 0x04
93 };
94
95 uint8x16_t gift_64_vec_sbox_subcells(const uint8x16_t cipher_state)
96 {
97         return vqtbl1q_u8(sbox_vec, cipher_state);
98 }
99
100 uint8x16_t gift_64_vec_sbox_subcells_inv(const uint8x16_t cipher_state)
101 {
102         return vqtbl1q_u8(sbox_vec_inv, cipher_state);
103 }
104
105 uint8x16_t gift_64_vec_sbox_permute(const uint8x16_t cipher_state)
```

```
106 {
107         // collect individual bits into 64-bit register
108         uint64_t new_cipher_state = 0UL;
109         uint64_t boxes[2];
110         vst1q_u64(boxes, cipher_state);
111 
112         for (size_t box = 0; box < 16; box++) {
113                 for (size_t i = 0; i < 4; i++) {
114                         const int bit = (boxes[box / 8] >> ((box % 8) * 8 + i)) & 0x1;
115                         new_cipher_state |= (uint64_t)bit << perm_64[box * 4 + i];
116                 }
117         }
118 
119         return gift_64_vec_sbox_bits_pack(new_cipher_state);
120 }
121 
122 uint8x16_t gift_64_vec_sbox_permute_inv(const uint8x16_t cipher_state)
123 {
124         // collect into 64-bit register (faster)
125         uint64_t new_cipher_state = 0;
126         uint64_t boxes[2];
127         vst1q_u64(boxes, cipher_state);
128 
129         // S-box 0-7
130         for (size_t box = 0; box < 8; box++) {
131                 for (size_t i = 0; i < 4; i++) {
132                         const int bit = (boxes[0] >> (box * 8 + i)) & 0x1;
133                         new_cipher_state |= (uint64_t)bit << perm_64_inv[box * 4 + i];
134                 }
135         }
136 
137         // S-box 8-15
138         for (size_t box = 0; box < 8; box++) {
139                 for (size_t i = 0; i < 4; i++) {
140                         const int bit = (boxes[1] >> (box * 8 + i)) & 0x1;
141                         new_cipher_state |= (uint64_t)bit << perm_64_inv[(box + 8) * 4
                                + i];
142                 }
143         }
144 
145         return gift_64_vec_sbox_bits_pack(new_cipher_state);
146 }
147 
148 void gift_64_vec_sbox_generate_round_keys(uint8x16_t rks[ROUNDS_GIFT_64],
149                                           const uint64_t key[2])
150 {
151         uint64_t key_state[] = {key[0], key[1]};
152         for (int round = 0; round < ROUNDS_GIFT_64; round++) {
153                 const int v = (key_state[0] >> 0 ) & 0xffff;
154                 const int u = (key_state[0] >> 16) & 0xffff;
155 
156                 // add round key (RK=U||V)
157                 uint64_t round_key = 0UL;
158                 for (size_t i = 0; i < 16; i++) {
159                         const int key_bit_v   = (v >> i)  & 0x1;
160                         const int key_bit_u   = (u >> i)  & 0x1;
161                         round_key ^= (uint64_t)key_bit_v << (i * 4 + 0);
162                         round_key ^= (uint64_t)key_bit_u << (i * 4 + 1);
163                 }
164 
165                 // add single bit
166                 round_key ^= 1UL << 63;
```

```
167
168                 // add round constants
169                 round_key ^= ((round_const[round] >> 0) & 0x1) << 3;
170                 round_key ^= ((round_const[round] >> 1) & 0x1) << 7;
171                 round_key ^= ((round_const[round] >> 2) & 0x1) << 11;
172                 round_key ^= ((round_const[round] >> 3) & 0x1) << 15;
173                 round_key ^= ((round_const[round] >> 4) & 0x1) << 19;
174                 round_key ^= ((round_const[round] >> 5) & 0x1) << 23;
175
176                 // pack into vector register
177                 rks[round] = gift_64_vec_sbox_bits_pack(round_key);
178
179                 // update key state
180                 int k0 = (key_state[0] >> 0 ) & 0xffffUL;
181                 int k1 = (key_state[0] >> 16) & 0xffffUL;
182                 k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
183                 k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
184                 key_state[0] >>= 32;
185                 key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
186                 key_state[1] >>= 32;
187                 key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
188         }
189 }
190
191 void gift_64_vec_sbox_init(void)
192 {
193         // construct sbox_vec
194         sbox_vec = vld1q_u64(sbox_vec_u64);
195
196         // construct sbox_vec_inv
197         sbox_vec_inv = vld1q_u64(sbox_vec_inv_u64);
198 }
199
200 uint64_t gift_64_vec_sbox_encrypt(const uint64_t m,
201                                   const uint8x16_t rks[restrict ROUNDS_GIFT_64])
202 {
203         // pack into vector register
204         uint8x16_t c = gift_64_vec_sbox_bits_pack(m);
205
206         // round loop
207         for (int round = 0; round < ROUNDS_GIFT_64; round++) {
208                 c = gift_64_vec_sbox_subcells(c);
209                 c = gift_64_vec_sbox_permute(c);
210                 c = veorq_u8(c, rks[round]);
211         }
212
213         // unpack
214         return gift_64_vec_sbox_bits_unpack(c);
215 }
216
217 uint64_t gift_64_vec_sbox_decrypt(const uint64_t c,
218                                   const uint8x16_t rks[restrict ROUNDS_GIFT_64])
219 {
220         // pack into vector register
221         uint8x16_t m = gift_64_vec_sbox_bits_pack(c);
222
223         // round loop (in reverse)
224         for (int round = ROUNDS_GIFT_64 - 1; round >= 0; round--) {
225                 m = veorq_u8(m, rks[round]);
226                 m = gift_64_vec_sbox_permute_inv(m);
227                 m = gift_64_vec_sbox_subcells_inv(m);
228         }
```

```
229
230         // unpack
231         return gift_64_vec_sbox_bits_unpack(m);
232 }
```

C.1.3 Bitslicing

Listing C.5: gift/vec_sliced.h

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

Listing C.6: gift/vec_sliced.c

```
1  #include <arm_neon.h>
2  #include <stdint.h>
3  #include <stddef.h>
4
5  #include "vec_sliced.h"
6
7  static uint64_t pack_shf_u64[] = {
8          0x1303120211011000UL, 0x1707160615051404UL, // S0/S1/S2/S3
9          0x1b0b1a0a19091808UL, 0x1f0f1e0e1d0d1c0cUL, // S4/S5/S6/S7
10 };
11
12 static uint64_t pack_shf_inv_u64[] = {
13         0x0e0c0a0806040200UL, 0x1e1c1a1816141210UL, // S0/S1/S2/S3
14         0x0f0d0b0907050301UL, 0x1f1d1b1917151311UL, // S4/S5/S6/S7
15 };
```

```
16
17 static uint64_t perm_u64[] = {
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[] = {
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 /*
51 uint8x16_t shl(const uint8x16_t v, const int n) */
52 #define shl(_a, v, n)                                                          \
53 {                                                                              \
54         uint64x2_t _overflow = vshrq_n_u64(v, 64 - n);                       \
55         _overflow = vextq_u64(vdupq_n_u64(0x0), _overflow, 1);               \
56         _a = vorrq_u8(vshlq_n_u64(v, n), _overflow);                         \
57 }
58
59 /*
60 uint8x16_t shr(const uint8x16_t v, const int n) */
61 #define shr(_a, v, n)                                                          \
62 {                                                                              \
63         uint64x2_t _overflow = vshlq_n_u64(v, 64 - n);                       \
64         _overflow = vextq_u64(_overflow, vdupq_n_u64(0x0), 1);               \
65         _a = vorrq_u8(vshrq_n_u64(v, n), _overflow);                         \
66 }
67
68 /* implemented as a macro so we can use vshlq_n_u8 with variable n
69 void gift_64_vec_sliced_swapmove(uint8x16_t *restrict a, uint8x16_t *restrict b,
70                                  const uint8x16_t m, const int n) */
71 #define gift_64_vec_sliced_swapmove(a, b, m, n)                                \
72 {                                                                              \
73         uint8x16_t _a;                                                       \
74         shr(_a, a, n);                                                       \
```

```
75          const uint8x16_t _t = vandq_u8(veorq_u8(_a, b), m);                 \
76          b = veorq_u8(b, _t);                                                \
77          shl(_a, _t, n);                                                     \
78          a = veorq_u8(a, _a);                                                \
79  }
80
81  void gift_64_vec_sliced_bits_pack(uint8x16x4_t m[restrict 2])
82  {
83          // take care not to shift mask bits out of the register
84          gift_64_vec_sliced_swapmove(m[0].val[0], m[0].val[1], pack_mask_0, 1);
85          gift_64_vec_sliced_swapmove(m[0].val[2], m[0].val[3], pack_mask_0, 1);
86          gift_64_vec_sliced_swapmove(m[1].val[0], m[1].val[1], pack_mask_0, 1);
87          gift_64_vec_sliced_swapmove(m[1].val[2], m[1].val[3], pack_mask_0, 1);
88
89          gift_64_vec_sliced_swapmove(m[0].val[0], m[0].val[2], pack_mask_1, 2);
90          gift_64_vec_sliced_swapmove(m[0].val[1], m[0].val[3], pack_mask_1, 2);
91          gift_64_vec_sliced_swapmove(m[1].val[0], m[1].val[2], pack_mask_1, 2);
92          gift_64_vec_sliced_swapmove(m[1].val[1], m[1].val[3], pack_mask_1, 2);
93
94          // make bytes (a0 b0 c0 d0 a4 b4 c4 d4 -> a0 b0 c0 d0 e0 f0 g0 h0)
95          gift_64_vec_sliced_swapmove(m[0].val[0], m[1].val[0], pack_mask_2, 4);
96          gift_64_vec_sliced_swapmove(m[0].val[2], m[1].val[2], pack_mask_2, 4);
97          gift_64_vec_sliced_swapmove(m[0].val[1], m[1].val[1], pack_mask_2, 4);
98          gift_64_vec_sliced_swapmove(m[0].val[3], m[1].val[3], pack_mask_2, 4);
99
100         // same plaintext slice bits into same register (so we only have to do
101         // what we are doing here once instead of every round)
102         const uint8x16x2_t pairs[4] = {
103                 { .val = { m[0].val[0], m[1].val[0] }},
104                 { .val = { m[0].val[1], m[1].val[1] }},
105                 { .val = { m[0].val[2], m[1].val[2] }},
106                 { .val = { m[0].val[3], m[1].val[3] }},
107         };
108
109         m[0].val[0] = vqtbl2q_u8(pairs[0], pack_shf.val[0]);
110         m[0].val[1] = vqtbl2q_u8(pairs[1], pack_shf.val[0]);
111         m[0].val[2] = vqtbl2q_u8(pairs[2], pack_shf.val[0]);
112         m[0].val[3] = vqtbl2q_u8(pairs[3], pack_shf.val[0]);
113
114         m[1].val[0] = vqtbl2q_u8(pairs[0], pack_shf.val[1]);
115         m[1].val[1] = vqtbl2q_u8(pairs[1], pack_shf.val[1]);
116         m[1].val[2] = vqtbl2q_u8(pairs[2], pack_shf.val[1]);
117         m[1].val[3] = vqtbl2q_u8(pairs[3], pack_shf.val[1]);
118 }
119
120 void gift_64_vec_sliced_bits_unpack(uint8x16x4_t m[restrict 2])
121 {
122         const uint8x16x2_t pairs[4] = {
123                 { .val = { m[0].val[0], m[1].val[0] }},
124                 { .val = { m[0].val[1], m[1].val[1] }},
125                 { .val = { m[0].val[2], m[1].val[2] }},
126                 { .val = { m[0].val[3], m[1].val[3] }},
127         };
128
129         m[0].val[0] = vqtbl2q_u8(pairs[0], pack_shf_inv.val[0]);
130         m[0].val[1] = vqtbl2q_u8(pairs[1], pack_shf_inv.val[0]);
131         m[0].val[2] = vqtbl2q_u8(pairs[2], pack_shf_inv.val[0]);
132         m[0].val[3] = vqtbl2q_u8(pairs[3], pack_shf_inv.val[0]);
133
134         m[1].val[0] = vqtbl2q_u8(pairs[0], pack_shf_inv.val[1]);
135         m[1].val[1] = vqtbl2q_u8(pairs[1], pack_shf_inv.val[1]);
136         m[1].val[2] = vqtbl2q_u8(pairs[2], pack_shf_inv.val[1]);
```

```
137         m[1].val[3] = vqtbl2q_u8(pairs[3], pack_shf_inv.val[1]);
138
139         // take care not to shift mask bits out of the register
140         gift_64_vec_sliced_swapmove(m[0].val[0], m[0].val[1], pack_mask_0, 1);
141         gift_64_vec_sliced_swapmove(m[0].val[2], m[0].val[3], pack_mask_0, 1);
142         gift_64_vec_sliced_swapmove(m[1].val[0], m[1].val[1], pack_mask_0, 1);
143         gift_64_vec_sliced_swapmove(m[1].val[2], m[1].val[3], pack_mask_0, 1);
144
145         gift_64_vec_sliced_swapmove(m[0].val[0], m[0].val[2], pack_mask_1, 2);
146         gift_64_vec_sliced_swapmove(m[0].val[1], m[0].val[3], pack_mask_1, 2);
147         gift_64_vec_sliced_swapmove(m[1].val[0], m[1].val[2], pack_mask_1, 2);
148         gift_64_vec_sliced_swapmove(m[1].val[1], m[1].val[3], pack_mask_1, 2);
149
150         // make bytes (a0 b0 c0 d0 a4 b4 c4 d4 -> a0 b0 c0 d0 e0 f0 g0 h0)
151         gift_64_vec_sliced_swapmove(m[0].val[0], m[1].val[0], pack_mask_2, 4);
152         gift_64_vec_sliced_swapmove(m[0].val[2], m[1].val[2], pack_mask_2, 4);
153         gift_64_vec_sliced_swapmove(m[0].val[1], m[1].val[1], pack_mask_2, 4);
154         gift_64_vec_sliced_swapmove(m[0].val[3], m[1].val[3], pack_mask_2, 4);
155 }
156
157 void gift_64_vec_sliced_subcells(uint8x16x4_t cs[restrict 2])
158 {
159         cs[0].val[1] = veorq_u8(cs[0].val[1],
160                                 vandq_u8(cs[0].val[0], cs[0].val[2]));
161         uint8x16_t t = veorq_u8(cs[0].val[0],
162                                 vandq_u8(cs[0].val[1], cs[0].val[3]));
163         cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
164         cs[0].val[0] = veorq_u8(cs[0].val[3], cs[0].val[2]);
165         cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
166         cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
167         cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
168         cs[0].val[3] = t;
169
170         cs[1].val[1] = veorq_u8(cs[1].val[1],
171                                 vandq_u8(cs[1].val[0], cs[1].val[2]));
172         t            = veorq_u8(cs[1].val[0],
173                                 vandq_u8(cs[1].val[1], cs[1].val[3]));
174         cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
175         cs[1].val[0] = veorq_u8(cs[1].val[3], cs[1].val[2]);
176         cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
177         cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
178         cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
179         cs[1].val[3] = t;
180 }
181
182 void gift_64_vec_sliced_subcells_inv(uint8x16x4_t cs[restrict 2])
183 {
184         uint8x16_t t = cs[0].val[3];
185         cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
186         cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
187         cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
188         cs[0].val[3] = veorq_u8(cs[0].val[0], cs[0].val[2]);
189         cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
190         cs[0].val[0] = veorq_u8(t, vandq_u8(cs[0].val[1], cs[0].val[3]));
191         cs[0].val[1] = veorq_u8(cs[0].val[1],
192                                 vandq_u8(cs[0].val[0], cs[0].val[2]));
193
194         t            = cs[1].val[3];
195         cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
196         cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
197         cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
198         cs[1].val[3] = veorq_u8(cs[1].val[0], cs[1].val[2]);
```

```
199         cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
200         cs[1].val[0] = veorq_u8(t, vandq_u8(cs[1].val[1], cs[1].val[3]));
201         cs[1].val[1] = veorq_u8(cs[1].val[1],
202                                 vandq_u8(cs[1].val[0], cs[1].val[2]));
203 }
204
205 void gift_64_vec_sliced_permute(uint8x16x4_t cs[restrict 2])
206 {
207         cs[0].val[0] = vqtbl1q_u8(cs[0].val[0], perm.val[0]);
208         cs[0].val[1] = vqtbl1q_u8(cs[0].val[1], perm.val[1]);
209         cs[0].val[2] = vqtbl1q_u8(cs[0].val[2], perm.val[2]);
210         cs[0].val[3] = vqtbl1q_u8(cs[0].val[3], perm.val[3]);
211
212         cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm.val[0]);
213         cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm.val[1]);
214         cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm.val[2]);
215         cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm.val[3]);
216 }
217
218 void gift_64_vec_sliced_permute_inv(uint8x16x4_t cs[restrict 2])
219 {
220         cs[0].val[0] = vqtbl1q_u8(cs[0].val[0], perm_inv.val[0]);
221         cs[0].val[1] = vqtbl1q_u8(cs[0].val[1], perm_inv.val[1]);
222         cs[0].val[2] = vqtbl1q_u8(cs[0].val[2], perm_inv.val[2]);
223         cs[0].val[3] = vqtbl1q_u8(cs[0].val[3], perm_inv.val[3]);
224
225         cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm_inv.val[0]);
226         cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm_inv.val[1]);
227         cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm_inv.val[2]);
228         cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm_inv.val[3]);
229 }
230
231 void gift_64_vec_sliced_generate_round_keys(uint8x16x4_t rks[restrict ROUNDS_GIFT_64
        ][2],
232                                             const uint64_t key[restrict 2])
233 {
234         uint64_t key_state[] = {key[0], key[1]};
235         for (int round = 0; round < ROUNDS_GIFT_64; round++) {
236                 const int v = (key_state[0] >> 0 ) & 0xffff;
237                 const int u = (key_state[0] >> 16) & 0xffff;
238
239                 // add round key (RK=U||V)
240                 // (slice 2 stays unused)
241                 uint64_t rk[6] = { 0x0UL };
242                 for (size_t i = 0; i < 8; i++) {
243                         int key_bit_v   = (v >> (i + 0)) & 0x1;
244                         int key_bit_u   = (u >> (i + 0)) & 0x1;
245                         rk[0]           ^= (uint64_t)key_bit_v << (i * 8);
246                         rk[2]           ^= (uint64_t)key_bit_u << (i * 8);
247
248                         key_bit_v       = (v >> (i + 8)) & 0x1;
249                         key_bit_u       = (u >> (i + 8)) & 0x1;
250                         rk[1]           ^= (uint64_t)key_bit_v << (i * 8);
251                         rk[3]           ^= (uint64_t)key_bit_u << (i * 8);
252                 }
253
254                 // add single bit
255                 rk[5] ^= 1UL << (7 * 8);
256
257                 // add round constants
258                 rk[4] ^= ((uint64_t)(round_const[round] >> 0) & 0x1) << (0 * 8);
259                 rk[4] ^= ((uint64_t)(round_const[round] >> 1) & 0x1) << (1 * 8);
```

```
260                 rk[4] ^= ((uint64_t)(round_const[round] >> 2) & 0x1) << (2 * 8);
261                 rk[4] ^= ((uint64_t)(round_const[round] >> 3) & 0x1) << (3 * 8);
262                 rk[4] ^= ((uint64_t)(round_const[round] >> 4) & 0x1) << (4 * 8);
263                 rk[4] ^= ((uint64_t)(round_const[round] >> 5) & 0x1) << (5 * 8);
264
265                 // extend bits to bytes
266                 for (size_t i = 0; i < 6; i++) {
267                         rk[i] |= rk[i] << 1;
268                         rk[i] |= rk[i] << 2;
269                         rk[i] |= rk[i] << 4;
270                 }
271
272                 rks[round][0].val[0] = vsetq_lane_u64(rk[0], rks[round][0].val[0], 0);
273                 rks[round][0].val[0] = vsetq_lane_u64(rk[1], rks[round][0].val[0], 1);
274                 rks[round][0].val[1] = vsetq_lane_u64(rk[2], rks[round][0].val[1], 0);
275                 rks[round][0].val[1] = vsetq_lane_u64(rk[3], rks[round][0].val[1], 1);
276                 rks[round][0].val[2] = vdupq_n_u8(0);
277                 rks[round][0].val[3] = vsetq_lane_u64(rk[4], rks[round][0].val[3], 0);
278                 rks[round][0].val[3] = vsetq_lane_u64(rk[5], rks[round][0].val[3], 1);
279                 rks[round][1]        = rks[round][0];
280
281                 // update key state
282                 int k0 = (key_state[0] >> 0 ) & 0xffffUL;
283                 int k1 = (key_state[0] >> 16) & 0xffffUL;
284                 k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
285                 k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
286                 key_state[0] >>= 32;
287                 key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
288                 key_state[1] >>= 32;
289                 key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
290         }
291 }
292
293 void gift_64_vec_sliced_init(void)
294 {
295         // bit packing shuffle
296         pack_shf = vld1q_u8_x2((uint8_t*)&pack_shf_u64[0]);
297
298         // inverse bit packing shuffle
299         pack_shf_inv = vld1q_u8_x2((uint8_t*)&pack_shf_inv_u64[0]);
300
301         // permutations
302         perm = vld1q_u8_x4((uint8_t*)&perm_u64[0]);
303
304         // inverse permutations
305         perm_inv = vld1q_u8_x4((uint8_t*)&perm_inv_u64[0]);
306
307         // packing masks
308         pack_mask_0 = vdupq_n_u8(0x55);
309         pack_mask_1 = vdupq_n_u8(0x33);
310         pack_mask_2 = vdupq_n_u8(0x0f);
311 }
312
313 void gift_64_vec_sliced_encrypt(uint64_t c[restrict 16],
314                                 const uint64_t m[restrict 16],
315                                 const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2])
316 {
317         uint8x16x4_t s[2];
318         s[0] = vld1q_u8_x4((uint8_t*)&m[0]);
319         s[1] = vld1q_u8_x4((uint8_t*)&m[8]);
320         gift_64_vec_sliced_bits_pack(s);
321
```

```
322         for (int round = 0; round < ROUNDS_GIFT_64; round++) {
323                 gift_64_vec_sliced_subcells(s);
324                 gift_64_vec_sliced_permute(s);
325 
326                 // round key addition
327                 s[0].val[0] = veorq_u8(s[0].val[0], rks[round][0].val[0]);
328                 s[0].val[1] = veorq_u8(s[0].val[1], rks[round][0].val[1]);
329                 s[0].val[2] = veorq_u8(s[0].val[2], rks[round][0].val[2]);
330                 s[0].val[3] = veorq_u8(s[0].val[3], rks[round][0].val[3]);
331                 s[1].val[0] = veorq_u8(s[1].val[0], rks[round][1].val[0]);
332                 s[1].val[1] = veorq_u8(s[1].val[1], rks[round][1].val[1]);
333                 s[1].val[2] = veorq_u8(s[1].val[2], rks[round][1].val[2]);
334                 s[1].val[3] = veorq_u8(s[1].val[3], rks[round][1].val[3]);
335         }
336 
337         gift_64_vec_sliced_bits_unpack(s);
338         vst1q_u8_x4((uint8_t*)&c[0], s[0]);
339         vst1q_u8_x4((uint8_t*)&c[8], s[1]);
340 }
341 
342 void gift_64_vec_sliced_decrypt(uint64_t m[restrict 16],
343                                 const uint64_t c[restrict 16],
344                                 const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2])
345 {
346         uint8x16x4_t s[2];
347         s[0] = vld1q_u8_x4((uint8_t*)&c[0]);
348         s[1] = vld1q_u8_x4((uint8_t*)&c[8]);
349         gift_64_vec_sliced_bits_pack(s);
350 
351         for (int round = ROUNDS_GIFT_64 - 1; round >= 0; round--) {
352                 // round key addition
353                 s[0].val[0] = veorq_u8(s[0].val[0], rks[round][0].val[0]);
354                 s[0].val[1] = veorq_u8(s[0].val[1], rks[round][0].val[1]);
355                 s[0].val[2] = veorq_u8(s[0].val[2], rks[round][0].val[2]);
356                 s[0].val[3] = veorq_u8(s[0].val[3], rks[round][0].val[3]);
357                 s[1].val[0] = veorq_u8(s[1].val[0], rks[round][1].val[0]);
358                 s[1].val[1] = veorq_u8(s[1].val[1], rks[round][1].val[1]);
359                 s[1].val[2] = veorq_u8(s[1].val[2], rks[round][1].val[2]);
360                 s[1].val[3] = veorq_u8(s[1].val[3], rks[round][1].val[3]);
361 
362                 gift_64_vec_sliced_permute_inv(s);
363                 gift_64_vec_sliced_subcells_inv(s);
364         }
365 
366         gift_64_vec_sliced_bits_unpack(s);
367         vst1q_u8_x4((uint8_t*)&m[0], s[0]);
368         vst1q_u8_x4((uint8_t*)&m[8], s[1]);
369 }
```

C.2 Camellia

C.2.1 Optimized non-SIMD

Listing C.7: `camellia/spec_opt.h`

```
1 #pragma once
2 
```

```
3  // 128-bit and 256-bit Camellia
4
5  #include <stdint.h>
6
7  #include "camellia_keys.h"
8
9  uint64_t camellia_spec_opt_F(uint64_t X, const uint64_t k);
10 uint64_t camellia_spec_opt_FL(uint64_t X, const uint64_t kl);
11 uint64_t camellia_spec_opt_FL_inv(uint64_t X, const uint64_t kl);
12 void camellia_spec_opt_feistel_round(uint64_t state[2], const uint64_t kr);
13 void camellia_spec_opt_feistel_round_inv(uint64_t state[2], const uint64_t kr);
14 void camellia_spec_opt_generate_round_keys_128(struct camellia_rks_128 *restrict rks,
15                                                const uint64_t key[restrict 2]);
16
17 void camellia_spec_opt_encrypt_128(uint64_t c[restrict 2],
18                                    const uint64_t m[restrict 2],
19                                    const struct camellia_rks_128 *restrict rks);
20
21 void camellia_spec_opt_decrypt_128(uint64_t m[restrict 2],
22                                    const uint64_t c[restrict 2],
23                                    const struct camellia_rks_128 *restrict rks);
```

Listing C.8: `camellia/spec_opt.c`

```
1  #include <stdint.h>
2  #include <stddef.h>
3  #include <string.h>
4
5  #include "spec_opt.h"
6  #include "spec_opt_table.h"
7
8  static const uint64_t keysched_const[] = {
9          0xa09e667f3bcc908bUL,
10         0xb67ae8584caa73b2UL,
11         0xc6ef372fe94f82beUL,
12         0x54ff53a5f1d36f1cUL,
13         0x10e527fade682d1dUL,
14         0xb05688c2b3e6c1fdUL
15 };
16
17 // rotates for values larger than 64 are done by swapping array elements
18 #define rol128(_a, _b, _n){\
19         uint64_t _t = _b;\
20         (_b) = ((_b) << (_n)) | ((_a) >> (64 - (_n)));\
21         (_a) = ((_a) << (_n)) | ((_t) >> (64 - (_n)));\
22 }
23
24 static uint32_t rol32_1(uint32_t a)
25 {
26         return (a << 1) | (a >> 31);
27 }
28
29 uint64_t camellia_spec_opt_F(uint64_t X, const uint64_t k)
30 {
31         X ^= k;
32
33         // compute P(S(X)) through large 64-bit lookup table
34         uint64_t result = 0UL;
35         result ^= SP0[(X >> 56) & 0xff];
36         result ^= SP1[(X >> 48) & 0xff];
37         result ^= SP2[(X >> 40) & 0xff];
```

```
38          result ^= SP3[(X >> 32) & 0xff];
39          result ^= SP4[(X >> 24) & 0xff];
40          result ^= SP5[(X >> 16) & 0xff];
41          result ^= SP6[(X >> 8 ) & 0xff];
42          result ^= SP7[(X >> 0 ) & 0xff];
43
44          return result;
45  }
46
47  uint64_t camellia_spec_opt_FL(uint64_t X, const uint64_t kl)
48  {
49          const uint32_t XL = (X >> 32);
50          const uint32_t XR = (X >> 0);
51
52          const uint32_t klL = (kl >> 32);
53          const uint32_t klR = (kl >> 0);
54
55          const uint32_t YR = rol32_1(XL & klL) ^ XR;
56          const uint32_t YL = (YR | klR) ^ XL;
57
58          return ((uint64_t)YL << 32) | (uint64_t)YR;
59  }
60
61  uint64_t camellia_spec_opt_FL_inv(uint64_t Y, const uint64_t kl)
62  {
63          const uint32_t YL = (Y >> 32);
64          const uint32_t YR = (Y >> 0);
65
66          const uint32_t klL = (kl >> 32);
67          const uint32_t klR = (kl >> 0);
68
69          const uint32_t XL = (YR | klR) ^ YL;
70          const uint32_t XR = rol32_1(XL & klL) ^ YR;
71
72          return ((uint64_t)XL << 32) | (uint64_t)XR;
73  }
74
75  void camellia_spec_opt_feistel_round(uint64_t state[2], const uint64_t kr)
76  {
77          const uint64_t Lr = state[1] ^ camellia_spec_opt_F(state[0], kr);
78          state[1] = state[0];
79          state[0] = Lr;
80  }
81
82  void camellia_spec_opt_feistel_round_inv(uint64_t state[2], const uint64_t kr)
83  {
84          const uint64_t Rr = state[0] ^ camellia_spec_opt_F(state[1], kr);
85          state[0] = state[1];
86          state[1] = Rr;
87  }
88
89  void camellia_spec_opt_generate_round_keys_128(struct camellia_rks_128 *restrict rks,
90                                                 const uint64_t key[restrict 2])
91  {
92          uint64_t KL[2], KA[2];
93
94          // compute KL
95          memcpy(KL, key, sizeof(KL));
96
97          // compute KA
98          memcpy(KA, KL, sizeof(KA));
99
```

```
100         KA[0] = camellia_spec_opt_F(KL[1] ^ camellia_spec_opt_F(KL[0], keysched_const
                [0]),
101                                     keysched_const[1]);
102         KA[1] = camellia_spec_opt_F(KL[0], keysched_const[0]);
103         camellia_spec_opt_feistel_round(KA, keysched_const[2]);
104         camellia_spec_opt_feistel_round(KA, keysched_const[3]);
105
106         struct camellia_rks_128 keys;
107
108         // KL-dependent subkeys
109         keys.kw[0] = KL[0]; keys.kw[1] = KL[1];
110         rol128(KL[0], KL[1], 15); // KL << 15
111         keys.ku[2] = KL[0]; keys.ku[3] = KL[1];
112         rol128(KL[0], KL[1], 30) // KL << 45
113                 keys.ku[6] = KL[0]; keys.ku[7] = KL[1];
114         rol128(KL[0], KL[1], 15) // KL << 60
115                 keys.ku[9] = KL[1];
116         rol128(KL[0], KL[1], 17) // KL << 77
117                 keys.kl[2] = KL[0]; keys.kl[3] = KL[1];
118         rol128(KL[0], KL[1], 17) // KL << 94
119                 keys.ku[12] = KL[0]; keys.ku[13] = KL[1];
120         rol128(KL[0], KL[1], 17) // KL << 111
121                 keys.ku[16] = KL[0]; keys.ku[17] = KL[1];
122
123         // KA-dependent subkeys
124         keys.ku[0] = KA[0]; keys.ku[1] = KA[1];
125         rol128(KA[0], KA[1], 15); // KA << 15
126                 keys.ku[4] = KA[0]; keys.ku[5] = KA[1];
127         rol128(KA[0], KA[1], 15); // KA << 30
128                 keys.kl[0] = KA[0]; keys.kl[1] = KA[1];
129         rol128(KA[0], KA[1], 15) // KA << 45
130                 keys.ku[8] = KA[0];
131         rol128(KA[0], KA[1], 15) // KA << 60
132                 keys.ku[10] = KA[0]; keys.ku[11] = KA[1];
133         rol128(KA[0], KA[1], 34) // KA << 94
134                 keys.ku[14] = KA[0]; keys.ku[15] = KA[1];
135         rol128(KA[0], KA[1], 17) // KA << 111
136                 keys.kw[2] = KA[0]; keys.kw[3] = KA[1];
137
138         memcpy(rks, &keys, sizeof(keys));
139
140         // KB not needed for 128 bit
141 }
142
143 void camellia_spec_opt_encrypt_128(uint64_t c[restrict 2],
144                                    const uint64_t m[restrict 2],
145                                    const struct camellia_rks_128 *restrict rks)
146 {
147         memcpy(c, m, sizeof(c[0]) * 2);
148
149         c[0] ^= rks->kw[0];
150         c[1] ^= rks->kw[1];
151
152         for (size_t i = 0; i < 6; i++) {
153                 camellia_spec_opt_feistel_round(c, rks->ku[i + 0]);
154         }
155
156         c[0] = camellia_spec_opt_FL(c[0], rks->kl[0]);
157         c[1] = camellia_spec_opt_FL_inv(c[1], rks->kl[1]);
158
159         for (size_t i = 0; i < 6; i++) {
160                 camellia_spec_opt_feistel_round(c, rks->ku[i + 6]);
```

```
161         }
162
163         c[0] = camellia_spec_opt_FL(c[0], rks->kl[2]);
164         c[1] = camellia_spec_opt_FL_inv(c[1], rks->kl[3]);
165
166         for (size_t i = 0; i < 6; i++) {
167                 camellia_spec_opt_feistel_round(c, rks->ku[i + 12]);
168         }
169
170         // swap c[0] and c[1] (concatenation of R||L)
171         uint64_t t = c[0];
172         c[0] = c[1]; c[1] = t;
173
174         c[0] ^= rks->kw[2];
175         c[1] ^= rks->kw[3];
176 }
177
178 void camellia_spec_opt_decrypt_128(uint64_t m[restrict 2],
179                                    const uint64_t c[restrict 2],
180                                    const struct camellia_rks_128 *restrict rks)
181 {
182         memcpy(m, c, sizeof(m[0]) * 2);
183
184         m[0] ^= rks->kw[2]; m[1] ^= rks->kw[3];
185
186         // swap c[0] and c[1] (concatenation of R||L)
187         uint64_t t = m[0];
188         m[0] = m[1]; m[1] = t;
189
190         for (size_t i = 6; i --> 0; ) {
191                 camellia_spec_opt_feistel_round_inv(m, rks->ku[i + 12]);
192         }
193
194         m[1] = camellia_spec_opt_FL(m[1], rks->kl[3]);
195         m[0] = camellia_spec_opt_FL_inv(m[0], rks->kl[2]);
196
197         for (size_t i = 6; i --> 0; ) {
198                 camellia_spec_opt_feistel_round_inv(m, rks->ku[i + 6]);
199         }
200
201         m[1] = camellia_spec_opt_FL(m[1], rks->kl[1]);
202         m[0] = camellia_spec_opt_FL_inv(m[0], rks->kl[0]);
203
204         for (size_t i = 6; i --> 0; ) {
205                 camellia_spec_opt_feistel_round_inv(m, rks->ku[i + 0]);
206         }
207
208         m[0] ^= rks->kw[0]; m[1] ^= rks->kw[1];
209
210 }
```

C.2.2 Byteslicing

Listing C.9: `camellia/bytesliced.h`

```
1 #pragma once
2
3 // 128-bit bitsliced camellia with 16 blocks encrypted in parallel
```

```
4
5
6  #include <stdint.h>
7  #include <arm_neon.h>
8
9  #include "camellia_keys.h"
10
11 void rol32_1(uint8x16x4_t *a);
12
13 void camellia_sliced_F(uint8x16x4_t X[restrict 2],
14                        const uint8x16x4_t k[restrict 2]);
15 void camellia_sliced_FL(uint8x16x4_t X[restrict 2],
16                         const uint8x16x4_t kl[restrict 2]);
17 void camellia_sliced_FL_inv(uint8x16x4_t Y[restrict 2],
18                             const uint8x16x4_t kl[restrict 2]);
19 void camellia_sliced_feistel_round(uint8x16x4_t state[restrict 4],
20                                    const uint8x16x4_t kr[restrict 2]);
21 void camellia_sliced_feistel_round_inv(uint8x16x4_t state[restrict 4],
22                                        const uint8x16x4_t kr[restrict 2]);
23 void camellia_sliced_generate_round_keys_128(struct camellia_rks_sliced_128 *restrict
       rks,
24                                               const uint64_t key[2]);
25
26 void camellia_sliced_pack(uint8x16x4_t packed[restrict 4],
27                           const uint64_t x[restrict 16][2]);
28
29 void camellia_sliced_unpack(uint64_t x[restrict 16][2],
30                             const uint8x16x4_t packed[restrict 4]);
31
32 void camellia_sliced_init(void);
33
34 void camellia_sliced_encrypt_128(uint64_t c[restrict 16][2],
35                                  const uint64_t m[restrict 16][2],
36                                  struct camellia_rks_sliced_128 *restrict rks);
37
38 void camellia_sliced_decrypt_128(uint64_t m[restrict 16][2],
39                                  const uint64_t c[restrict 16][2],
40                                  struct camellia_rks_sliced_128 *restrict rks);
```

Listing C.10: `camellia/bytesliced.c`

```
1  #include <arm_neon.h>
2  #include <stdint.h>
3  #include <stddef.h>
4  #include <string.h>
5
6  #include "bytesliced.h"
7  #include "spec_opt.h" // need the spec_opt key schedule
8
9  static uint8x16x4_t pack_group;
10 static uint8x16x4_t pack_group_inv;
11 static uint8x16x4_t pack_single;
12
13 static uint8x16_t lower_4_bits_mask;
14 static uint8x16_t shiftrows_inv;
15 static uint8x16_t rol32_1_overflow;
16
17 // store two for low ([0]) and high ([1]) bits of matrix multiplication
18 static uint8x16x2_t prefilter_0;
19 static uint8x16x2_t prefilter_3;  // s3(x) = s0(x <<< 1)
20 static uint8x16x2_t postfilter_0;
```

```
21 static uint8x16x2_t postfilter_1; // s1(x) = s0(x) <<< 1
22 static uint8x16x2_t postfilter_2; // s2(x) = s0(x) >>> 1
23
24 void rol32_1(uint8x16x4_t *a)
25 {
26         uint8x16_t a3 = a->val[3];
27
28         for (size_t i = 3; i > 0; i--) {
29                 uint8x16_t *curr = &a->val[i];
30                 uint8x16_t *prev = &a->val[i - 1];
31
32                 uint8x16_t overflow = vshrq_n_u8(*prev, 7);
33                 *curr = vorrq_u8(vshlq_n_u8(*curr, 1), overflow);
34         }
35
36         uint8x16_t overflow = vshrq_n_u8(a3, 7);
37         a->val[0] = vorrq_u8(vshlq_n_u8(a->val[0], 1), overflow);
38 }
39
40 static uint8x16_t s(const uint8x16_t X,
41                     const uint8x16x2_t prefilter,
42                     const uint8x16x2_t postfilter)
43 {
44         // prefilter
45         uint8x16_t pre_low  = vqtbl1q_u8(prefilter.val[0],
46                                          vandq_u8(X, lower_4_bits_mask));
47         uint8x16_t pre_high = vqtbl1q_u8(prefilter.val[1],
48                                          vshrq_n_u8(X, 4));
49         uint8x16_t pre = veorq_u8(pre_low, pre_high);
50
51         // inverse ShiftRows
52         pre = vqtbl1q_u8(pre, shiftrows_inv);
53
54         // AES single round encryption (x <- AESSubBytes(AESShiftRows(x)))
55         uint8x16_t aesd = vaeseq_u8(pre, vdupq_n_u8(0x0));
56
57         // postfilter
58         uint8x16_t post_low  = vqtbl1q_u8(postfilter.val[0],
59                                           vandq_u8(aesd, lower_4_bits_mask));
60         uint8x16_t post_high = vqtbl1q_u8(postfilter.val[1],
61                                           vshrq_n_u8(aesd, 4));
62         uint8x16_t post = veorq_u8(post_low, post_high);
63
64         return post;
65 }
66
67 void camellia_sliced_F(uint8x16x4_t X[restrict 2],
68                        const uint8x16x4_t k[restrict 2])
69 {
70         // key additions
71         for (size_t byte = 0; byte < 8; byte++) {
72                 uint8x16_t *reg = &X[byte / 4].val[byte % 4];
73
74                 *reg = veorq_u8(*reg, k[byte / 4].val[byte % 4]);
75         }
76
77         // S-boxes (beware of endianness)
78         X[1].val[3] = s(X[1].val[3], prefilter_0, postfilter_0); // s0
79         X[1].val[2] = s(X[1].val[2], prefilter_0, postfilter_1); // s1
80         X[1].val[1] = s(X[1].val[1], prefilter_0, postfilter_2); // s2
81         X[1].val[0] = s(X[1].val[0], prefilter_3, postfilter_0); // s3
82         X[0].val[3] = s(X[0].val[3], prefilter_0, postfilter_1); // s1
```

```
83          X[0].val[2] = s(X[0].val[2], prefilter_0, postfilter_2); // s2
84          X[0].val[1] = s(X[0].val[1], prefilter_3, postfilter_0); // s3
85          X[0].val[0] = s(X[0].val[0], prefilter_0, postfilter_0); // s0
86
87          // permutation
88          X[1].val[3] = veorq_u8(X[1].val[3], X[0].val[2]);
89          X[1].val[2] = veorq_u8(X[1].val[2], X[0].val[1]);
90          X[1].val[1] = veorq_u8(X[1].val[1], X[0].val[0]);
91          X[1].val[0] = veorq_u8(X[1].val[0], X[0].val[3]);
92          X[0].val[3] = veorq_u8(X[0].val[3], X[1].val[1]);
93          X[0].val[2] = veorq_u8(X[0].val[2], X[1].val[0]);
94          X[0].val[1] = veorq_u8(X[0].val[1], X[1].val[3]);
95          X[0].val[0] = veorq_u8(X[0].val[0], X[1].val[2]);
96
97          X[1].val[3] = veorq_u8(X[1].val[3], X[0].val[0]);
98          X[1].val[2] = veorq_u8(X[1].val[2], X[0].val[3]);
99          X[1].val[1] = veorq_u8(X[1].val[1], X[0].val[2]);
100         X[1].val[0] = veorq_u8(X[1].val[0], X[0].val[1]);
101         X[0].val[3] = veorq_u8(X[0].val[3], X[1].val[0]);
102         X[0].val[2] = veorq_u8(X[0].val[2], X[1].val[3]);
103         X[0].val[1] = veorq_u8(X[0].val[1], X[1].val[2]);
104         X[0].val[0] = veorq_u8(X[0].val[0], X[1].val[1]);
105
106
107         // X[0] and X[1] are swapped now; this is
108         // taken into account in the feistel round
109 }
110
111 void camellia_sliced_FL(uint8x16x4_t X[restrict 2],
112                         const uint8x16x4_t kl[restrict 2])
113 {
114         const uint8x16x4_t XL = X[1];
115         const uint8x16x4_t XR = X[0];
116
117         const uint8x16x4_t klL = kl[1];
118         const uint8x16x4_t klR = kl[0];
119
120         uint8x16x4_t YR_prerotate = {
121                 .val = {
122                         vandq_u8(XL.val[0], klL.val[0]),
123                         vandq_u8(XL.val[1], klL.val[1]),
124                         vandq_u8(XL.val[2], klL.val[2]),
125                         vandq_u8(XL.val[3], klL.val[3])
126                 }
127         };
128
129         rol32_1(&YR_prerotate);
130
131         const uint8x16x4_t YR = {
132                 .val = {
133                         veorq_u8(YR_prerotate.val[0], XR.val[0]),
134                         veorq_u8(YR_prerotate.val[1], XR.val[1]),
135                         veorq_u8(YR_prerotate.val[2], XR.val[2]),
136                         veorq_u8(YR_prerotate.val[3], XR.val[3])
137                 }
138         };
139
140         const uint8x16x4_t YL = {
141                 .val = {
142                         veorq_u8(vorrq_u8(YR.val[0], klR.val[0]), XL.val[0]),
143                         veorq_u8(vorrq_u8(YR.val[1], klR.val[1]), XL.val[1]),
144                         veorq_u8(vorrq_u8(YR.val[2], klR.val[2]), XL.val[2]),
```

```
145                         veorq_u8(vorrq_u8(YR.val[3], klR.val[3]), XL.val[3])
146                 }
147         };
148
149         X[0] = YR;
150         X[1] = YL;
151 }
152
153 void camellia_sliced_FL_inv(uint8x16x4_t Y[restrict 2],
154                             const uint8x16x4_t kl[restrict 2])
155 {
156         const uint8x16x4_t YL = Y[1];
157         const uint8x16x4_t YR = Y[0];
158
159         const uint8x16x4_t klL = kl[1];
160         const uint8x16x4_t klR = kl[0];
161
162         const uint8x16x4_t XL = {
163                 .val = {
164                         veorq_u8(vorrq_u8(YR.val[0], klR.val[0]), YL.val[0]),
165                         veorq_u8(vorrq_u8(YR.val[1], klR.val[1]), YL.val[1]),
166                         veorq_u8(vorrq_u8(YR.val[2], klR.val[2]), YL.val[2]),
167                         veorq_u8(vorrq_u8(YR.val[3], klR.val[3]), YL.val[3])
168                 }
169         };
170
171         uint8x16x4_t XR_prerotate = {
172                 .val = {
173                         vandq_u8(XL.val[0], klL.val[0]),
174                         vandq_u8(XL.val[1], klL.val[1]),
175                         vandq_u8(XL.val[2], klL.val[2]),
176                         vandq_u8(XL.val[3], klL.val[3])
177                 }
178         };
179
180         rol32_1(&XR_prerotate);
181
182         const uint8x16x4_t XR = {
183                 .val = {
184                         veorq_u8(XR_prerotate.val[0], YR.val[0]),
185                         veorq_u8(XR_prerotate.val[1], YR.val[1]),
186                         veorq_u8(XR_prerotate.val[2], YR.val[2]),
187                         veorq_u8(XR_prerotate.val[3], YR.val[3])
188                 }
189         };
190
191         Y[0] = XR;
192         Y[1] = XL;
193 }
194
195 void camellia_sliced_feistel_round(uint8x16x4_t state[restrict 4],
196                                    const uint8x16x4_t kr[restrict 2])
197 {
198         uint8x16x4_t F[2] = {
199                 state[0], state[1]
200         };
201
202         // F function swaps result
203         camellia_sliced_F(F, kr);
204
205         for (size_t j = 0; j < 4; j++) {
206                 F[0].val[j] = veorq_u8(state[3].val[j], F[0].val[j]);
```

```
207                 F[1].val[j] = veorq_u8(state[2].val[j], F[1].val[j]);
208         }
209
210         state[2] = state[0];
211         state[3] = state[1];
212         state[0] = F[1];
213         state[1] = F[0];
214 }
215
216 void camellia_sliced_feistel_round_inv(uint8x16x4_t state[restrict 4],
217                                        const uint8x16x4_t kr[restrict 2])
218 {
219         uint8x16x4_t F[2] = {
220                 state[2], state[3]
221         };
222
223         // F function swaps result
224         camellia_sliced_F(F, kr);
225
226         for (size_t j = 0; j < 4; j++) {
227                 F[0].val[j] = veorq_u8(state[1].val[j], F[0].val[j]);
228                 F[1].val[j] = veorq_u8(state[0].val[j], F[1].val[j]);
229         }
230
231         state[0] = state[2];
232         state[1] = state[3];
233         state[2] = F[1];
234         state[3] = F[0];
235 }
236
237 void camellia_sliced_generate_round_keys_128(struct camellia_rks_sliced_128 *restrict
    rks,
238                                              const uint64_t key[restrict 2])
239 {
240         // use standard key derivation
241         struct camellia_rks_128 rks_128;
242         camellia_spec_opt_generate_round_keys_128(&rks_128, key);
243
244         // now pack round keys by use of vdupq_n_u8 since all bytes are the same
245         // in a bytesliced representation
246
247         // whitening and FL layer keys
248         for (size_t i = 0; i < 4; i++) {
249                 for (size_t byte = 0; byte < 8; byte++) {
250                         uint8x16_t *reg_kw = &rks->kw[i][byte / 4].val[byte % 4];
251                         uint8x16_t *reg_kl = &rks->kl[i][byte / 4].val[byte % 4];
252
253                         *reg_kw = vdupq_n_u8((rks_128.kw[i] >> (8 * byte)) & 0xff);
254                         *reg_kl = vdupq_n_u8((rks_128.kl[i] >> (8 * byte)) & 0xff);
255                 }
256         }
257
258         // F function keys
259         for (size_t i = 0; i < 18; i++) {
260                 for (size_t byte = 0; byte < 8; byte++) {
261                         uint8x16_t *reg_ku = &rks->ku[i][byte / 4].val[byte % 4];
262
263                         *reg_ku = vdupq_n_u8((rks_128.ku[i] >> (8 * byte)) & 0xff);
264                 }
265         }
266 }
267
```

```
268 void camellia_sliced_pack(uint8x16x4_t packed[restrict 4],
269                           const uint64_t x[restrict 16][2])
270 {
271         for (size_t i = 0; i < 4; i++) {
272                 packed[i] = vld1q_u8_x4((uint8_t*)&x[i * 4][0]);
273         }
274 
275         uint8x16x4_t packed_0[4];
276 
277         for (size_t i = 0; i < 4; i++) {
278                 for (size_t j = 0; j < 4; j++) {
279                         packed_0[i].val[j] = vqtbl4q_u8(packed[j], pack_group.val[i]);
280                 }
281         }
282 
283         for (size_t i = 0; i < 4; i++) {
284                 for (size_t j = 0; j < 4; j++) {
285                         packed[i].val[j] = vqtbl4q_u8(packed_0[i], pack_single.val[j]);
286                 }
287         }
288 }
289 
290 void camellia_sliced_unpack(uint64_t x[restrict 16][2],
291                             const uint8x16x4_t packed[restrict 4])
292 {
293 
294         uint8x16x4_t unpacked[4];
295         for (size_t i = 0; i < 4; i++) {
296                 for (size_t j = 0; j < 4; j++) {
297                         unpacked[i].val[j] = vqtbl4q_u8(packed[j], pack_group_inv.val[i
                                ]);
298                 }
299         }
300 
301         uint8x16x4_t unpacked_0[4];
302 
303         // pack_single_inv = pack_group
304         for (size_t i = 0; i < 4; i++) {
305                 for (size_t j = 0; j < 4; j++) {
306                         unpacked_0[i].val[j] = vqtbl4q_u8(unpacked[i], pack_group.val[j
                                ]);
307                 }
308         }
309 
310         for (size_t i = 0; i < 4; i++) {
311                 vst1q_u8_x4((uint8_t*)&x[i * 4][0], unpacked_0[i]);
312         }
313 }
314 
315 
316 void camellia_sliced_init(void)
317 {
318         static const uint64_t pack_group_u64[4][2] = {
319                 { 0x1312111003020100UL, 0x3332313023222120UL },
320                 { 0x1716151407060504UL, 0x3736353427262524UL },
321                 { 0x1b1a19180b0a0908UL, 0x3b3a39382b2a2928UL },
322                 { 0x1f1e1d1c0f0e0d0cUL, 0x3f3e3d3c2f2e2d2cUL },
323         };
324 
325         static const uint64_t pack_group_inv_u64[4][2] = {
326                 { 0x3121110130201000UL, 0x3323130332221202UL },
327                 { 0x3525150534241404UL, 0x3727170736261606UL },
```

```
328                 { 0x3929190938281808UL, 0x3b2b1b0b3a2a1a0aUL },
329                 { 0x3d2d1d0d3c2c1c0cUL, 0x3f2f1f0f3e2e1e0eUL }
330         };
331
332         static const uint64_t pack_single_u64[4][2] = {
333                 { 0x1c1814100c080400UL, 0x3c3834302c282420UL },
334                 { 0x1d1915110d090501UL, 0x3d3935312d292521UL },
335                 { 0x1e1a16120e0a0602UL, 0x3e3a36322e2a2622UL },
336                 { 0x1f1b17130f0b0703UL, 0x3f3b37332f2b2723UL },
337         };
338
339         static const uint64_t shiftrows_inv_u64[2] = {
340                 0x0b0e0104070a0d00UL, 0x0306090c0f020508UL };
341
342         static const uint64_t rol32_1_overflow_u64[2] = {
343                 0x0605040302010010, 0x0e0d0c0b0a090807UL };
344
345         static const uint64_t prefilter_0_u64[2][2] = {
346                 { 0x862b832eed40e845UL, 0x88258d20e34ee64bUL },
347                 { 0x2a7bdb8aa0f15100UL, 0x2372d283a9f85809UL } };
348
349         static const uint64_t prefilter_3_u64[2][2] = {
350                 { 0x25204e4b2b2e4045UL, 0x74711f1a7a7f1114UL },
351                 { 0x7283f8097b8af100UL, 0xdf2e55a4d6275cadUL } };
352
353         static const uint64_t postfilter_0_u64[2][2] = {
354                 { 0x31c1c2323fcfcc3cUL, 0xd12122d2df2f2cdcUL },
355                 { 0xa8512ed77f86f900UL, 0x0cf58a73db225da4UL } };
356
357         static const uint64_t postfilter_1_u64[2][2] = {
358                 { 0x628385647e9f9978UL, 0xa34244a5bf5e58b9UL },
359                 { 0x51a25caffe0df300UL, 0x18eb15e6b744ba49UL } };
360
361         static const uint64_t postfilter_2_u64[2][2] = {
362                 { 0x98e061199fe7661eUL, 0xe8901169ef97166eUL },
363                 { 0x54a817ebbf43fc00UL, 0x06fa45b9ed11ae52UL } };
364
365         pack_group          = vld1q_u8_x4((uint8_t*)pack_group_u64);
366         pack_group_inv      = vld1q_u8_x4((uint8_t*)pack_group_inv_u64);
367         pack_single         = vld1q_u8_x4((uint8_t*)pack_single_u64);
368
369         lower_4_bits_mask   = vdupq_n_u8(0x0f);
370         shiftrows_inv       = vld1q_u8((uint8_t*)shiftrows_inv_u64);
371         rol32_1_overflow    = vld1q_u8((uint8_t*)rol32_1_overflow_u64);
372
373         prefilter_0         = vld1q_u8_x2((uint8_t*)prefilter_0_u64);
374         prefilter_3         = vld1q_u8_x2((uint8_t*)prefilter_3_u64);
375         postfilter_0        = vld1q_u8_x2((uint8_t*)postfilter_0_u64);
376         postfilter_1        = vld1q_u8_x2((uint8_t*)postfilter_1_u64);
377         postfilter_2        = vld1q_u8_x2((uint8_t*)postfilter_2_u64);
378 }
379
380 void camellia_sliced_encrypt_128(uint64_t c[restrict 16][2],
381                                  const uint64_t m[restrict 16][2],
382                                  struct camellia_rks_sliced_128 *restrict rks)
383 {
384         uint8x16x4_t state[4];
385         camellia_sliced_pack(state, m);
386
387         // kw0/kw1
388         for (size_t byte = 0; byte < 16; byte++) {
389                 uint8x16_t *reg     = &state[byte / 4].val[byte % 4];
```

```
390                 uint8x16x4_t *key   = &rks->kw[byte / 8 + 0][(byte % 8) / 4];
391
392                 *reg = veorq_u8(*reg, key->val[byte % 4]);
393         }
394
395         for (size_t i = 0; i < 6; i++) {
396                 camellia_sliced_feistel_round(state, rks->ku[i + 0]);
397         }
398
399         camellia_sliced_FL(&state[0], rks->kl[0]);
400         camellia_sliced_FL_inv(&state[2], rks->kl[1]);
401
402         for (size_t i = 0; i < 6; i++) {
403                 camellia_sliced_feistel_round(state, rks->ku[i + 6]);
404         }
405
406         camellia_sliced_FL(&state[0], rks->kl[2]);
407         camellia_sliced_FL_inv(&state[2], rks->kl[3]);
408
409         for (size_t i = 0; i < 6; i++) {
410                 camellia_sliced_feistel_round(state, rks->ku[i + 12]);
411         }
412
413         // swap state[0,1] and state[2,3] (concatenation of R||L)
414         uint8x16x4_t tmp = state[0];
415         state[0] = state[2];
416         state[2] = tmp;
417         tmp = state[1];
418         state[1] = state[3];
419         state[3] = tmp;
420
421         // kw2/kw3
422         for (size_t byte = 0; byte < 16; byte++) {
423                 uint8x16_t *reg     = &state[byte / 4].val[byte % 4];
424                 uint8x16x4_t *key   = &rks->kw[byte / 8 + 2][(byte % 8) / 4];
425
426                 *reg = veorq_u8(*reg, key->val[byte % 4]);
427         }
428
429         camellia_sliced_unpack(c, state);
430 }
431
432 void camellia_sliced_decrypt_128(uint64_t m[restrict 16][2],
433                                  const uint64_t c[restrict 16][2],
434                                  struct camellia_rks_sliced_128 *restrict rks)
435 {
436         uint8x16x4_t state[4];
437         camellia_sliced_pack(state, c);
438
439         // kw2/kw3
440         for (size_t byte = 0; byte < 16; byte++) {
441                 uint8x16_t *reg     = &state[byte / 4].val[byte % 4];
442                 uint8x16x4_t *key   = &rks->kw[byte / 8 + 2][(byte % 8) / 4];
443
444                 *reg = veorq_u8(*reg, key->val[byte % 4]);
445         }
446
447
448         // swap state[0,1] and state[2,3] (concatenation of R||L)
449         uint8x16x4_t tmp = state[0];
450         state[0] = state[2];
451         state[2] = tmp;
```

```
452         tmp = state[1];
453         state[1] = state[3];
454         state[3] = tmp;
455
456         for (size_t i = 6; i --> 0; ) {
457                 camellia_sliced_feistel_round_inv(state, rks->ku[i + 12]);
458         }
459
460         camellia_sliced_FL(&state[2], rks->kl[3]);
461         camellia_sliced_FL_inv(&state[0], rks->kl[2]);
462
463         for (size_t i = 6; i --> 0; ) {
464                 camellia_sliced_feistel_round_inv(state, rks->ku[i + 6]);
465         }
466
467         camellia_sliced_FL(&state[2], rks->kl[1]);
468         camellia_sliced_FL_inv(&state[0], rks->kl[0]);
469
470         for (size_t i = 6; i --> 0; ) {
471                 camellia_sliced_feistel_round_inv(state, rks->ku[i + 0]);
472         }
473
474         // kw0/kw1
475         for (size_t byte = 0; byte < 16; byte++) {
476                 uint8x16_t *reg     = &state[byte / 4].val[byte % 4];
477                 uint8x16x4_t *key   = &rks->kw[byte / 8 + 0][(byte % 8) / 4];
478
479                 *reg = veorq_u8(*reg, key->val[byte % 4]);
480         }
481
482         camellia_sliced_unpack(m, state);
483 }
```

Bibliography

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

[2] Alexandre Adomnicai, Zakaria Najm, and Thomas Peyrin. "Fixslicing: A New GIFT Representation". Cryptology ePrint Archive, Paper 2020/412. https://eprint.iacr.org/2020/412. 2020. URL: https://eprint.iacr.org/2020/412.

[3] Jussi Kivilinna. "Block Ciphers : fast Implementations on x86-64 Architecture". In: 2013.

[4] 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.

[5] 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.

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

[7] 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).

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

[9] 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.

[10] Eli Biham. "A fast new DES implementation in software". In: *Fast Software Encryption*. Ed. by Eli Biham. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 260–272. ISBN: 978-3-540-69243-0.

[11] Akashi Satoh and Sumio Morioka. "Unified Hardware Architecture for 128-Bit Block Ciphers AES and Camellia". In: vol. 2779. Sept. 2003, pp. 304–318. ISBN: 978-3-540-40833-8. DOI: 10.1007/978-3-540-45238-6_25.

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