Efficient Implementation Strategies for Block Ciphers on ARMv8

Bachelorarbeit

Bastian Engel

May 18, 2023

Abstract

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

Declaration

I hereby declare that 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

1 Introduction **5**
1.1 Notation 5
1.2 Block ciphers 5
1.2.1 GIFT 6
1.2.2 Camellia 8
1.3 The ARMv8 platform 11
1.3.1 General architecture 12
1.3.2 NEON 12
1.3.3 NEON Intrinsics 13

2 Implementation strategies **15**
2.1 Strategies for `GIFT` 15
2.1.1 Table-based 15
2.1.2 Using `vperm` 16
2.1.3 Bitslicing 17
2.2 Strategies for Camellia 22
2.2.1 Platform-independent techniques 22
2.2.2 Byteslicing 23

3 Implementation **28**
3.1 Pipelining 28
3.2 GIFT 29
3.2.1 Table-based 29
3.2.2 Using `vperm` 30
3.2.3 Bitslicing 31
3.3 Camellia 34
3.3.1 Optimized non-SIMD implementation 34
3.3.2 Bytesliced implementation 35

4 Evaluation **38**
4.1 Performance evaluation 38
4.2 Conclusion 39

A Detailed benchmarking results **41**
A.1 GIFT 41
A.2 Camellia 42

B C source code **43**
B.1 Implementations for SPN 43
B.1.1 Table-based 43
B.1.2 Using `vperm` 46
B.1.3 Bitslicing 51

C Lorem dolor **58**

Chapter 1

Introduction

1.1 Notation

1.2 Block ciphers

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

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

1.2.1 GIFT

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

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

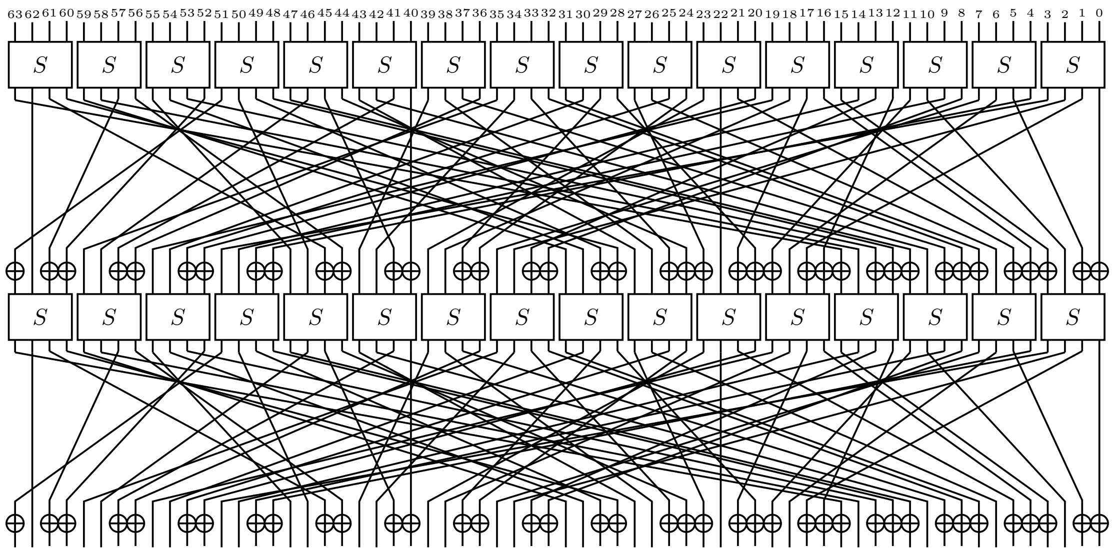


Figure 1.1: Two rounds of GIFT-64

Substitution layer

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

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

Permutation layer

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

$$P_{64}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 16\left(\left(3\left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4)\right) \bmod 4\right) + (i \bmod 4)$$
$$P_{128}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 32\left(\left(3\left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4)\right) \bmod 4\right) + (i \bmod 4)$$

Round key addition

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

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

Round key extraction and key schedule

Round key extraction differs for GIFT-64 and GIFT-128. Let $K_{(128)} = k_7||k_6||\ldots||k_0$ 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:

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

1.2.2 Camellia

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

Encryption

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

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

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

Table 1.1: Camellia encryption

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

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

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

Key schedule

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

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

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

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

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:

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

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:

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

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:

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

where

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

FL^{-1}:

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

where

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

1.3 The ARMv8 platform

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

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

Table 1.3: ARMv8 profiles

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

Table 1.4: ARMv8 execution states

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

1.3.1 General architecture

ARMv8 is a RISC architecture employing simple data processing instructions operating only on registers as well as dedicated load/store instructions to transfer data from register to memory and back. This enables faster execution of individual instructions, a 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`.

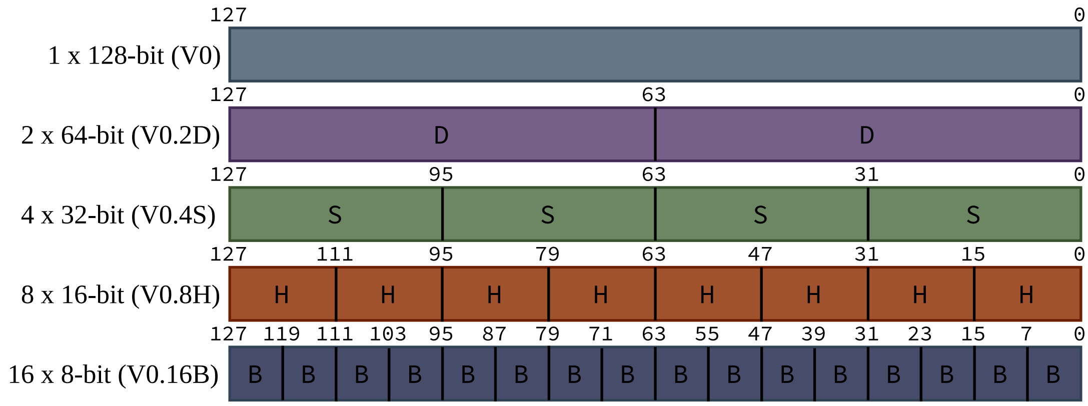


Figure 1.2: Divisions of the V0 register

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
```

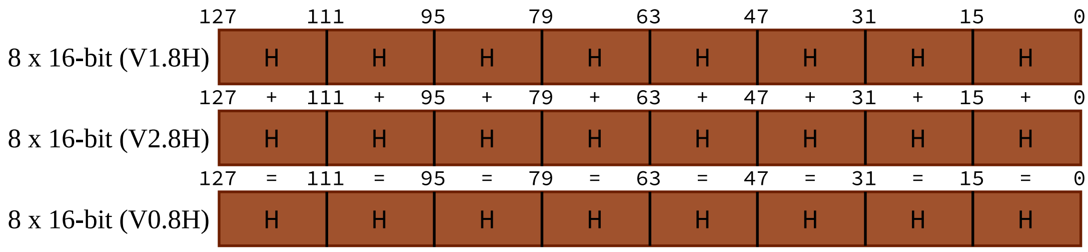


Figure 1.3: Addition of two vector registers

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

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

```
uint16x8_t result = vmulq_u16(a, b);
```

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

Table 1.6: Common NEON intrinsics

Intrinsic		Summary
`uint64_t`	`vgetq_lane_u64(void)`	Extract a single lane
`void`	`vsetq_lane_u64(uint64_t)`	Insert a single lane
`uint64x2_t`	`vdupq_n_u64(uint64_t)`	Initialize all lanes to same value
`void`	`vst1q_u64(uint64_t*, uint64x2_t)`	Store from register to memory
`uint64x2_t`	`vld1q_u64(uint64_t*, uint64x2_t)`	Load from memory to register
`uint8x16_t`	`veorq_u8(uint8x16_t, uint8x16_t)`	bitwise XOR
`uint8x16_t`	`vandq_u8(uint8x16_t, uint8x16_t)`	bitwise AND
`uint8x16_t`	`vorrq_u8(uint8x16_t, uint8x16_t)`	bitwise OR
`uint8x16_t`	`vmvnq_u8(uint8x16_t)`	bitwise NOT
`uint8x16_t`	`vqtbl2q_u8(uint8x16_t, uint8x16_t)`	permutation (`TBL`)
`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 [7]:

- Table-based implementations
- `vperm` implementations
- Bitslice implementations

2.1.1 Table-based

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

A common approach is to tabulate the output of each S-box, including the diffusion layer, and then XORing the results together. Let n denote the internal cipher state size and s the size of a single S-box in bits. For each S-box $S_i, i \in \{0, \ldots, \frac{n}{s}\}$, 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
```

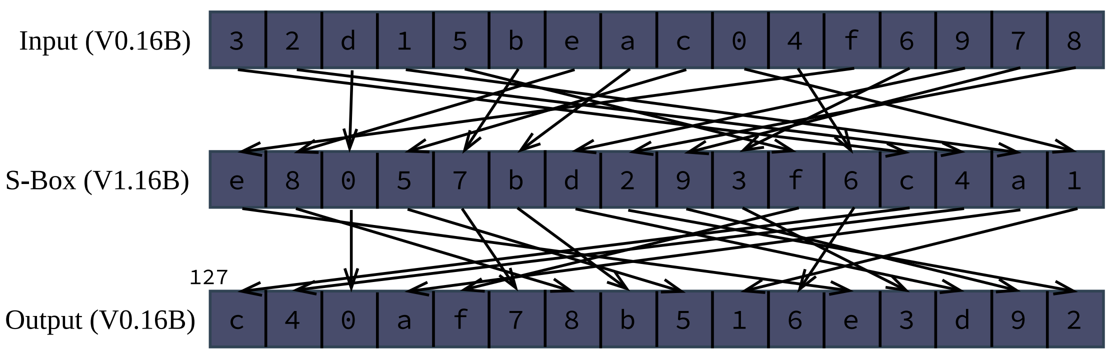


Figure 2.1: Performing the S-Box lookup in parallel

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[8]. 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.

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.

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

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.

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

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 Platform-independent techniques

The original paper proposes various platform-independent ways to implement Camellia efficiently. 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:

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

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.

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 [9]. NEON itself possesses cryptographic extensions for finite field arithmetic and AES as well as SHA calculations. These can

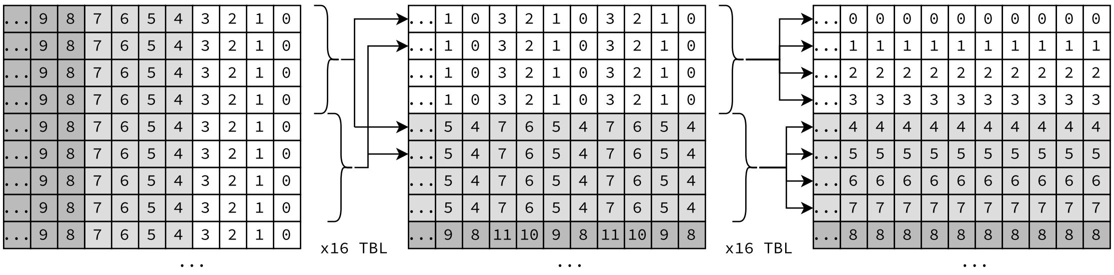


Figure 2.2: Camellia bytesliced packing function with 32 `TBL` operations. Curly brackets represent a 4-element vector array being used as source registers.

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[10]:

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

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.

$$\mathtt{AESE}(x) = \mathrm{SubBytes}(\mathrm{ShiftRow(x)}) \Leftrightarrow \mathtt{AESE}(\mathrm{ShiftRow}^{-1}(x)) = \mathrm{SubBytes}(x)$$

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

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

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

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

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:

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

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

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

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:

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

This simplifies our final equation:

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

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:

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

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.

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

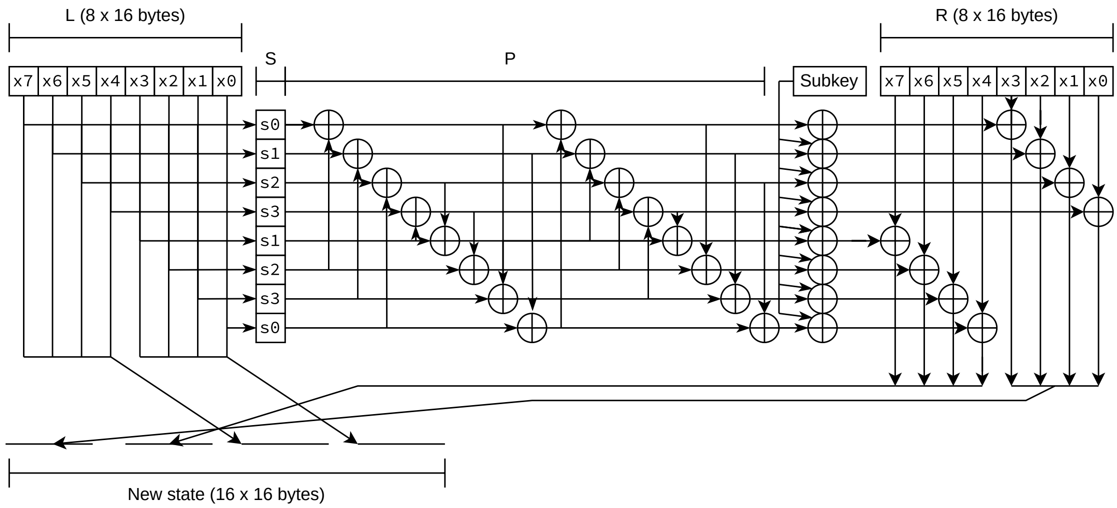


Figure 2.3: Camellia bytesliced Feistel round function

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 $(i+4)(\text{mod}4)$ for the final result.

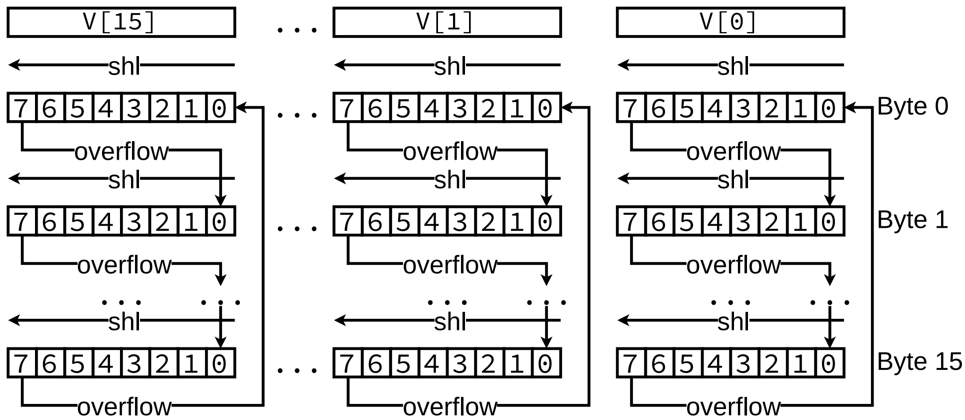


Figure 2.4: Camellia bytesliced 1-bit left rotate

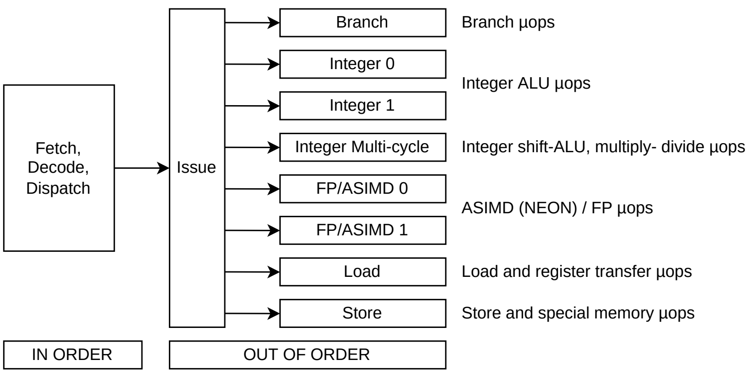
Chapter 3

Implementation

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

3.1 Pipelining

Understanding certain choices requires an understanding of the Cortex-A73 instruction pipeline[11]. 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.



h!

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

```c
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

```c
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 AArch64 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);

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

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

3.2.3 Bitslicing

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

```
/*
uint8x16_t shl(const uint8x16_t v, const int n) */
#define shl(_a, v, n)                                                   \
{                                                                       \
        uint64x2_t _overflow = vshrq_n_u64(v, 64 - n);                 \
        _overflow = vextq_u64(vdupq_n_u64(0x0), _overflow, 1);         \
        _a = vorrq_u8(vshlq_n_u64(v, n), _overflow);                   \
}
```

```c
/*
uint8x16_t shr(const uint8x16_t v, const int n) */
#define shr(_a, v, n)                                                     \
{                                                                         \
        uint64x2_t _overflow = vshlq_n_u64(v, 64 - n);                    \
        _overflow = vextq_u64(_overflow, vdupq_n_u64(0x0), 1);            \
        _a = vorrq_u8(vshrq_n_u64(v, n), _overflow);                      \
}

/* implemented as a macro so we can use vshlq_n_u8 with variable n
void gift_64_vec_sliced_swapmove(uint8x16_t *restrict a, uint8x16_t *restrict b,
                                 const uint8x16_t m, const int n) */
#define gift_64_vec_sliced_swapmove(a, b, m, n)                           \
{                                                                         \
        uint8x16_t _a;                                                    \
        shr(_a, a, n);                                                    \
        const uint8x16_t _t = vandq_u8(veorq_u8(_a, b), m);               \
        b = veorq_u8(b, _t);                                              \
        shl(_a, _t, n);                                                   \
        a = veorq_u8(a, _a);                                              \
}
```

Round function

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

Listing 3.7: Bitsliced GIFT S-box

```c
void gift_64_vec_sliced_subcells(uint8x16x4_t cs[restrict 2])
{
        cs[0].val[1] = veorq_u8(cs[0].val[1],
                                vandq_u8(cs[0].val[0], cs[0].val[2]));
        uint8x16_t t = veorq_u8(cs[0].val[0],
                                vandq_u8(cs[0].val[1], cs[0].val[3]));
        cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
        cs[0].val[0] = veorq_u8(cs[0].val[3], cs[0].val[2]);
        cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
        cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
        cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
        cs[0].val[3] = t;

        cs[1].val[1] = veorq_u8(cs[1].val[1],
                                vandq_u8(cs[1].val[0], cs[1].val[2]));
        t            = veorq_u8(cs[1].val[0],
                                vandq_u8(cs[1].val[1], cs[1].val[3]));
        cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
        cs[1].val[0] = veorq_u8(cs[1].val[3], cs[1].val[2]);
        cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
        cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
        cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
        cs[1].val[3] = t;
}
```

Listing 3.8: Bitsliced GIFT permutation

```c
void gift_64_vec_sliced_permute(uint8x16x4_t cs[restrict 2])
```

```
{
        cs[0].val[0] = vqtbl1q_u8(cs[0].val[0], perm.val[0]);
        cs[0].val[1] = vqtbl1q_u8(cs[0].val[1], perm.val[1]);
        cs[0].val[2] = vqtbl1q_u8(cs[0].val[2], perm.val[2]);
        cs[0].val[3] = vqtbl1q_u8(cs[0].val[3], perm.val[3]);

        cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm.val[0]);
        cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm.val[1]);
        cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm.val[2]);
        cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm.val[3]);
}
```

Listing 3.9: Round function

```
for (int round = 0; round < ROUNDS_GIFT_64; round++) {
    gift_64_vec_sliced_subcells(s);
    gift_64_vec_sliced_permute(s);

    // round key addition
    s[0].val[0] = veorq_u8(s[0].val[0], rks[round][0].val[0]);
    s[0].val[1] = veorq_u8(s[0].val[1], rks[round][0].val[1]);
    s[0].val[2] = veorq_u8(s[0].val[2], rks[round][0].val[2]);
    s[0].val[3] = veorq_u8(s[0].val[3], rks[round][0].val[3]);
    s[1].val[0] = veorq_u8(s[1].val[0], rks[round][1].val[0]);
    s[1].val[1] = veorq_u8(s[1].val[1], rks[round][1].val[1]);
    s[1].val[2] = veorq_u8(s[1].val[2], rks[round][1].val[2]);
    s[1].val[3] = veorq_u8(s[1].val[3], rks[round][1].val[3]);
}
```

Listing 3.10: Round function asm

```
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 assembly.

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 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

```c
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

```c
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
```

```c
	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];
			uint8x16_t *reg_kl = &rks->kl[i][byte / 4].val[byte % 4];

			*reg_kw = vdupq_n_u8((rks_128.kw[i] >> (8 * byte)) & 0xff);
			*reg_kl = vdupq_n_u8((rks_128.kl[i] >> (8 * byte)) & 0xff);
		}
	}

	// F function keys
	for (size_t i = 0; i < 18; i++) {
		for (size_t byte = 0; byte < 8; byte++) {
			uint8x16_t *reg_ku = &rks->ku[i][byte / 4].val[byte % 4];

			*reg_ku = vdupq_n_u8((rks_128.ku[i] >> (8 * byte)) & 0xff);
		}
	}
}
```

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

```c
static uint8x16_t s(const uint8x16_t X,
                    const uint8x16x2_t prefilter,
                    const uint8x16x2_t postfilter)
{
	// prefilter
	uint8x16_t pre_low  = vqtbl1q_u8(prefilter.val[0],
	                                 vandq_u8(X, lower_4_bits_mask));
	uint8x16_t pre_high = vqtbl1q_u8(prefilter.val[1],
	                                 vshrq_n_u8(X, 4));
	uint8x16_t pre = veorq_u8(pre_low, pre_high);

	// inverse ShiftRows
	pre = vqtbl1q_u8(pre, shiftrows_inv);

	// AES single round encryption (x <- AESSubBytes(AESShiftRows(x)))
	uint8x16_t aesd = vaeseq_u8(pre, vdupq_n_u8(0x0));

	// postfilter
	uint8x16_t post_low  = vqtbl1q_u8(postfilter.val[0],
	                                  vandq_u8(aesd, lower_4_bits_mask));
	uint8x16_t post_high = vqtbl1q_u8(postfilter.val[1],
	                                  vshrq_n_u8(aesd, 4));
	uint8x16_t post = veorq_u8(post_low, post_high);
```

```

        return post;
}
```

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 discuss advantages, disadvantages and possible use cases.

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 (TODO REF HERE) taking place in the actual encryption function.

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

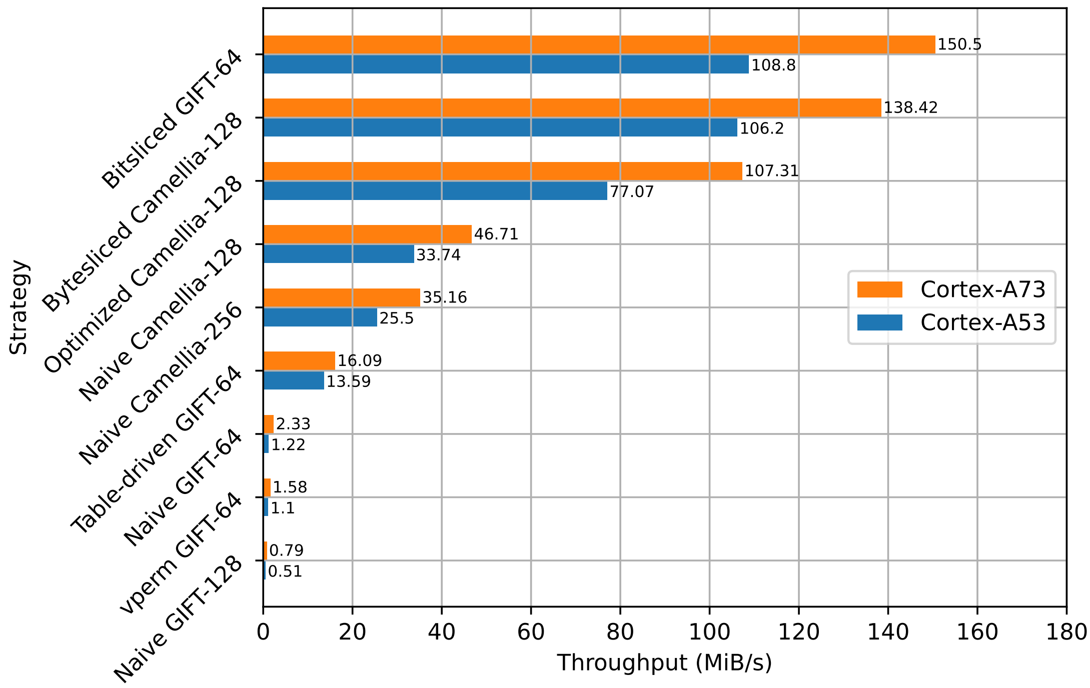


Figure 4.1: Throughput in MiB/s for each strategy and processor type

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.

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

Acknowledgements

I want to thank ...

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

C source code

B.1 Implementations for SPN

B.1.1 Table-based

Listing B.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 B.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
};
```

```

static const uint64_t tables[16][16] = {
        { 0x0000000000000001UL, 0x0008000000020000UL, 0x0000000400000000UL, 0
            x0008000400000000UL, 0x0000000400020000UL, 0x0008000400020001UL, 0
            x0000000000020001UL, 0x0008000000000001UL, 0x0000000000020000UL, 0
            x0008000400000001UL, 0x0008000000020001UL, 0x0000000400020001UL, 0
            x0000000400000001UL, 0x0000000000000000UL, 0x0008000000000000UL, 0
            x0008000400020000UL },
        { 0x0001000000000000UL, 0x0000000800000002UL, 0x0000000000040000UL, 0
            x0000000800040000UL, 0x0000000000040002UL, 0x0001000800040002UL, 0
            x0001000000000002UL, 0x0001000800000000UL, 0x0000000000000002UL, 0
            x0001000800040000UL, 0x0001000800000002UL, 0x0001000000040002UL, 0
            x0001000000040000UL, 0x0000000000000000UL, 0x0000000800000000UL, 0
            x0000000800040002UL },
        { 0x0000000100000000UL, 0x0002000000080000UL, 0x0000000000000004UL, 0
            x0000000000080004UL, 0x0002000000000004UL, 0x0002000100080004UL, 0
            x0002000100000000UL, 0x0000000100080000UL, 0x0002000000000000UL, 0
            x0000000100080004UL, 0x0002000100080000UL, 0x0002000100000004UL, 0
            x0000000100000004UL, 0x0000000000000000UL, 0x0000000000080000UL, 0
            x0002000000080004UL },
        { 0x0000000000010000UL, 0x0000000200000008UL, 0x0004000000000000UL, 0
            x0004000000000008UL, 0x0004000200000000UL, 0x0004000200010008UL, 0
            x0000000200010000UL, 0x0000000000010008UL, 0x0000000200000000UL, 0
            x0004000000010008UL, 0x0000000200010008UL, 0x0004000200010000UL, 0
            x0004000000010000UL, 0x0000000000000000UL, 0x0000000000000008UL, 0
            x0004000200000008UL },
        { 0x0000000000000010UL, 0x0080000000200000UL, 0x0000004000000000UL, 0
            x0080004000000000UL, 0x0000004000200000UL, 0x0080004000200010UL, 0
            x0000000000200010UL, 0x0080000000000010UL, 0x0000000000200000UL, 0
            x0080004000000010UL, 0x0080000000200010UL, 0x0000004000200010UL, 0
            x0000004000000010UL, 0x0000000000000000UL, 0x0080000000000000UL, 0
            x0080004000200000UL },
        { 0x0010000000000000UL, 0x0000008000000020UL, 0x0000000000400000UL, 0
            x0000008000400000UL, 0x0000000000400020UL, 0x0010008000400020UL, 0
            x0010000000000020UL, 0x0010008000000000UL, 0x0000000000000020UL, 0
            x0010008000400000UL, 0x0010008000000020UL, 0x0010000000400020UL, 0
            x0010000000400000UL, 0x0000000000000000UL, 0x0000008000000000UL, 0
            x0000008000400020UL },
        { 0x0000001000000000UL, 0x0020000000800000UL, 0x0000000000000040UL, 0
            x0000000000800040UL, 0x0020000000000040UL, 0x0020001000800040UL, 0
            x0020001000000000UL, 0x0000001000800000UL, 0x0020000000000000UL, 0
            x0000001000800040UL, 0x0020001000800000UL, 0x0020001000000040UL, 0
            x0000001000000040UL, 0x0000000000000000UL, 0x0000000000800000UL, 0
            x0020000000800040UL },
        { 0x0000000000100000UL, 0x0000002000000080UL, 0x0040000000000000UL, 0
            x0040000000000080UL, 0x0040002000000000UL, 0x0040002000100080UL, 0
            x0000002000100000UL, 0x0000000000100080UL, 0x0000002000000000UL, 0
            x0040000000100080UL, 0x0000002000100080UL, 0x0040002000100000UL, 0
            x0040000000100000UL, 0x0000000000000000UL, 0x0000000000000080UL, 0
            x0040002000000080UL },
        { 0x0000000000000100UL, 0x0800000002000000UL, 0x0000040000000000UL, 0
            x0800040000000000UL, 0x0000040002000000UL, 0x0800040002000100UL, 0
            x0000000002000100UL, 0x0800000000000100UL, 0x0000000002000000UL, 0
            x0800040000000100UL, 0x0800000002000100UL, 0x0000040002000100UL, 0
            x0000040000000100UL, 0x0000000000000000UL, 0x0800000000000000UL, 0
            x0800040002000000UL },
        { 0x0100000000000000UL, 0x0000080000000200UL, 0x0000000004000000UL, 0
            x0000080004000000UL, 0x0000000004000200UL, 0x0100080004000200UL, 0
            x0100000000000200UL, 0x0100080000000000UL, 0x0000000000000200UL, 0
            x0100080004000000UL, 0x0100080000000200UL, 0x0100000004000200UL, 0
            x0100000004000000UL, 0x0000000000000000UL, 0x0000080000000000UL, 0
            x0000080004000200UL },
```

```
        { 0x0000010000000000UL, 0x0200000008000000UL, 0x0000000000000400UL, 0
            x0000000008000400UL, 0x0200000000000400UL, 0x0200010008000400UL, 0
            x0200010000000000UL, 0x0000010008000000UL, 0x0200000000000000UL, 0
            x0000010008000400UL, 0x0200010008000000UL, 0x0200010000000400UL, 0
            x0000010000000400UL, 0x0000000000000000UL, 0x0000000008000000UL, 0
            x0200000008000400UL },
        { 0x0000000001000000UL, 0x0000020000000800UL, 0x0400000000000000UL, 0
            x0400000000000800UL, 0x0400020000000000UL, 0x0400020001000800UL, 0
            x0000020001000000UL, 0x0000000001000800UL, 0x0000020000000000UL, 0
            x0400000001000800UL, 0x0000020001000800UL, 0x0400020001000000UL, 0
            x0400000001000000UL, 0x0000000000000000UL, 0x0000000000000800UL, 0
            x0400020000000800UL },
        { 0x0000000000001000UL, 0x8000000020000000UL, 0x0000400000000000UL, 0
            x8000400000000000UL, 0x0000400020000000UL, 0x8000400020001000UL, 0
            x0000000020001000UL, 0x8000000000001000UL, 0x0000000020000000UL, 0
            x8000400000001000UL, 0x8000000020001000UL, 0x0000400020001000UL, 0
            x0000400000001000UL, 0x0000000000000000UL, 0x8000000000000000UL, 0
            x8000400020000000UL },
        { 0x1000000000000000UL, 0x0000800000002000UL, 0x0000000040000000UL, 0
            x0000800040000000UL, 0x0000000040002000UL, 0x1000800040002000UL, 0
            x1000000000002000UL, 0x1000800000000000UL, 0x0000000000002000UL, 0
            x1000800040000000UL, 0x1000800000002000UL, 0x1000000040002000UL, 0
            x1000000040000000UL, 0x0000000000000000UL, 0x0000800000000000UL, 0
            x0000800040002000UL },
        { 0x0000100000000000UL, 0x2000000080000000UL, 0x0000000000004000UL, 0
            x0000000080004000UL, 0x2000000000004000UL, 0x2000100080004000UL, 0
            x2000100000000000UL, 0x0000100080000000UL, 0x2000000000000000UL, 0
            x0000100080004000UL, 0x2000100080000000UL, 0x2000100000004000UL, 0
            x0000100000004000UL, 0x0000000000000000UL, 0x0000000080000000UL, 0
            x2000000080004000UL },
        { 0x0000000010000000UL, 0x0000200000008000UL, 0x4000000000000000UL, 0
            x4000000000008000UL, 0x4000200000000000UL, 0x4000200010008000UL, 0
            x0000200010000000UL, 0x0000000010008000UL, 0x0000200000000000UL, 0
            x4000000010008000UL, 0x0000200010008000UL, 0x4000200010000000UL, 0
            x4000000010000000UL, 0x0000000000000000UL, 0x0000000000008000UL, 0
            x4000200000008000UL }
};

void gift_64_table_generate_round_keys(uint64_t rks[restrict ROUNDS_GIFT_64],
                                       const uint64_t key[restrict 2])
{
        uint64_t key_state[] = {key[0], key[1]};
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
                int v = (key_state[0] >> 0 ) & 0xffff;
                int u = (key_state[0] >> 16) & 0xffff;

                // add round key (RK=U||V)
                rks[round] = 0UL;
                for (size_t i = 0; i < 16; i++) {
                        int key_bit_v   = (v >> i)  & 0x1;
                        int key_bit_u   = (u >> i)  & 0x1;
                        rks[round] ^= (uint64_t)key_bit_v << (i * 4 + 0);
                        rks[round] ^= (uint64_t)key_bit_u << (i * 4 + 1);
                }

                // add single bit
                rks[round] ^= 1UL << 63;

                // add round constants
                rks[round] ^= ((round_const[round] >> 0) & 0x1) << 3;
                rks[round] ^= ((round_const[round] >> 1) & 0x1) << 7;
                rks[round] ^= ((round_const[round] >> 2) & 0x1) << 11;
```

```c
                rks[round] ^= ((round_const[round] >> 3) & 0x1) << 15;
                rks[round] ^= ((round_const[round] >> 4) & 0x1) << 19;
                rks[round] ^= ((round_const[round] >> 5) & 0x1) << 23;

                // update key state
                int k0 = (key_state[0] >> 0 ) & 0xffffUL;
                int k1 = (key_state[0] >> 16) & 0xffffUL;
                k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
                k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
                key_state[0] >>= 32;
                key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
                key_state[1] >>= 32;
                key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
        }
}

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

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

        return new_cipher_state;
}

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

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

        return c;
}
```

B.1.2 Using **vperm**

Listing B.3: `gift_vec_sbox.h`

```c
#pragma once

#include <arm_neon.h>
#include <stdint.h>

#define ROUNDS_GIFT_64 28

// expose for benchmarking
uint8x16_t gift_64_vec_sbox_bits_pack(const uint64_t a);
uint64_t   gift_64_vec_sbox_bits_unpack(const uint8x16_t a);
uint8x16_t gift_64_vec_sbox_subcells(const uint8x16_t cipher_state);
uint8x16_t gift_64_vec_sbox_subcells_inv(const uint8x16_t cipher_state);
```

```c
uint8x16_t gift_64_vec_sbox_permute(const uint8x16_t cipher_state);
uint8x16_t gift_64_vec_sbox_permute_inv(const uint8x16_t cipher_state);
void       gift_64_vec_sbox_generate_round_keys(uint8x16_t rks[ROUNDS_GIFT_64],
                                                const uint64_t key[restrict 2]);

// construct tables
void gift_64_vec_sbox_init(void);

uint64_t gift_64_vec_sbox_encrypt(const uint64_t m,
                                  const uint8x16_t rks[restrict ROUNDS_GIFT_64]);
uint64_t gift_64_vec_sbox_decrypt(const uint64_t c,
                                  const uint8x16_t rks[restrict ROUNDS_GIFT_64]);
```

Listing B.4: `gift_vec_sbox.c`

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

#include "vec_sbox.h"

static uint64_t sbox_vec_u64[2] = {
        0x09030f060c040a01UL, 0x0e080005070b0d02UL
};

static uint64_t sbox_vec_inv_u64[2] = {
        0x0b040c020608000dUL, 0x050f09030a01070eUL
};

static uint8x16_t sbox_vec;
static uint8x16_t sbox_vec_inv;

// split S-box bits into vector lanes
uint8x16_t gift_64_vec_sbox_bits_pack(const uint64_t a)
{
        uint8x16_t v;
        v = vsetq_lane_u64(
        (uint64_t)((a >> 4 * 0) & 0xf) << 8 * 0 |
        (uint64_t)((a >> 4 * 1) & 0xf) << 8 * 1 |
        (uint64_t)((a >> 4 * 2) & 0xf) << 8 * 2 |
        (uint64_t)((a >> 4 * 3) & 0xf) << 8 * 3 |
        (uint64_t)((a >> 4 * 4) & 0xf) << 8 * 4 |
        (uint64_t)((a >> 4 * 5) & 0xf) << 8 * 5 |
        (uint64_t)((a >> 4 * 6) & 0xf) << 8 * 6 |
        (uint64_t)((a >> 4 * 7) & 0xf) << 8 * 7, v, 0);

        v = vsetq_lane_u64(
        (uint64_t)((a >> 4 * 8)  & 0xf) << 8 * 0 |
        (uint64_t)((a >> 4 * 9)  & 0xf) << 8 * 1 |
        (uint64_t)((a >> 4 * 10) & 0xf) << 8 * 2 |
        (uint64_t)((a >> 4 * 11) & 0xf) << 8 * 3 |
        (uint64_t)((a >> 4 * 12) & 0xf) << 8 * 4 |
        (uint64_t)((a >> 4 * 13) & 0xf) << 8 * 5 |
        (uint64_t)((a >> 4 * 14) & 0xf) << 8 * 6 |
        (uint64_t)((a >> 4 * 15) & 0xf) << 8 * 7, v, 1);

        return v;
}

// merge S-box bits into single uint64_t
uint64_t gift_64_vec_sbox_bits_unpack(const uint8x16_t v)
```

```
{
        uint64_t a = 0UL;
        uint64_t lane = vgetq_lane_u64(v, 0);
        a = (uint64_t)((lane >> 8 * 0) & 0xf) << 4 * 0 |
            (uint64_t)((lane >> 8 * 1) & 0xf) << 4 * 1 |
            (uint64_t)((lane >> 8 * 2) & 0xf) << 4 * 2 |
            (uint64_t)((lane >> 8 * 3) & 0xf) << 4 * 3 |
            (uint64_t)((lane >> 8 * 4) & 0xf) << 4 * 4 |
            (uint64_t)((lane >> 8 * 5) & 0xf) << 4 * 5 |
            (uint64_t)((lane >> 8 * 6) & 0xf) << 4 * 6 |
            (uint64_t)((lane >> 8 * 7) & 0xf) << 4 * 7;

        lane = vgetq_lane_u64(v, 1);
        a |= (uint64_t)((lane >> 8 * 0) & 0xf) << 4 * 8  |
             (uint64_t)((lane >> 8 * 1) & 0xf) << 4 * 9  |
             (uint64_t)((lane >> 8 * 2) & 0xf) << 4 * 10 |
             (uint64_t)((lane >> 8 * 3) & 0xf) << 4 * 11 |
             (uint64_t)((lane >> 8 * 4) & 0xf) << 4 * 12 |
             (uint64_t)((lane >> 8 * 5) & 0xf) << 4 * 13 |
             (uint64_t)((lane >> 8 * 6) & 0xf) << 4 * 14 |
             (uint64_t)((lane >> 8 * 7) & 0xf) << 4 * 15;

        return a;
}

static const size_t perm_64[] = {
        0, 17, 34, 51, 48, 1, 18, 35, 32, 49, 2, 19, 16, 33, 50, 3,
        4, 21, 38, 55, 52, 5, 22, 39, 36, 53, 6, 23, 20, 37, 54, 7,
        8, 25, 42, 59, 56, 9, 26, 43, 40, 57, 10, 27, 24, 41, 58, 11,
        12, 29, 46, 63, 60, 13, 30, 47, 44, 61, 14, 31, 28, 45, 62, 15
};

static const size_t perm_64_inv[] = {
        0, 5, 10, 15, 16, 21, 26, 31, 32, 37, 42, 47, 48, 53, 58, 63,
        12, 1, 6, 11, 28, 17, 22, 27, 44, 33, 38, 43, 60, 49, 54, 59,
        8, 13, 2, 7, 24, 29, 18, 23, 40, 45, 34, 39, 56, 61, 50, 55,
        4, 9, 14, 3, 20, 25, 30, 19, 36, 41, 46, 35, 52, 57, 62, 51
};

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
};

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

uint8x16_t gift_64_vec_sbox_subcells_inv(const uint8x16_t cipher_state)
{
        return vqtbl1q_u8(sbox_vec_inv, cipher_state);
}

uint8x16_t gift_64_vec_sbox_permute(const uint8x16_t cipher_state)
```

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

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

        return gift_64_vec_sbox_bits_pack(new_cipher_state);
}

uint8x16_t gift_64_vec_sbox_permute_inv(const uint8x16_t cipher_state)
{
        // collect into 64-bit register (faster)
        uint64_t new_cipher_state = 0;
        uint64_t boxes[2];
        vst1q_u64(boxes, cipher_state);

        // S-box 0-7
        for (size_t box = 0; box < 8; box++) {
                for (size_t i = 0; i < 4; i++) {
                        const int bit = (boxes[0] >> (box * 8 + i)) & 0x1;
                        new_cipher_state |= (uint64_t)bit << perm_64_inv[box * 4 + i];
                }
        }

        // S-box 8-15
        for (size_t box = 0; box < 8; box++) {
                for (size_t i = 0; i < 4; i++) {
                        const int bit = (boxes[1] >> (box * 8 + i)) & 0x1;
                        new_cipher_state |= (uint64_t)bit << perm_64_inv[(box + 8) * 4
                            + i];
                }
        }

        return gift_64_vec_sbox_bits_pack(new_cipher_state);
}

void gift_64_vec_sbox_generate_round_keys(uint8x16_t rks[ROUNDS_GIFT_64],
                                          const uint64_t key[2])
{
        uint64_t key_state[] = {key[0], key[1]};
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
                const int v = (key_state[0] >> 0 ) & 0xffff;
                const int u = (key_state[0] >> 16) & 0xffff;

                // add round key (RK=U||V)
                uint64_t round_key = 0UL;
                for (size_t i = 0; i < 16; i++) {
                        const int key_bit_v   = (v >> i)  & 0x1;
                        const int key_bit_u   = (u >> i)  & 0x1;
                        round_key ^= (uint64_t)key_bit_v << (i * 4 + 0);
                        round_key ^= (uint64_t)key_bit_u << (i * 4 + 1);
                }

                // add single bit
                round_key ^= 1UL << 63;
```

```
                // add round constants
                round_key ^= ((round_const[round] >> 0) & 0x1) << 3;
                round_key ^= ((round_const[round] >> 1) & 0x1) << 7;
                round_key ^= ((round_const[round] >> 2) & 0x1) << 11;
                round_key ^= ((round_const[round] >> 3) & 0x1) << 15;
                round_key ^= ((round_const[round] >> 4) & 0x1) << 19;
                round_key ^= ((round_const[round] >> 5) & 0x1) << 23;

                // pack into vector register
                rks[round] = gift_64_vec_sbox_bits_pack(round_key);

                // update key state
                int k0 = (key_state[0] >> 0 ) & 0xffffUL;
                int k1 = (key_state[0] >> 16) & 0xffffUL;
                k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
                k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
                key_state[0] >>= 32;
                key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
                key_state[1] >>= 32;
                key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
        }
}

void gift_64_vec_sbox_init(void)
{
        // construct sbox_vec
        sbox_vec = vld1q_u64(sbox_vec_u64);

        // construct sbox_vec_inv
        sbox_vec_inv = vld1q_u64(sbox_vec_inv_u64);
}

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

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

        // unpack
        return gift_64_vec_sbox_bits_unpack(c);
}

uint64_t gift_64_vec_sbox_decrypt(const uint64_t c,
                                  const uint8x16_t rks[restrict ROUNDS_GIFT_64])
{
        // pack into vector register
        uint8x16_t m = gift_64_vec_sbox_bits_pack(c);

        // round loop (in reverse)
        for (int round = ROUNDS_GIFT_64 - 1; round >= 0; round--) {
                m = veorq_u8(m, rks[round]);
                m = gift_64_vec_sbox_permute_inv(m);
                m = gift_64_vec_sbox_subcells_inv(m);
        }
```

```c

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

B.1.3 Bitslicing

Listing B.5: `gift_vec_sliced.h`

```c
#pragma once

#include <stdint.h>
#include <arm_neon.h>

#define ROUNDS_GIFT_64 28

// expose for benchmarking
uint8x16_t shl(const uint8x16_t v, const int n);
uint8x16_t shr(const uint8x16_t v, const int n);
void gift_64_vec_sliced_swapmove(uint8x16_t *restrict a, uint8x16_t *restrict b,
                                 const uint8x16_t m, const int n);
void gift_64_vec_sliced_bits_pack(uint8x16x4_t m[restrict 2]);
void gift_64_vec_sliced_bits_unpack(uint8x16x4_t m[restrict 2]);

void gift_64_vec_sliced_subcells(uint8x16x4_t cipher_state[restrict 2]);
void gift_64_vec_sliced_subcells_inv(uint8x16x4_t cipher_state[restrict 2]);
void gift_64_vec_sliced_permute(uint8x16x4_t cipher_state[restrict 2]);
void gift_64_vec_sliced_permute_inv(uint8x16x4_t cipher_state[2]);
void gift_64_vec_sliced_generate_round_keys(uint8x16x4_t rks[restrict ROUNDS_GIFT_64
    ][2],
                                            const uint64_t key[restrict 2]);

void gift_64_vec_sliced_init(void);

void gift_64_vec_sliced_encrypt(uint64_t c[restrict 16],
                                const uint64_t m[restrict 16],
                                const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2]);
void gift_64_vec_sliced_decrypt(uint64_t m[restrict 16],
                                const uint64_t c[restrict 16],
                                const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2]);
```

Listing B.6: `gift_vec_sliced.c`

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

#include "vec_sliced.h"

static uint64_t pack_shf_u64[] = {
        0x1303120211011000UL, 0x1707160615051404UL, // S0/S1/S2/S3
        0x1b0b1a0a19091808UL, 0x1f0f1e0e1d0d1c0cUL, // S4/S5/S6/S7
};

static uint64_t pack_shf_inv_u64[] = {
        0x0e0c0a0806040200UL, 0x1e1c1a1816141210UL, // S0/S1/S2/S3
        0x0f0d0b0907050301UL, 0x1f1d1b1917151311UL, // S4/S5/S6/S7
};
```

```

static uint64_t perm_u64[] = {
        0x0f0b07030c080400UL, 0x0d0905010e0a0602UL, // S0/S4
        0x0c0804000d090501UL, 0x0e0a06020f0b0703UL, // S1/S5
        0x0d0905010e0a0602UL, 0x0f0b07030c080400UL, // S2/S6
        0x0e0a06020f0b0703UL, 0x0c0804000d090501UL  // S3/S7
};


static uint64_t perm_inv_u64[] = {
        0x05090d0104080c00UL, 0x070b0f03060a0e02UL, // S0/S4
        0x090d0105080c0004UL, 0x0b0f03070a0e0206UL, // S1/S5
        0x0d0105090c000408UL, 0x0f03070b0e02060aUL, // S2/S6
        0x0105090d0004080cUL, 0x03070b0f02060a0eUL  // S3/S7
};

static uint8x16x2_t pack_shf;
static uint8x16x2_t pack_shf_inv;
static uint8x16x4_t perm;
static uint8x16x4_t perm_inv;

static uint8x16_t pack_mask_0;
static uint8x16_t pack_mask_1;
static uint8x16_t pack_mask_2;

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

/*
uint8x16_t shl(const uint8x16_t v, const int n) */
#define shl(_a, v, n)                                                   \
{                                                                       \
        uint64x2_t _overflow = vshrq_n_u64(v, 64 - n);                  \
        _overflow = vextq_u64(vdupq_n_u64(0x0), _overflow, 1);          \
        _a = vorrq_u8(vshlq_n_u64(v, n), _overflow);                    \
}

/*
uint8x16_t shr(const uint8x16_t v, const int n) */
#define shr(_a, v, n)                                                   \
{                                                                       \
        uint64x2_t _overflow = vshlq_n_u64(v, 64 - n);                  \
        _overflow = vextq_u64(_overflow, vdupq_n_u64(0x0), 1);          \
        _a = vorrq_u8(vshrq_n_u64(v, n), _overflow);                    \
}

/* implemented as a macro so we can use vshlq_n_u8 with variable n
void gift_64_vec_sliced_swapmove(uint8x16_t *restrict a, uint8x16_t *restrict b,
                                 const uint8x16_t m, const int n) */
#define gift_64_vec_sliced_swapmove(a, b, m, n)                         \
{                                                                       \
        uint8x16_t _a;                                                  \
        shr(_a, a, n);                                                  \
```

```
        const uint8x16_t _t = vandq_u8(veorq_u8(_a, b), m);                   \
        b = veorq_u8(b, _t);                                                  \
        shl(_a, _t, n);                                                       \
        a = veorq_u8(a, _a);                                                  \
}

void gift_64_vec_sliced_bits_pack(uint8x16x4_t m[restrict 2])
{
        // take care not to shift mask bits out of the register
        gift_64_vec_sliced_swapmove(m[0].val[0], m[0].val[1], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(m[0].val[2], m[0].val[3], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(m[1].val[0], m[1].val[1], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(m[1].val[2], m[1].val[3], pack_mask_0, 1);

        gift_64_vec_sliced_swapmove(m[0].val[0], m[0].val[2], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(m[0].val[1], m[0].val[3], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(m[1].val[0], m[1].val[2], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(m[1].val[1], m[1].val[3], pack_mask_1, 2);

        // make bytes (a0 b0 c0 d0 a4 b4 c4 d4 -> a0 b0 c0 d0 e0 f0 g0 h0)
        gift_64_vec_sliced_swapmove(m[0].val[0], m[1].val[0], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(m[0].val[2], m[1].val[2], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(m[0].val[1], m[1].val[1], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(m[0].val[3], m[1].val[3], pack_mask_2, 4);

        // same plaintext slice bits into same register (so we only have to do
        // what we are doing here once instead of every round)
        const uint8x16x2_t pairs[4] = {
                { .val = { m[0].val[0], m[1].val[0] }},
                { .val = { m[0].val[1], m[1].val[1] }},
                { .val = { m[0].val[2], m[1].val[2] }},
                { .val = { m[0].val[3], m[1].val[3] }},
        };

        m[0].val[0] = vqtbl2q_u8(pairs[0], pack_shf.val[0]);
        m[0].val[1] = vqtbl2q_u8(pairs[1], pack_shf.val[0]);
        m[0].val[2] = vqtbl2q_u8(pairs[2], pack_shf.val[0]);
        m[0].val[3] = vqtbl2q_u8(pairs[3], pack_shf.val[0]);

        m[1].val[0] = vqtbl2q_u8(pairs[0], pack_shf.val[1]);
        m[1].val[1] = vqtbl2q_u8(pairs[1], pack_shf.val[1]);
        m[1].val[2] = vqtbl2q_u8(pairs[2], pack_shf.val[1]);
        m[1].val[3] = vqtbl2q_u8(pairs[3], pack_shf.val[1]);
}

void gift_64_vec_sliced_bits_unpack(uint8x16x4_t m[restrict 2])
{
        const uint8x16x2_t pairs[4] = {
                { .val = { m[0].val[0], m[1].val[0] }},
                { .val = { m[0].val[1], m[1].val[1] }},
                { .val = { m[0].val[2], m[1].val[2] }},
                { .val = { m[0].val[3], m[1].val[3] }},
        };

        m[0].val[0] = vqtbl2q_u8(pairs[0], pack_shf_inv.val[0]);
        m[0].val[1] = vqtbl2q_u8(pairs[1], pack_shf_inv.val[0]);
        m[0].val[2] = vqtbl2q_u8(pairs[2], pack_shf_inv.val[0]);
        m[0].val[3] = vqtbl2q_u8(pairs[3], pack_shf_inv.val[0]);

        m[1].val[0] = vqtbl2q_u8(pairs[0], pack_shf_inv.val[1]);
        m[1].val[1] = vqtbl2q_u8(pairs[1], pack_shf_inv.val[1]);
        m[1].val[2] = vqtbl2q_u8(pairs[2], pack_shf_inv.val[1]);
```

```
        m[1].val[3] = vqtbl2q_u8(pairs[3], pack_shf_inv.val[1]);

        // take care not to shift mask bits out of the register
        gift_64_vec_sliced_swapmove(m[0].val[0], m[0].val[1], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(m[0].val[2], m[0].val[3], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(m[1].val[0], m[1].val[1], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(m[1].val[2], m[1].val[3], pack_mask_0, 1);

        gift_64_vec_sliced_swapmove(m[0].val[0], m[0].val[2], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(m[0].val[1], m[0].val[3], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(m[1].val[0], m[1].val[2], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(m[1].val[1], m[1].val[3], pack_mask_1, 2);

        // make bytes (a0 b0 c0 d0 a4 b4 c4 d4 -> a0 b0 c0 d0 e0 f0 g0 h0)
        gift_64_vec_sliced_swapmove(m[0].val[0], m[1].val[0], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(m[0].val[2], m[1].val[2], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(m[0].val[1], m[1].val[1], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(m[0].val[3], m[1].val[3], pack_mask_2, 4);
}

void gift_64_vec_sliced_subcells(uint8x16x4_t cs[restrict 2])
{
        cs[0].val[1] = veorq_u8(cs[0].val[1],
                                vandq_u8(cs[0].val[0], cs[0].val[2]));
        uint8x16_t t = veorq_u8(cs[0].val[0],
                                vandq_u8(cs[0].val[1], cs[0].val[3]));
        cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
        cs[0].val[0] = veorq_u8(cs[0].val[3], cs[0].val[2]);
        cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
        cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
        cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
        cs[0].val[3] = t;

        cs[1].val[1] = veorq_u8(cs[1].val[1],
                                vandq_u8(cs[1].val[0], cs[1].val[2]));
        t            = veorq_u8(cs[1].val[0],
                                vandq_u8(cs[1].val[1], cs[1].val[3]));
        cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
        cs[1].val[0] = veorq_u8(cs[1].val[3], cs[1].val[2]);
        cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
        cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
        cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
        cs[1].val[3] = t;
}

void gift_64_vec_sliced_subcells_inv(uint8x16x4_t cs[restrict 2])
{
        uint8x16_t t = cs[0].val[3];
        cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
        cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
        cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
        cs[0].val[3] = veorq_u8(cs[0].val[0], cs[0].val[2]);
        cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
        cs[0].val[0] = veorq_u8(t, vandq_u8(cs[0].val[1], cs[0].val[3]));
        cs[0].val[1] = veorq_u8(cs[0].val[1],
                                vandq_u8(cs[0].val[0], cs[0].val[2]));

        t            = cs[1].val[3];
        cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
        cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
        cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
        cs[1].val[3] = veorq_u8(cs[1].val[0], cs[1].val[2]);
```

```
        cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
        cs[1].val[0] = veorq_u8(t, vandq_u8(cs[1].val[1], cs[1].val[3]));
        cs[1].val[1] = veorq_u8(cs[1].val[1],
                                vandq_u8(cs[1].val[0], cs[1].val[2]));
}

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

        cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm.val[0]);
        cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm.val[1]);
        cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm.val[2]);
        cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm.val[3]);
}

void gift_64_vec_sliced_permute_inv(uint8x16x4_t cs[restrict 2])
{
        cs[0].val[0] = vqtbl1q_u8(cs[0].val[0], perm_inv.val[0]);
        cs[0].val[1] = vqtbl1q_u8(cs[0].val[1], perm_inv.val[1]);
        cs[0].val[2] = vqtbl1q_u8(cs[0].val[2], perm_inv.val[2]);
        cs[0].val[3] = vqtbl1q_u8(cs[0].val[3], perm_inv.val[3]);

        cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm_inv.val[0]);
        cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm_inv.val[1]);
        cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm_inv.val[2]);
        cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm_inv.val[3]);
}

void gift_64_vec_sliced_generate_round_keys(uint8x16x4_t rks[restrict ROUNDS_GIFT_64
    ][2],
                                            const uint64_t key[restrict 2])
{
        uint64_t key_state[] = {key[0], key[1]};
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
                const int v = (key_state[0] >> 0 ) & 0xffff;
                const int u = (key_state[0] >> 16) & 0xffff;

                // add round key (RK=U||V)
                // (slice 2 stays unused)
                uint64_t rk[6] = { 0x0UL };
                for (size_t i = 0; i < 8; i++) {
                        int key_bit_v   = (v >> (i + 0)) & 0x1;
                        int key_bit_u   = (u >> (i + 0)) & 0x1;
                        rk[0]           ^= (uint64_t)key_bit_v << (i * 8);
                        rk[2]           ^= (uint64_t)key_bit_u << (i * 8);

                        key_bit_v       = (v >> (i + 8)) & 0x1;
                        key_bit_u       = (u >> (i + 8)) & 0x1;
                        rk[1]           ^= (uint64_t)key_bit_v << (i * 8);
                        rk[3]           ^= (uint64_t)key_bit_u << (i * 8);
                }

                // add single bit
                rk[5] ^= 1UL << (7 * 8);

                // add round constants
                rk[4] ^= ((uint64_t)(round_const[round] >> 0) & 0x1) << (0 * 8);
                rk[4] ^= ((uint64_t)(round_const[round] >> 1) & 0x1) << (1 * 8);
```

```
                rk[4] ^= ((uint64_t)(round_const[round] >> 2) & 0x1) << (2 * 8);
                rk[4] ^= ((uint64_t)(round_const[round] >> 3) & 0x1) << (3 * 8);
                rk[4] ^= ((uint64_t)(round_const[round] >> 4) & 0x1) << (4 * 8);
                rk[4] ^= ((uint64_t)(round_const[round] >> 5) & 0x1) << (5 * 8);

                // extend bits to bytes
                for (size_t i = 0; i < 6; i++) {
                        rk[i] |= rk[i] << 1;
                        rk[i] |= rk[i] << 2;
                        rk[i] |= rk[i] << 4;
                }

                rks[round][0].val[0] = vsetq_lane_u64(rk[0], rks[round][0].val[0], 0);
                rks[round][0].val[0] = vsetq_lane_u64(rk[1], rks[round][0].val[0], 1);
                rks[round][0].val[1] = vsetq_lane_u64(rk[2], rks[round][0].val[1], 0);
                rks[round][0].val[1] = vsetq_lane_u64(rk[3], rks[round][0].val[1], 1);
                rks[round][0].val[2] = vdupq_n_u8(0);
                rks[round][0].val[3] = vsetq_lane_u64(rk[4], rks[round][0].val[3], 0);
                rks[round][0].val[3] = vsetq_lane_u64(rk[5], rks[round][0].val[3], 1);
                rks[round][1]        = rks[round][0];

                // update key state
                int k0 = (key_state[0] >> 0 ) & 0xffffUL;
                int k1 = (key_state[0] >> 16) & 0xffffUL;
                k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
                k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
                key_state[0] >>= 32;
                key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
                key_state[1] >>= 32;
                key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
        }
}

void gift_64_vec_sliced_init(void)
{
        // bit packing shuffle
        pack_shf = vld1q_u8_x2((uint8_t*)&pack_shf_u64[0]);

        // inverse bit packing shuffle
        pack_shf_inv = vld1q_u8_x2((uint8_t*)&pack_shf_inv_u64[0]);

        // permutations
        perm = vld1q_u8_x4((uint8_t*)&perm_u64[0]);

        // inverse permutations
        perm_inv = vld1q_u8_x4((uint8_t*)&perm_inv_u64[0]);

        // packing masks
        pack_mask_0 = vdupq_n_u8(0x55);
        pack_mask_1 = vdupq_n_u8(0x33);
        pack_mask_2 = vdupq_n_u8(0x0f);
}

void gift_64_vec_sliced_encrypt(uint64_t c[restrict 16],
                                const uint64_t m[restrict 16],
                                const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2])
{
        uint8x16x4_t s[2];
        s[0] = vld1q_u8_x4((uint8_t*)&m[0]);
        s[1] = vld1q_u8_x4((uint8_t*)&m[8]);
        gift_64_vec_sliced_bits_pack(s);

```

```
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
                gift_64_vec_sliced_subcells(s);
                gift_64_vec_sliced_permute(s);

                // round key addition
                s[0].val[0] = veorq_u8(s[0].val[0], rks[round][0].val[0]);
                s[0].val[1] = veorq_u8(s[0].val[1], rks[round][0].val[1]);
                s[0].val[2] = veorq_u8(s[0].val[2], rks[round][0].val[2]);
                s[0].val[3] = veorq_u8(s[0].val[3], rks[round][0].val[3]);
                s[1].val[0] = veorq_u8(s[1].val[0], rks[round][1].val[0]);
                s[1].val[1] = veorq_u8(s[1].val[1], rks[round][1].val[1]);
                s[1].val[2] = veorq_u8(s[1].val[2], rks[round][1].val[2]);
                s[1].val[3] = veorq_u8(s[1].val[3], rks[round][1].val[3]);
        }

        gift_64_vec_sliced_bits_unpack(s);
        vst1q_u8_x4((uint8_t*)&c[0], s[0]);
        vst1q_u8_x4((uint8_t*)&c[8], s[1]);
}

void gift_64_vec_sliced_decrypt(uint64_t m[restrict 16],
                                const uint64_t c[restrict 16],
                                const uint8x16x4_t rks[restrict ROUNDS_GIFT_64][2])
{
        uint8x16x4_t s[2];
        s[0] = vld1q_u8_x4((uint8_t*)&c[0]);
        s[1] = vld1q_u8_x4((uint8_t*)&c[8]);
        gift_64_vec_sliced_bits_pack(s);

        for (int round = ROUNDS_GIFT_64 - 1; round >= 0; round--) {
                // round key addition
                s[0].val[0] = veorq_u8(s[0].val[0], rks[round][0].val[0]);
                s[0].val[1] = veorq_u8(s[0].val[1], rks[round][0].val[1]);
                s[0].val[2] = veorq_u8(s[0].val[2], rks[round][0].val[2]);
                s[0].val[3] = veorq_u8(s[0].val[3], rks[round][0].val[3]);
                s[1].val[0] = veorq_u8(s[1].val[0], rks[round][1].val[0]);
                s[1].val[1] = veorq_u8(s[1].val[1], rks[round][1].val[1]);
                s[1].val[2] = veorq_u8(s[1].val[2], rks[round][1].val[2]);
                s[1].val[3] = veorq_u8(s[1].val[3], rks[round][1].val[3]);

                gift_64_vec_sliced_permute_inv(s);
                gift_64_vec_sliced_subcells_inv(s);
        }

        gift_64_vec_sliced_bits_unpack(s);
        vst1q_u8_x4((uint8_t*)&m[0], s[0]);
        vst1q_u8_x4((uint8_t*)&m[8], s[1]);
}
```

Appendix C

Lorem dolor

Bibliography

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

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

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

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

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

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

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

[8] 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.

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

[10] 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.

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