

# Fast Skinny-128 SIMD Implementations for Sequential Modes of Operation

Alexandre Adomnicai<sup>1</sup>, Kazuhiko Minematsu<sup>2,3</sup>, and Maki Shigeri<sup>4</sup>

<sup>1</sup> CryptoNext Security, Paris, France, [alex.adomnicai@gmail.com](mailto:alex.adomnicai@gmail.com)

<sup>2</sup> NEC, Kawasaki, Japan, [k-minematsu@nec.com](mailto:k-minematsu@nec.com)

<sup>3</sup> Yokohama National University, Japan

<sup>4</sup> NEC Solution Innovators, Hokuriku, Japan, [m-shigeri-pb@nec.com](mailto:m-shigeri-pb@nec.com)

**Keywords:** Skinny · Romulus · NIST LWC · SIMD

**Abstract.** This paper reports new software implementation results for the Skinny-128 tweakable block ciphers on various SIMD architectures. More precisely, we introduce a decomposition of the 8-bit S-box into four 4-bit S-boxes in order to take advantage of vector permute instructions, leading to significant performance improvements over previous constant-time implementations. Since our approach is of particular interest when Skinny-128 is used in sequential modes of operation, we also report how it benefits to the Romulus authenticated encryption scheme, a finalist of the NIST LWC standardization process.

## 1 Introduction

The Internet of Things (IoT) does not come without its challenges, and security concerns remain a major barrier to its adoption. One of the technical considerations is the efficiency/security trade-off of cryptographic implementations on IoT devices. The restrictions in terms of power and memory introduce challenges that do not exist when using cryptography in more conventional IT platforms. Moreover, environments in which IoT devices are deployed make these devices vulnerable to unforeseen physical threats where attackers may tamper with them directly. It implies that cryptographic implementations must show some resilience against physical attacks to avoid key recoveries that might lead to a compromised network, as already demonstrated in practice [27]. Therefore, numerous symmetric-key ciphers have been proposed by taking all these aspects into consideration at the design level. They are categorized as lightweight cryptography, aiming at providing better hardware and/or software implementation properties on embedded devices. In this context, the National Institute of Standards and Technology (NIST) initiated a process that started in 2018, with the goal of selecting the future Authenticated Encryption with Associated Data (AEAD) standard(s) for constrained environments [23]. AEAD algorithms ensure confidentiality, integrity, and authenticity of data in a single primitive. Romulus [19] is one of the ten proposals currently competing for standardization

in the final round. It is based on *Skinny* [7], a tweakable block cipher standardized in ISO/IEC 18033-7. If *Skinny* shows outstanding results when implemented in hardware, the picture is more mixed when it comes to software. Although recent works have been undertaken to optimize its performance on 32-bit microcontrollers, for example by Adomnicai and Peyrin [2], it is not clear what is the best implementation strategy on more advanced architectures. Although the main goal of lightweight cryptography is to provide optimized encryption and authentication solutions for resource-constrained devices (e.g. low-cost microcontrollers), it will be inevitably deployed on more sophisticated platforms for interoperability purposes. For instance, many IoT networks adopt a star topology where numerous low-end devices communicate with a single server that has to decrypt received data. Mobile devices (e.g. smartphones, tablets) are also commonly used for network monitoring purposes, requiring to handle secure communications with many nodes simultaneously. Therefore, software performance of lightweight cryptography does matter on mid-range to high-end microprocessors as well. Most of these platforms are now equipped with single instruction multiple data (SIMD) units whose goal is to vectorize calculations by performing the same operation on multiple data operands concurrently. On Intel processors, SIMD units have been available since the advent of the MMX instruction set architecture extension, initially designed to speed up the performance of multimedia applications. Similarly, ARM introduced SIMD extensions with the NEON technology being implemented on all ARM Cortex-A series processors. To date, the best software *Skinny*-128 implementation results reported on SIMD architectures are obtained by processing many 128-bit blocks in parallel (e.g. 64 using AVX2 [7]) thanks to bitslicing. While relying on implementations that operate on a large amount of data at once is not necessarily relevant in the context of IoT, where payloads are usually a few dozen of bytes only, it is even less of interest for sequential (i.e. non-parallelizable) modes of operation as in *Romulus*.

*Our contributions.* In this paper, we optimize the performance of *Skinny*-128 for sequential modes of operation on SIMD platforms, with a focus on ARM processors with NEON technology. First, we briefly review various publicly available software implementations of *Skinny*-128 and highlight that the 8-bit S-box component is the most-time consuming part of the encryption process. To address this issue, we propose an optimization trick which consists in decomposing the S-box into smaller ones so that we can take advantage of SIMD-specific vector permute instructions to reach competitive performance without introducing secret-dependent timing variations. While it has already been shown that lightweight ciphers with 4-bit S-boxes can highly benefit from such instructions [8,5], it is less trivial for designs with larger (e.g. 8-bit) S-boxes. Still, a similar implementation trick has been first proposed by Hamburg for AES on Intel processors [18]. Our work shows that this is also quite effective for *Skinny*-128 by introducing a novel decomposition of its 8-bit S-box into 4 tables. As a result, we observe a speedup by a factor that ranges from 1.5 to 3.5 depending on the computing platform, compared to the fixsliced implementation strategy [2], which is currently the fastest option for *Skinny*-128 when used in non-parallelizable operating

modes. We also port our implementations on Intel platforms, improving the performance by a factor of 4. These results straightforwardly apply to **Romulus** as shown by our benchmarks. Finally, our software implementations are released into the public domain at <https://github.com/aadomn/skinny>.

## 2 Skinny in software

### 2.1 The Skinny-128 tweakable block ciphers

A tweakable block cipher is family of permutations where both key and tweak are used to select a permutation. Skinny follows the tweakkey framework [21] which treats the tweak and the key in the same way in a structure called *tweakkey*. It is up to the user what part of this tweakkey will be key material and/or tweak material. The internal state of Skinny-128 as well as the tweakkey states consist of a  $4 \times 4$  square arrays of bytes. The number of tweakkey states ranges from one to three (namely  $TK1$ ,  $TK2$  and  $TK3$ ), and is directly linked to the quantity of tweakkey material which is either 128, 256, or 384 bits. The corresponding versions are denoted by Skinny-128-128, Skinny-128-256, and Skinny-128-384, and are composed of 40, 48, and 56 encryption rounds, respectively. One encryption round is itself composed of five operations in the following order: **SubCells**, **AddConstants**, **AddRoundTweakey**, **ShiftRows** and **MixColumns** as illustrated in Figure 1.

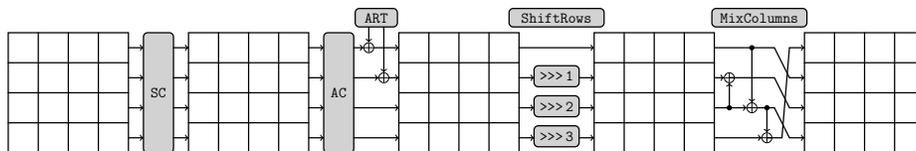


Fig. 1: The Skinny round function (from [20])

**SubCells** refers to the non-linear layer and consists in applying an 8-bit S-box, depicted in Figure 2, to each byte individually.

**AddConstants** consists in combining three round constants  $c_0, c_1, c_2$  with the three topmost bytes in the first state column using bitwise exclusive-OR (XOR). The round constants are defined as below:

$$\begin{aligned} c_0 &= 0 \parallel 0 \parallel 0 \parallel 0 \parallel 0 \parallel rc_3 \parallel rc_2 \parallel rc_1 \parallel rc_0 \\ c_1 &= 0 \parallel rc_5 \parallel rc_4 \\ c_2 &= 0 \parallel 1 \parallel 1 \end{aligned}$$

where  $rc_i$  are defined by the following 6-bit LFSR

$$(rc_5 \parallel rc_4 \parallel rc_3 \parallel rc_2 \parallel rc_1 \parallel rc_0) \rightarrow (rc_4 \parallel rc_3 \parallel rc_2 \parallel rc_1 \parallel rc_0 \parallel rc_5 \oplus rc_4 \oplus 1).$$

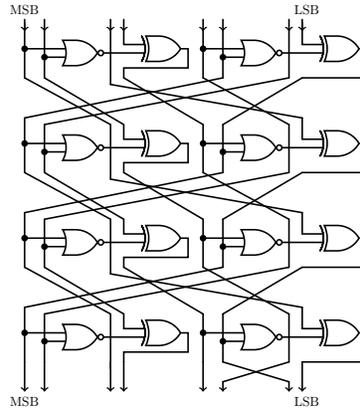


Fig. 2: The Skinny-128 S-box (from [20])

**AddRoundTweakey** extracts the two topmost rows of each tweakey array and adds them to the internal state using bitwise XOR. Then all tweakey arrays are updated by applying a byte permutation to the state and an 8-bit LFSR to each byte, as illustrated in Figure 3. Finally, **ShiftRows** and **MixColumns** refer to the linear layer, ensuring diffusion within the state.

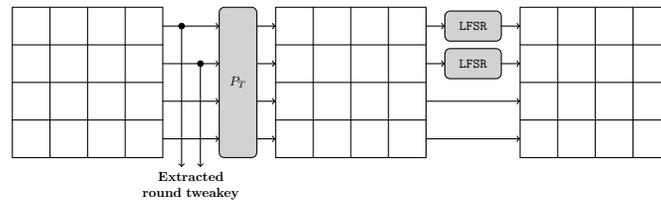


Fig. 3: Tweakey state update (from [20])

## 2.2 Publicly available software implementations

The original publication of **Skinny** reports efficient bitsliced implementations<sup>1</sup> with **Skinny-128-128** running at 3.78 and 3.43 cycles per byte (cpb) on Haswell and Skylake architectures respectively, by taking advantage of Intel AVX2 instructions. However, it requires to process 64 blocks in parallel which makes it quite inefficient for sequential modes of operation since it would basically decrease computation speed by a factor of 64.

Regarding 32-bit implementations, as for the AES T-tables, it is possible to combine multiple steps of the round function into table lookups. This has

<sup>1</sup> [https://github.com/kste/skinny\\_avx](https://github.com/kste/skinny_avx)

been investigated in [14] in order to optimize ForkAE [3], which is based on the Skinny round function, but there are no implementation results reported for Skinny itself. For platforms where cache-based attacks are a concern, one should favor constant-time implementations to avoid timing side-channels that could leak information about the secret key. An application of the fixslicing technique to Skinny-128 was recently proposed in this regard [2]. Fixslicing is a specific instance of bitslicing where at least one slice remains fixed (potentially leading to an alternative representation for a few rounds), with the aim of optimizing the diffusion layer. It was originally proposed by Adomnicai, Najm and Peyrin [1] with an application to GIFT-COFB, a NIST LWC finalist based on the GIFT block cipher [6] and the COFB mode of operation [13]. According to [2], fixsliced Skinny-128 runs around 191 cpb on ARMv7-M when processing a single block at a time (with precomputed round tweakeys). Note that there is also a constant-time implementation from Weatherley which stores the internal state in a byte-wise fashion but implements the S-box in a bitsliced manner by means of bitmasks and bitwise operations [31]. While it is around 2.5 times slower than the fixsliced version on ARMv7-M when processing a single block at time [2], it requires half RAM to store the round tweakeys. Still, the improvement factor should be significantly reduced on 64-bit platforms since the byte-wise representation can benefit from larger registers to apply the S-box on only two 64-bit words instead of four 32-bit words.

There has also been work on Skinny-128 optimizations using the ARM NEON extension, with the objective to enhance the performance of the ForkAE lightweight encryption scheme [14]. The implementation strategy is the same as [31]: the authors use a single 128-bit NEON register to store the entire internal state and rely on a bitsliced approach for the S-box, requiring 63 instructions in total<sup>2</sup>. Because no implementation results are reported for Skinny-128 itself, it is not clear how it performs compared to the fixsliced<sup>3</sup> and byte-wise<sup>4</sup> implementations. To clarify this point, we performed a simple benchmark on the following three ARM CPUs implementing the NEON extension: the Cortex-A7, Cortex-A53 and Cortex-A72 processors briefly described hereafter. We used the SUPERCOP benchmarking suite [9] using gcc 8.3.0. The results are reported in Table 1.

*ARM Cortex-A7.* The Cortex-A7 is an in-order pipeline CPU core with moderate performance but an extremely small die size and very low power consumption. It was initially introduced for entry-level smartphones and now progressively finds its place in system-on-chips dedicated to the IoT [22]. It is based on the 32-bit ARMv7-A architecture which has 16 32-bit general-purpose ARM registers  $R_0$ - $R_{15}$  and 32 64-bit NEON registers  $D_0$ - $D_{31}$ . These NEON registers can also be manipulated as 16 128-bit registers  $Q_0$ - $Q_{15}$  where each  $Q_i$  maps to the

<sup>2</sup> [https://github.com/ArneDeprez1/ForkAE-SW/blob/master/Neon\\_SIMD/sbox\\_neon.S](https://github.com/ArneDeprez1/ForkAE-SW/blob/master/Neon_SIMD/sbox_neon.S)

<sup>3</sup> [https://github.com/aadomn/skinny/tree/master/crypto\\_tbc/skinny128/1\\_block/opt32](https://github.com/aadomn/skinny/tree/master/crypto_tbc/skinny128/1_block/opt32)

<sup>4</sup> <https://github.com/rweather/skinny-c>

pair  $(D_{2i}, D_{2i+1})$ . For our benchmarks, we use the Raspberry Pi 2 Model B featuring the 900MHz quad-core Cortex-A7 Broadcom BCM2836 chipset running Raspbian 10 (buster).

*ARM Cortex-A53.* The Cortex-A53 is one of the first two processors implementing the ARMv8-A architecture and is typically found in entry-level smartphone and other embedded devices. ARMv8-A introduces a new 64-bit instruction set known as A64 which operates in the 64-bit execution state called AArch64. It also provides a 32-bit execution state called AArch32 to ensure backward compatibility with ARMv7-A. While AArch32 has the same number of registers as ARMv7-A, AArch64, by comparison, has 31 64-bit ARM registers  $X_0$ - $X_{30}$  and 32 128-bit NEON registers  $V_0$ - $V_{31}$ . For our benchmarks, we use the Raspberry Pi 3 Model B featuring the 1.2 GHz quad-core Cortex-A53 Broadcom BCM2837 chipset running Debian 10 (buster).

*ARM Cortex-A72.* Finally, the Cortex-A72 is based on the ARMv8-A architecture and is designed for the mobile market. It is considered as a high performant core which is often combined with lower performance processors such as the Cortex-A53 to achieve better tradeoffs between energy and performance. For our benchmarks, we use the Raspberry Pi 4 Model B featuring the 1.5GHz quad-core Cortex-A72 Broadcom BCM2711 chipset running Debian 10 (buster).

Algorithm	Implementation	Speed (clock cycles)		
		A7	A53	A72
Skinny-128-384 block encryption	Fixsliced [2]	<b>5 492</b>	<b>2 814</b>	<b>2 655</b>
	Byte-wise [31]	10 328	3 055	2 993
	ARMv7-A NEON [14]	10 563	-	-
Skinny-128-384 tweakey schedule	Fixsliced [2]	<b>3 901</b>	3 210	2 082
	Byte-wise [31]	7 855	<b>2 294</b>	<b>1 568</b>
	ARMv7-A NEON [14]	4 127	-	-

Table 1: Performance comparison between three Skinny-128 implementations. Best results are bolded.

As expected, fixslicing appears as the most efficient implementation strategy for the encryption round function. While the byte-wise approach shows better performance for the tweakey schedule on 64-bit platforms, we decide to take the fixsliced implementation as a reference in terms of performance since many

operation modes leave some tweaky states unchanged across calls to Skinny-128. Another observation that stems from our benchmark is that the use of NEON on the Cortex-A7 outperforms the non-vectorized byte-wise implementation for the tweaky schedule only, resulting in similar performance for the encryption round function. This is likely due to the fact that, on the Cortex-A7, while most NEON instructions have a throughput of either 1 and 2 instructions per clock cycle when operating on doubleword and quadword registers, respectively, latencies are typically 4 cycles or more [28]. Therefore, directly using the result of the previous instruction will cause a stall. While the ARMv7-A NEON S-box implementation tries to mitigate such additional costs by carefully scheduling instructions, its sequential aspect makes it impossible to completely avoid it. The performance bottleneck of the encryption round function is clearly the 8-bit S-box, which is responsible for about 60% of the clock cycles in the fixsliced setting, versus 80% in the byte-wise setting. Therefore, optimizing this operation would significantly enhance the overall Skinny-128 performance.

### 3 Optimizing the S-box layer

#### 3.1 NEON vector permute instructions

NEON instruction set features a vector permute instruction named `tbl` which performs a table lookup at byte level. As originally introduced in ARMv7-A, it operates on doubleword registers providing a 64-bit output at a time. The table can be specified from 1 to 4 double-word registers, allowing up to 32 bytes. While the `tbl` instruction insert zeroes for out-of-bounds indices, its sister instruction `tbx` leaves the destination unchanged instead. In ARMv8-A, these instructions are operating on 128-bit wide registers allowing to specify a table of up to 64 bytes. Therefore, a single instruction is enough to compute an S-box up to either 5-bit or 6-bit on ARMv7-A and ARMv8-A, respectively. It is also possible to go further by combining several `tbl/tbx` calls. For instance, on ARMv8-A, given an 8-bit S-box one can split it into four 6-bit S-boxes: the first one covering bytes from 0 to 63, the second one covering bytes from 64 to 127, etc. First, a `tbl` instruction with the first 6-bit S-box is performed. If a byte is out-of-bounds the result is set to 0, or the final S-box output otherwise. Then, 64 is subtracted to each byte before applying the second 6-bit S-box using the `tbx` instruction (so that non-zero bytes calculated in the previous instruction are not affected). The same reasoning applies for the two remaining 6-bit S-boxes as detailed in Listing 1.1. This technique was actually applied to the AES S-box in order to boost its performance for ARMv8-A processors that do not include the optional Cryptography Extension [11,15]. Although the same trick applies to ARMv7-A as well, the limited number of registers coupled with the fact that permute instructions operate on 64-bit doublewords makes it inefficient for an 8-bit S-box, as it would occupy all the 32 NEON registers available. It would be still possible to store it in memory and perform loads during the calculation, but it would have a significant impact on performance. Another drawback of this technique is its inefficiency in some processors. It has been observed that `tbl/tbx`

performance can greatly vary from one platform to another [4]. For instance, in AArch64 mode, a `tbl` instruction with 4 input registers has a latency of 15 cycles on the A72 compared to only 5 on the A53, as summarized in Table 2. Those latency issues can be mitigated by executing several instances in parallel. However, because the internal state fits in a single 128-bit register, it is only of interest for parallelizable modes of operation. Another potential solution would be to use a clever decomposition of the S-box rather than simply splitting it into several parts.

```

1  tbl v1.16b, {v16.16b - v19.16b}, v0.16b // S-box for bytes in [0,63]
2  sub v0.16b, v0.16b, v15.16b           // Subtracts 64 to each byte
3  tbx v1.16b, {v20.16b - v23.16b}, v0.16b // S-box for bytes in [64,127]
4  sub v0.16b, v0.16b, v15.16b           // Subtracts 64 to each byte
5  tbx v1.16b, {v24.16b - v27.16b}, v0.16b // S-box for bytes in [128,191]
6  sub v0.16b, v0.16b, v15.16b           // Subtracts 64 to each byte
7  tbx v1.16b, {v28.16b - v31.16b}, v0.16b // S-box for bytes in [192,255]

```

Listing 1.1: ARMv8-A NEON implementation of an 8-bit S-box stored in `v16-v31`. The input register is `v0` and while the output register is `v1`. `v15` is supposed to contain `0x40...40`.

Execution Mode	Instructions	Throughput (ops/cycle)			Latency (cycles)		
		A7	A53	A72	A7	A53	A72
AArch32	<code>vtbl/vtbx</code> from 1/2 sources (64-bit wide)	1	1	2	4	2	3
	<code>vtbl/vtbx</code> from 3/4 sources (64-bit wide)	1/2	1/2	1	5	3	6
AArch64	<code>tbl/tbx</code> from $n$ sources (64-bit wide)	-	$1/n$	$2/n$	-	$n+1$	$3n$
	<code>tbl/tbx</code> from $n$ sources (128-bit wide)	-	$1/n$	$1/(2n-1)$	-	$n+1$	$3n+3$

Table 2: Effective execution latency and throughput for Neon vector permute instructions.

### 3.2 S-box decomposition

The decomposition of an S-box into smaller ones is a well-known technique to achieve compact hardware implementations. Building large S-boxes from smaller ones is actually a design strategy that has been used in many ciphers (e.g. Whirlpool [30], CLEFIA [29], Streebog and Kuznyechik [25]). This is also useful for side-channel countermeasures such as threshold implementations, where having a decomposition into functions with lower algebraic degrees allows to reach a secure implementation with fewer shares. Note that improvements for first-order

threshold implementations of Skinny-128 have been recently proposed thanks to novel S-box decompositions [12]. Such decompositions are also of interest in software, as it allows to build S-boxes that combine strong cryptanalytic properties and efficient bitsliced implementations (see e.g. *Scream* [17], *Robin* and *Fantomas* [16]). More closely related to our case study, a decomposition of the AES S-box has been proposed to achieve a constant-time implementation on Intel SIMD architectures [18]. It consists in representing  $\mathbb{F}_{2^8}$  as a degree-2 field extension of  $\mathbb{F}_{2^4}$  which allows computation of the AES S-box using small look-up tables that fit in `pshufb` instructions.

In the case of Skinny-128, we aim at finding an S-box decomposition that minimizes the number of inner S-boxes as well as their input size. Limiting the number of inner S-boxes will reduce the number of vector permute instructions while limiting their input size will reduce the number of input registers for these instructions (which has an impact on latency). Note that their output size can be anything between 1 and 8 bits since vector permute instructions operate at byte level. Another criterion to take into account is the way the input bits are positioned within bytes. Indeed for vector permute instructions, we want to be able to extract these input bits easily in order to store them in a contiguous manner. The ideal instruction to do so would be an SIMD equivalent of Intel `pext` which, for each byte, would apply a bitmask and pack the selected bits (either contiguous or non-contiguous) into contiguous low-order bit positions, as illustrated in Figure 4.

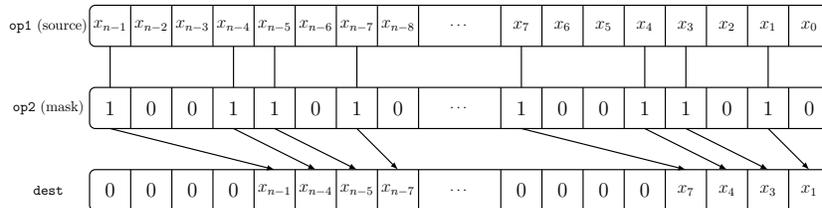


Fig. 4: Byte-wise SIMD parallel bit extract instruction. Each cell refers to a bit.

Unfortunately, there is no such instruction on ARM and replicating it in pure software is non-trivial as it would require many bit manipulations. To avoid such additional costs, we considered decompositions such that input bits of the inner S-boxes are always stored contiguously so that they can be extracted using a single bitmask or bitshift. We naturally started our investigations by looking at what can be done when applying a 4-bit S-box to each nibble individually. As highlighted in Figure 5, it appears that a single output term, namely  $y_6$ , exclusively depends on the most significant nibble while two output terms, namely  $y_5$  and  $y_3$ , exclusively depend on the less significant nibble. It implies that additional inner S-boxes are necessary to compute the remaining five terms, with their inputs consisting of output terms from both previous 4-bit S-boxes. Because

those S-box outputs will be stored in two distinct registers, we will inevitably spend some cycles to end up with output terms from both S-boxes in the same register. To mix up these output bits, we suggest taking advantage of the fact that the output size of inner S-boxes is up to 8 bits. Therefore, without additional cost, we can do some bit rearrangement with the next inner S-boxes in mind before merging both outputs in the same register using a bitwise XOR. This way we can simply extract the input bits using a single bitmask or bitshift as done previously. Note that a clever merge of both outputs using a bitwise XOR also allows computing some logic gates for free. This is typically the case for the XOR required to compute the output term  $y_2 = \neg(x_2 \vee x_1) \oplus x_6$ , which means that after the merge, we now have four (instead of three) output terms among eight:  $y_2, y_3, y_5, y_6$  as highlighted in Figure 5. As a result, we are not simply interested in pure table decompositions, but rather in decompositions with potential additional bitwise operations. In order to investigate what would be the best strategy for the remaining four terms, we give a more formal definition of the 8-bit S-box in Equation (1).

$$S: \{0, 1\}^8 \rightarrow \{0, 1\}^8$$

$$\begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} \mapsto \begin{pmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{pmatrix} = \begin{pmatrix} (x_7 \vee x_6 \oplus x_4) \wedge (x_3 \vee x_2 \oplus x_0) \oplus x_5 \\ \neg(x_7 \vee x_6) \oplus x_4 \\ \neg(x_3 \vee x_2) \oplus x_0 \\ \neg((\neg(y_6 \vee y_5) \oplus x_5) \vee y_6) \oplus x_3 \\ \neg(y_5 \vee x_3) \oplus x_1 \\ \neg(x_2 \vee x_1) \oplus x_6 \\ \neg(y_7 \vee y_2) \oplus x_7 \\ \neg(y_1 \vee y_3) \oplus x_2 \end{pmatrix} \quad (1)$$

In all logic, the four output terms that require only two inner S-boxes are defined by the component functions with the lowest algebraic degree. Among the four remaining terms,  $y_7$  and  $y_4$  are of degree 4,  $y_1$  is of degree 5 and  $y_0$  is of degree 6. Without considering  $y_0$ , which is the output term of the highest degree, another single inner S-box with four input bits would be sufficient. Indeed, one can see that  $y_7, y_4$  and  $y_1$  can be partially computed from  $y_6, y_5, y_2$  and  $x_5$  which are all available after the first layer of inner S-boxes (including the merge step). Note that we would actually need  $x_7$  and  $x_3$  as well for additional bitwise XORs, but we assume that they can be included in the computation for free with a clever merge as detailed above. However,  $y_0$  makes things more complicated as it requires to consider two additional terms:  $x_7$  and  $y_3$ . Therefore, we could theoretically get the remaining four output terms with a single 6-bit S-box call, but this option is only worth consideration for the ARMv8-A architecture since ARMv7-A `vtbl` and Intel SSSE3 `pshufb` instructions do not support such sizes. Instead, we suggest to slightly decompose the last output term as detailed in

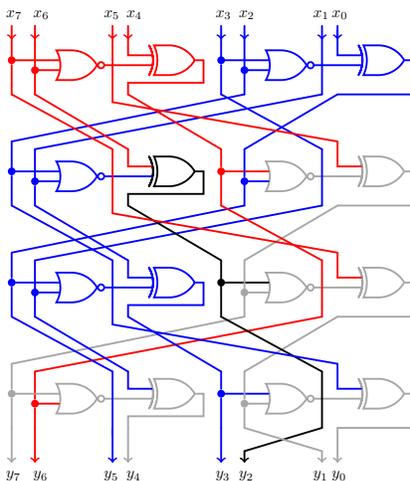


Fig. 5: The Skinny-128 S-box. Two inner S-boxes with 4-bit input (respectively highlighted in red and blue) are sufficient to get four output terms (including XOR when merging outputs, highlighted in black).

Equation (2) so that each operand of the bitwise XOR can be computed separately and then merged together. It still requires to consider  $y_3$  as additional input, resulting in a 5-bit S-box.

$$\begin{aligned}
 y_0 &= \neg(y_1 \vee y_3) \oplus x_2 \\
 &= \neg(\neg(\neg(y_7 \vee y_2) \oplus x_7) \vee y_3) \oplus x_2 \\
 &= (y_7 \vee y_2 \oplus x_7) \wedge \neg y_3 \oplus x_2 \\
 &= [(y_7 \vee y_2) \wedge \neg y_3] \oplus [x_7 \wedge \neg y_3 \oplus x_2]
 \end{aligned} \tag{2}$$

In order to achieve a version with 4-bit inner S-boxes only, one can use an additional bitwise AND operation at the cost of latency cycles. Both approaches, namely  $D_{4444}$  and  $D_{4454}$ , are formally defined in Appendixes A and B, respectively. Note that those decompositions are not the only possible ones, and other similar solutions surely exist. Still, given the restriction on the contiguous storage of the input bits, we believe that it is not possible to reach less than four inner S-box calls when limiting the number of input bits to 4 or 5. As shown in Table 3, our decompositions make the S-box layer faster on all processors, except on the A72 where they reach the same performance as the fixsliced version. According to the performance, it seems that one should favor  $D_{4454}$  over  $D_{4444}$ . However, the fact that the 5-bit inner S-box  $S_2$  requires an additional 128-bit register is troublesome on the ARMv7-A architecture. We suggest to keep this register free on ARMv7-A as it allows to avoid stack usage during the entire Skinny-128 encryption as detailed in Section 4.2.

Implementation	Ref	Speed (clock cycles)		
		A7	A53	A72
Fixsliced	[2]	40	32	33
Byte-wise	[31]	169	62	70
ARMv7-A NEON	[14]	163	-	-
ARMv8-A <code>tbl/tbx</code> split	[11]	-	26	64
$D_{4444}$	Ours	34	14	33
$D_{4454}$	Ours	30	13	33

Table 3: Performance comparison of various software implementations of the Skinny-128 S-box.

## 4 Other optimizations

### 4.1 Linear layer

The linear layer consists of the `ShiftRows` followed by the `MixColumns`. The main complication with our representation, i.e. the 128-bit internal state stored row-wise in a 128-bit register, is that XORs within the `MixColumns` are performed row-wise. In order to avoid additional bitmasks and bitshifts to add the correspondig rows together, our implementations take again advantage of vector permute instructions by expressing the `MixColumns` as the XOR of three operands as detailed in Equation (3).

$$\text{MixColumns} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix} = \begin{pmatrix} r_0 \oplus r_2 \oplus r_3 \\ r_0 \\ r_1 \oplus r_2 \\ r_0 \oplus r_2 \end{pmatrix} = \begin{pmatrix} r_3 \\ r_0 \\ r_1 \\ r_2 \end{pmatrix} \oplus \begin{pmatrix} r_2 \\ 0 \\ r_2 \\ r_0 \end{pmatrix} \oplus \begin{pmatrix} r_0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (3)$$

Since the operands are just different rows reordering of the internal state, they can be easily computed using a table lookup instruction. We also include the `ShiftRows` calculation within those instructions as it comes at no cost. Note that because the third operand only consists of the first row, whose bytes are not shifted by the `ShiftRows`, we simply perform a bitwise AND instead of a table lookup instruction. Therefore, the entire linear layer can be computed using 2 128-bit wide vector permute instructions, 1 bitwise AND, and 2 XORs. This translates to 6 and 5 instructions on ARMv7-A and ARMv8-A architectures, respectively.

## 4.2 Tweakey schedule

When it comes to the tweakey expansion, two options are left to the implementer: precalculation versus on-the-fly computation. It usually refers to a time-memory trade-off as on-the-fly computations allow to reduce memory usage at the cost of additional operations, and vice-versa. In our case, we decided to consider both approaches depending on the target platform. For instance, as highlighted in Table 3, our implementations suffer from many stall cycles on the A7 and A72 processors due to the latency of vector permute instructions on those platforms. Therefore, we suggest to take advantage of these stall cycles in order to compute the round tweakeys on-the-fly, in the background. We naturally implement the byte permutation using vector permute instructions while using bitwise operations for the LFSRs. On the ARMv8-A architecture, we compute double updates at once as illustrated in Figure 6 so that we can divide by a factor of two the number of instructions. Thanks to the large number of registers available in the

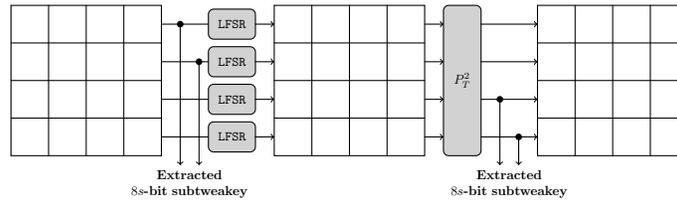


Fig. 6: Tweakey state double update

NEON SIMD unit, we can fit all the working variables in the NEON bank register without additional usage of the stack, even for Skinny-128-384 which requires three 128-bit tweakey states. This is what motivates us to use the  $D_{4444}$  decomposition on ARMv7-A since it requires two fewer 64-bit registers than  $D_{4454}$  (which would imply additional loads/stores on the stack).

## 5 Implementation results

### 5.1 ARM NEON

Table 4 reports benchmark results on the selected ARM NEON processors for Skinny-128-384+ and Romulus using the SUPERCOP benchmarking suite [9]. Skinny-128-384+ is a round-reduced version of Skinny-128-384 (decreased from 56 to 40) used in Romulus in order to enhance the performance while preserving a high security margin<sup>5</sup>. The latest specification of Romulus includes a nonce-based AE (Romulus-N), a nonce-misuse-resistant AE [26] (Romulus-M), a leakage-resilient AE (Romulus-T) based on TEDT [10], and a hash function (Romulus-H)

<sup>5</sup> [https://groups.google.com/a/list.nist.gov/g/lwc-forum/c/5\\_mqi9irD0U](https://groups.google.com/a/list.nist.gov/g/lwc-forum/c/5_mqi9irD0U)

based on MDPH [24]. We implemented and benchmarked all these members in order to compare our work with the fixsliced approach, which defines the most efficient software implementation available on this platform as highlighted in Section 2. For the sake of completeness, we considered both alternatives with precalculated and on-the-fly calculated round tweakeys. On top of running faster, a clear advantage of our NEON implementations over previous work is the RAM usage, which is smaller by a factor of 4 when computing the tweakey schedule on the fly. As expected, taking advantage of stall cycles on the A7 and A72 to compute the tweakey schedule in the background allows to reach the best performance on those platforms. Still, precomputing the round tweakeys is the most efficient approach on the A53. This is not only because there are fewer cycles latency for vector permute instructions on this platform: in each variant of Romulus, the tweakey is only differing from a byte or so across many calls to Skinny-128-384+ (see the Romulus specification for more details [19]). Therefore, it is possible to run some precalculations since the tweakeys  $TK2$  and  $TK3$  remain fixed. In the end, the improvement factor of our NEON implementations over fixslicing differs significantly depending on the processor. For Skinny-128-384+ (including the tweakey schedule) and Romulus, it is roughly 1.5, 2, and 3.5 on the A72, A7, and A53, respectively.

## 5.2 Intel Streaming SIMD Extensions

We also ported our implementations on Intel using Supplemental Streaming SIMD Extensions 3 (SSSE3) intrinsics. To do so, we naturally opted for the  $D_{4444}$  decomposition since the vector permute instruction `pshufb` operates on 16-byte vectors only. In order to save 1 instruction per S-box call, we simply reordered the output bits of  $S_0$  and  $S_1$  so that we can extract  $y_3$  using a single bitmask instead of a shift (there is no `_mm_srli_epi8` so we would need `_mm_srli_epi16` followed by `_mm_and_si128` to discard the bits from adjacent bytes). Because there are no high latency issues related to `pshufb` and because the amount of RAM usage is relatively small for such platforms, we only considered the variant with precalculation of the round tweakeys in order to reach the best performance. As reported in Table 5, the improvement in terms of performance ranges between 4 and 5 on Whiskey Lake and Comet Lake microarchitectures. Note that the advanced vector extension AVX2 should not lead to significant enhancements since its `vpshufb` instruction operates within 128-bit lanes. The only advantage is to either (1) process two blocks in parallel (which is only of interest for Romulus-H that rely on the MDPH mode) or (2) to use a sparse representation by storing bytes within 16-bit words in order to use a single `_mm_srli_epi16` instruction when extracting topmost nibbles for the S-box decomposition. Still, the `vpshufb` instruction in AVX512 allows to handle permutations across entire 512-bit registers, making possible to implement the  $D_{4454}$  decomposition on Intel as well.

Algorithm	Implementation	Speed (cycles/byte)			RAM
		A7	A53	A72	(bytes)
Skinny-128-384+					
encryption only	Fixsliced [2]	254	129	123	-
	Ours	<b>127</b>	<b>57</b>	<b>111</b>	-
encryption + tweakkey schedule	Fixsliced [2]	431	321	201	736
	Ours (precalculate)	177	<b>84</b>	148	368
	Ours (on-the-fly)	<b>143</b>	85	<b>112</b>	<b>16</b>
Romulus					
Romulus-N nonce-based AEAD	Fixsliced [2]	239	165	137	1 088
	Ours (precalculate)	112	<b>48</b>	94	544
	Ours (on-the-fly)	<b>110</b>	64	<b>85</b>	<b>240</b>
Romulus-M nonce misuse-resistant AEAD	Fixsliced [2]	337	245	199	1 136
	Ours (precalculate)	153	<b>69</b>	130	640
	Ours (on-the-fly)	<b>144</b>	83	<b>113</b>	<b>272</b>
Romulus-T leakage-resilient AEAD	Fixsliced [2]	705	551	387	1 136
	Ours (precalculate)	321	<b>145</b>	273	640
	Ours (on-the-fly)	<b>289</b>	158	<b>226</b>	<b>272</b>
Romulus-H hash function	Fixsliced [2]	318	227	187	1 104
	Ours (precalculate)	161	<b>71</b>	138	544
	Ours (on-the-fly)	<b>150</b>	85	<b>116</b>	<b>224</b>

Table 4: Benchmark on ARM Cortex-A processors. Results are given when processing 4096 bytes for Romulus (2048-byte additional data and 2048-byte message) and a single block (i.e. 16 bytes) for Skinny-128-384+. The function ‘encryption only’ takes as input the round tweakkeys fully precomputed while ‘encryption + tweakkey schedule’ simply requires the 48-byte tweakkey.

Algorithm	Implementation	Speed (cycles/byte)	
		i5-8365U (Whiskey Lake)	i7-10510U (Comet Lake)
Skinny-128-384+			
encryption only	Fixsliced [2]	150	165
	Ours	<b>44</b>	<b>47</b>
encryption + tweakkey schedule	Fixsliced [2]	282	305
	Ours	<b>58</b>	<b>62</b>
Romulus			
Romulus-N nonce-based AEAD	Fixsliced [2]	161	175
	Ours	<b>37</b>	<b>40</b>
Romulus-M nonce misuse-resistant AEAD	Fixsliced [2]	234	252
	Ours	<b>51</b>	<b>55</b>
Romulus-T leakage-resilient AEAD	Fixsliced [2]	453	491
	Ours	<b>109</b>	<b>118</b>
Romulus-H hash function	Fixsliced [2]	220	238
	Ours	<b>54</b>	<b>58</b>

Table 5: Benchmark on Intel processors. Results are given when processing 4096 bytes for Romulus (2048-byte additional data and 2048-byte message) and a single block (i.e. 16 bytes) for Skinny-128-384+. The function ‘encryption only’ takes as input the round tweakkeys fully precomputed while ‘encryption + tweakkey schedule’ simply requires the 48-byte tweakkey. Benchmarks were run by carefully disabling the TurboBoost technology.

## 6 Conclusion and future work

We introduced SIMD implementations of Skinny-128 whose performance outperform previous work by up to a factor of 4 on various platforms. The main optimization consists in decomposing the 8-bit S-box in smaller S-boxes with 4/5-bit inputs in order to take advantage of vector permute instructions. It is very likely that other S-boxes in the literature may benefit from a similar implementation technique, and developing a generic tool that would list the relevant decompositions regarding vector permute instructions could be useful for other designs. More generally, we believe that the design of large S-boxes with efficient decompositions could provide attractive trade-offs between security and performance on SIMD platforms. Our work also highlights that performance can greatly vary from a microarchitecture to another, due to possible design discrepancies regarding vector permute instructions. Finally, we did not discuss the integration of countermeasures against power side-channel attacks but studying the relevance of lookup table masking schemes combined with vector permute instructions might be an interesting direction for future research.

**Acknowledgements.** We are grateful to Thomas Peyrin as well as the anonymous reviewers for their comments that improved the quality of this article.

## References

1. Adomnicaï, A., Najm, Z., Peyrin, T.: Fixslicing: A new GIFT representation fast constant-time implementations of GIFT and GIFT-COFB on ARM cortex-m. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**(3), 402–427 (2020)
2. Adomnicaï, A., Peyrin, T.: Fixslicing AES-like Ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2021**(1), 402–425 (Dec 2020), <https://tches.iacr.org/index.php/TCHES/article/view/8739>
3. Andreeva, E., Lallemand, V., Purnal, A., Reyhanitabar, R., Roy, A., Vizár, D.: ForkAE v.1. Submission to the NIST Lightweight Cryptography Project (2019)
4. Aufranc, J.L.: How ARM Nerfed NEON Permute Instructions in ARMv8. <https://www.cnx-software.com/2017/08/07/how-arm-nerfed-neon-permute-instructions-in-armv8> (2017), accessed: 2021-11-25
5. Banik, S., Bao, Z., Isobe, T., Kubo, H., Liu, F., Minematsu, K., Sakamoto, K., Shibata, N., Shigeri, M.: WARP : Revisiting GFN for Lightweight 128-Bit Block Cipher. In: Dunkelman, O., Jacobson, Jr., M.J., O’Flynn, C. (eds.) *Selected Areas in Cryptography*. pp. 535–564. Springer International Publishing, Cham (2021)
6. Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption. In: CHES. *Lecture Notes in Computer Science*, vol. 10529, pp. 321–345. Springer (2017)
7. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In: CRYPTO (2). *Lecture Notes in Computer Science*, vol. 9815, pp. 123–153. Springer (2016)
8. Benadjila, R., Guo, J., Lomné, V., Peyrin, T.: Implementing Lightweight Block Ciphers on x86 Architectures. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) *Selected Areas in Cryptography – SAC 2013*. pp. 324–351. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
9. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. <https://bench.cr.yp.to>, accessed: 2022-02-25
10. Berti, F., Guo, C., Pereira, O., Peters, T., Standaert, F.: TEDT, a Leakage-Resist AEAD Mode for High Physical Security Applications. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**(1), 256–320 (2020)
11. Biesheuvel, A.: Accelerated AES for the Arm64 Linux kernel. <https://www.linaro.org/blog/accelerated-aes-for-the-arm64-linux-kernel/> (2017), accessed: 2021-10-25
12. Caforio, A., Collins, D., Glamocanin, O., Banik, S.: Improving First-Order Threshold Implementations of SKINNY. *Cryptology ePrint Archive*, Report 2021/1425 (2021), <https://ia.cr/2021/1425>
13. Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-Based Authenticated Encryption: How Small Can We Go? In: CHES. *Lecture Notes in Computer Science*, vol. 10529, pp. 277–298. Springer (2017)
14. Deprez, A., Andreeva, E., Mera, J.M.B., Karmakar, A., Purnal, A.: Optimized Software Implementations for the Lightweight Encryption Scheme ForkAE. In:

- Liardet, P.Y., Mentens, N. (eds.) Smart Card Research and Advanced Applications. pp. 68–83. Springer International Publishing, Cham (2021)
15. Fujii, H., Rodrigues, F.C., López, J.: Fast AES Implementation Using ARMv8 ASIMD Without Cryptography Extension. In: Seo, J.H. (ed.) Information Security and Cryptology – ICISC 2019. pp. 84–101. Springer International Publishing, Cham (2020)
  16. Grosso, V., Leurent, G., Standaert, F.X., Varıcı, K.: LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations. In: Cid, C., Rechberger, C. (eds.) Fast Software Encryption. pp. 18–37. Springer Berlin Heidelberg, Berlin, Heidelberg (2015), <https://hal.inria.fr/hal-01093491/document>
  17. Grosso, V., Varıcı, A.K., Gaspar, L.: Scream - side-channel resistant authenticated encryption with masking (2015), <https://competitions.cr.yt.to/round2/screamv3.pdf>
  18. Hamburg, M.: Accelerating AES with Vector Permute Instructions. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009. pp. 18–32. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
  19. Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: Duel of the Titans: The Romulus and Remus Families of Lightweight AEAD Algorithms. IACR Transactions on Symmetric Cryptology **2020**(1), 43–120 (May 2020), <https://tosc.iacr.org/index.php/ToSC/article/view/8560>
  20. Jean, J.: TikZ for Cryptographers. <https://www.iacr.org/authors/tikz/> (2016)
  21. Jean, J., Nikolic, I., Peyrin, T.: Tweaks and Keys for Block Ciphers: The TWEAKEKEY Framework. In: ASIACRYPT (2014)
  22. Mauro, A.D., Fatemi, H., de Gyvez, J.P., Benini, L.: Idleness-Aware Dynamic Power Mode Selection on the i.MX 7ULP IoT Edge Processor. Journal of Low Power Electronics and Applications **10**(2) (2020), <https://www.mdpi.com/2079-9268/10/2/19>
  23. McKay, K., Bassham, L., Turan, M.S., Mouha, N.: Report on Lightweight Cryptography (2017-03-28 00:03:00 2017), [https://tsapps.nist.gov/publication/get\\_pdf.cfm?pub\\_id=922743](https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=922743)
  24. Naito, Y.: Optimally Indifferentiable Double-Block-Length Hashing Without Post-processing and with Support for Longer Key Than Single Block. In: LATIN-CRYPT. Lecture Notes in Computer Science, vol. 11774, pp. 65–85. Springer (2019)
  25. Perrin, L.: Partitions in the S-Box of Streebog and Kuznyechik. IACR Transactions on Symmetric Cryptology **2019**(1), 302–329 (Mar 2019), <https://tosc.iacr.org/index.php/ToSC/article/view/7405>
  26. Rogaway, P., Shrimpton, T.: A Provable-Security Treatment of the Key-Wrap Problem. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 4004, pp. 373–390. Springer (2006)
  27. Ronen, E., Shamir, A., Weingarten, A.O., O’Flynn, C.: IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 195–212 (2017)
  28. Rullgard, M.: Cortex-A7 instruction cycle timings. <https://hardwarebug.org/2014/05/15/cortex-a7-instruction-cycle-timings> (2014), accessed: 2021-10-25
  29. Shirai, T., Shibutani, K., Akishita, T., Moriai, S., Iwata, T.: The 128-Bit Blockcipher CLEFIA. In: Proceedings of the 14th International Conference on Fast Software Encryption. p. 181–195. FSE’07, Springer-Verlag, Berlin, Heidelberg (2007)
  30. S.L.M, P., Rijmen, V.: The Whirlpool Hashing Function (2003)
  31. Weatherley, R.: SKINNY tweakable block cipher. <https://github.com/rweather/skinny-c> (2017)

## Appendix

This appendix formally defines the S-box decompositions  $D_{4444}$  and  $D_{4454}$  introduced in Section 3.

### A $D_{4444}$ decomposition

$$S_0 \begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ x_7 \\ x_5 \\ \neg(x_7 \vee x_6) \oplus x_4 \\ 0 \\ x_6 \end{pmatrix} \quad S_1 \begin{pmatrix} x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} = \begin{pmatrix} \neg((\neg(x_3 \vee x_2) \oplus x_0) \vee x_3) \oplus x_1 \\ x_3 \\ x_2 \\ 0 \\ 0 \\ 0 \\ \neg(x_3 \vee x_2) \oplus x_0 \\ \neg(x_2 \vee x_1) \end{pmatrix}$$

$$S_2 \begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ x_6 \\ x_7 \\ 0 \\ x_4 \\ (x_7 \vee \neg x_4) \oplus x_5 \end{pmatrix} \quad S_3 \begin{pmatrix} x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} = \begin{pmatrix} \neg(x_2 \vee x_1) \oplus x_3 \\ x_2 \\ x_1 \\ \neg((\neg(x_2 \vee x_1) \oplus x_3) \vee x_2) \\ 0 \\ x_0 \\ \neg((\neg(x_2 \vee x_1) \oplus x_3) \vee x_0) \\ \neg((\neg(x_2 \vee x_1) \oplus x_3) \vee x_0) \end{pmatrix}$$

$$S_0 \begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \end{pmatrix} \oplus S_1 \begin{pmatrix} x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} = \begin{pmatrix} y_3 \\ x_3 \\ x_2 \\ x_7 \\ x_5 \\ y_6 \\ y_5 \\ y_2 \end{pmatrix} \quad S_2 \begin{pmatrix} y_3 \\ x_3 \\ x_2 \\ x_7 \end{pmatrix} \oplus S_3 \begin{pmatrix} x_5 \\ y_6 \\ y_5 \\ y_2 \end{pmatrix} \vee \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{pmatrix}$$

**B**  $D_{4454}$  decomposition

$$S_0 \begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_5 \\ \neg(x_7 \vee x_6) \oplus x_4 \\ 0 \\ x_6 \\ 0 \\ 0 \\ 0 \\ x_7 \end{pmatrix} \quad S_1 \begin{pmatrix} x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \neg(x_3 \vee x_2) \oplus x_0 \\ \neg(x_2 \vee x_1) \\ \neg((\neg(x_3 \vee x_2) \oplus x_0) \vee x_3) \oplus x_1 \\ x_3 \\ x_2 \\ 0 \end{pmatrix}$$

$$S_2 \begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \end{pmatrix} = \begin{pmatrix} \neg(x_6 \vee x_5) \oplus x_7 \\ x_6 \\ x_5 \\ \neg((\neg(x_6 \vee x_5) \oplus x_7) \vee x_6) \\ 0 \\ x_4 \\ \neg((\neg(x_6 \vee x_5) \oplus x_7) \vee x_4) \\ \neg((\neg(x_6 \vee x_5) \oplus x_7) \vee x_4) \vee x_3 \end{pmatrix} \quad S_3 \begin{pmatrix} x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ x_2 \\ x_3 \\ 0 \\ x_0 \\ (x_3 \vee \neg x_0) \oplus x_1 \end{pmatrix}$$

$$S_0 \begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \end{pmatrix} \oplus S_1 \begin{pmatrix} x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} = \begin{pmatrix} x_5 \\ y_6 \\ y_5 \\ y_2 \\ y_3 \\ x_3 \\ x_2 \\ x_7 \end{pmatrix} \quad S_2 \begin{pmatrix} x_5 \\ y_6 \\ y_5 \\ y_2 \\ y_3 \end{pmatrix} \oplus S_3 \begin{pmatrix} y_3 \\ x_3 \\ x_2 \\ x_7 \end{pmatrix} = \begin{pmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{pmatrix}$$