# Fixslicing - Application to Some NIST LWC Round 2 Candidates

Alexandre Adomnicai and Thomas Peyrin

Temasek Laboratories, Singapore
Nanyang Technological University, Singapore
firstname.lastname@ntu.edu.sg

**Abstract.** This work briefly recaps the benefits of the fixslicing implementation strategy when applied to AES-128, GIFT-128 and Skinny-128 and shows how it impacts the software performance of some selected NIST LWC round 2 candidates built upon those internal primitives, namely GIFT-COFB, Romulus, Skinny-AEAD and SAEAES. Benchmark results for constant-time implementations on ARM Cortex-M3 are reported for payloads up to 256 bytes.

**Keywords:** NIST LWC · Fixslicing · AES · GIFT · Skinny

## 1 Introduction

The NIST LightWeight Cryptography (NIST LWC) competition started in 2018 with the goal of selecting the future authenticated encryption standard(s) for resource-constrained environments. Some candidates are more hardware-oriented and therefore less efficient in software, and vice versa. However, the border between the two categories may be thinner than it seems. For instance, the fixslicing implementation strategy [ANP20] was originally introduced as a new representation for the hardware-oriented GIFT block cipher [BPP+17] to achieve very efficient software constant-time implementations. As a result, the round 2 candidate GIFT-COFB [BCI+20] ranks among the fastest candidates on microcontrollers [Wea20,RPM20]. It was recently highlighted in [AP20] that the main idea behind the fixsliced GIFT representation is actually generic by providing new bitsliced speed records for AES and Skinny-128 on 32-bit architectures.

This work aims at giving insights into the benefits of fixslicing in the context of the NIST LWC standardization process by reporting fixsliced implementation results on ARM Cortex-M based microcontrollers for some selected round 2 candidates built upon the AES-128, GIFT-128 and Skinny-128 internal primitives.

## 2 The fixslicing implementation strategy

### 2.1 Overview

Fixslicing is a specific instance of bitslicing, a generic implementation strategy to achieve software constant-time implementations, where at least one of the slices remains fixed. A fixed $n$-slice consists of $n$ bits that remain at the same position through the entire algorithm execution (except when applying some corrections to resynchronize with the classical representation). The goal of fixslicing is to take advantage of an alternative representation for a few rounds in order to make the linear layer less costly to implement. This technique is especially of interest for Substitution-bitPermutation Network (SbPN) designs where the linear layer is basically free in hardware (it consists of simple wirings) but usually expensive in software since the bits have to be moved around using many bitmasks, shifts and bitwise ORs. Fixslicing was originally introduced as a new GIFT representation allowing to boost its performance on 32-bit microcontrollers up to a factor of 7 when compared to classical bitsliced implementations [ANP20]. The bit permutations used in the GIFT block ciphers have the special property that, from a bitsliced perspective, all bits within a slice remains in the same one through the permutation. The authors showed that fixing one of the slices and adjusting the others so that the bits are correctly aligned for the S-box makes the linear layer software-friendly, hence the ferm 'fixslicing'. Note that a similar optimization has been previously applied to the PRESENT block cipher [BKL+07] by introducing an alternative representation of the cipher thanks to a decomposition of the permutation layer over two consecutive rounds [RAL17].

### 2.2 Application to AES-like ciphers

In its original publication, the fixslicing implementation strategy was mentioned to be generic for other SbPN designs with the special property that all bits within a slice remains in the same one through the permutation layer. However, a recent application to AES-like ciphers [AP20] demonstrated that fixslicing has a much wider scope of applications than initially thought. More precisely, the authors showed that an application of fixslicing to the AES allows to reduce the number of operations in the linear layer by 41% when compared to classical bitsliced implementations on 32-bit platforms. To put it in a nutshell, fixslicing AES-like ciphers is equivalent to omit the `ShiftRows` by fixing all the slices to never move and adjusting the `MixColumns` calculations accordingly. The authors also report fixsliced Skinny-128 implementations results on ARM Cortex-M3 and show that it outperforms prior results reported in the literature by a factor of 4. Especially, it is shown how to take advantage of some symmetry in the 8-bit S-box in order to implement it using 4 slices instead of 8. This allows to achieve fixsliced implementations that operate on a single block at a time, which is highly valuable for operating modes that do not provide parallelism. The Table 1 recaps the results previously reported in the literature for the internal primitives we will consider in the following section. Regarding Skinny-128-384, we also consider the new variant Skinny-128-384+ which corresponds to Skinny-128-384 reduced from 56 to 40 rounds. This variant was introduced to provide more attractive security margin/efficiency trade-offs as Skinny-128-384 guarantees a much larger security margin when compared to other candidates' internal primitives [Pey20]. Note that no results for the fixsliced Skinny-128 tweakey schedule have been previously reported in the literature. Since this building block will be needed to benchmark our selected Skinny-based candidates, we describe hereafter how we did implement it.

| Algorithm | Ref | Parallel | Speed (cycles) | | Code size | RAM (bytes) | |
|---|---|---|---|---|---|---|---|
| | | Blocks | M3 | M4 | (bytes) | In/Output | Stack |
| GIFT-128 key exp. | [ANP20] | 1 | 1 813 | 1 812 | 1 100 | 336 | 56 |
| GIFTb-128 encryption | [ANP20] | 1 | 1 297 | 1 279 | 994 | 16 (+320) | 64 |
| AES-128 key exp. | [AP20] | 2 | 4 135 | 4 173 | 962 | 368 | 112 |
| AES-128 encryption | [AP20] | 2 | 1 397 | 1 418 | 2 556 | 32 (+352) | 112 |
| Skinny-128-384 encryption | [AP20] | 1 | 4 223 | 4 238 | 1 536 | 16 (+896) | 60 |
| | | 2 | 2 566 | 2 579 | 1 636 | 32 (+1 792) | 60 |
| Skinny-128-384+ encryption | [AP20] | 1 | 3 055 | 3 066 | 1 504 | 16 (+640) | 60 |
| | | 2 | 1 862 | 1 872 | 1 620 | 32 (+1 280) | 60 |

Table 1: Constant-time implementation results on ARM Cortex-M3/4 for fixsliced implementations of AES-128, GIFT-128 and Skinny-128-384. For encryption routines, speed is expressed in cycles per block. In/Output refers to the amount of memory needed to store the input and ouput plus the temporary variables (including the round keys).

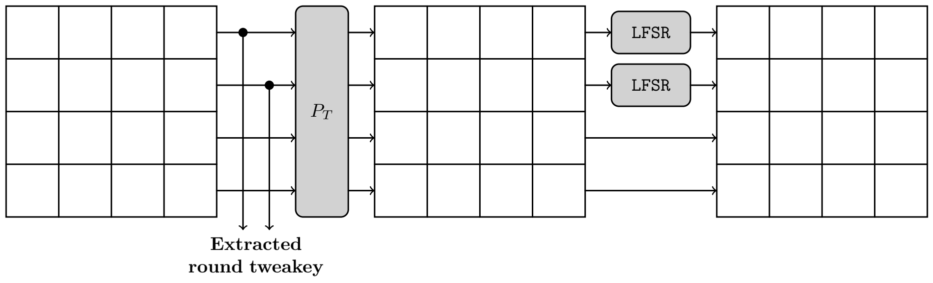
## 2.3 Efficient implementation of the Skinny tweakey schedule

Skinny follows the Tweakey framework [JNP14] and thus takes a tweakey input instead of a key or a pair key/tweak. The Skinny family of tweakable block ciphers has three main tweakey size versions: $t = n$, $t = 2n$ and $t = 3n$ where $n$ refers to the block size and $z = t/n$ refers to the tweakey size to block size ratio. The tweakey state is also viewed as a collection of $z$ $4 \times 4$ square arrays of bytes, denoted $TK1$ when $z = 1$, $TK1$ and $TK2$ when $z = 2$, and finally $TK1$, $TK2$, $TK3$ when $z = 3$. During the round tweakeys addition, the first and second rows of all tweakey arrays are extracted and bitwise exclusive-ORed to the cipher internal state, respecting the array positioning. Then, the tweakey arrays are updated as follows. First, a permutation $P_T$ is applied on the cell positions of all tweakey arrays.

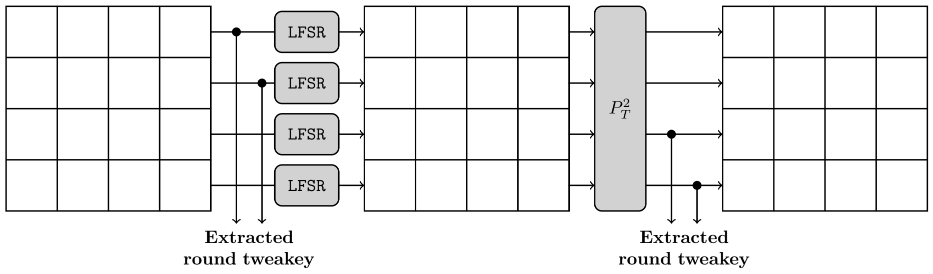
$$P_T = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7]$$

Then, every cell of the first and second rows of $TK2$ and $TK3$ (for the versions where $TK2$ and $TK3$ are used) are individually updated with an LFSR. The Figure 1a illustrates one round of the tweakey schedule.

As a matter of efficiency, we suggest to compute double rounds of the tweakey schedule instead of single ones. The main reason lies in the fact that, during a round, the first and second rows of all tweakey arrays are just moved to the third and fourth rows. When implementing the tweakey schedule in a bitsliced fashion, it is more efficient to compute the operations on the entire state. Considering double rounds is equivalent to applying the permutation and the LFSR to all rows, as depicted in Figure 1b.



(a) Single round (from [Jea16])



(b) Double round

Fig. 1: Single and double round implementations of the Skinny tweakey schedule

While the LFSRs are specific to each tweakey array, the permutation remains the same. Therefore, when considering several tweakey arrays (i.e. $z = 2$ or $z = 3$), computing the permutation after having extracted and exclusive-ORed all tweakeys together allows to speed up the computations. However, the drawback of this approach is that one has to apply the permutation to the round tweakeys as many times as it should have been done in the naive approach. Because $P_T^{16} = Id$, instead of calling $P_T^2$ as many times as required, we can compute $P_T^i$ for $i \in \{2, 4, \cdots, 14\}$ depending on the round number. The Figure 2 illustrates our efficient implementation of the tweakey schedule.

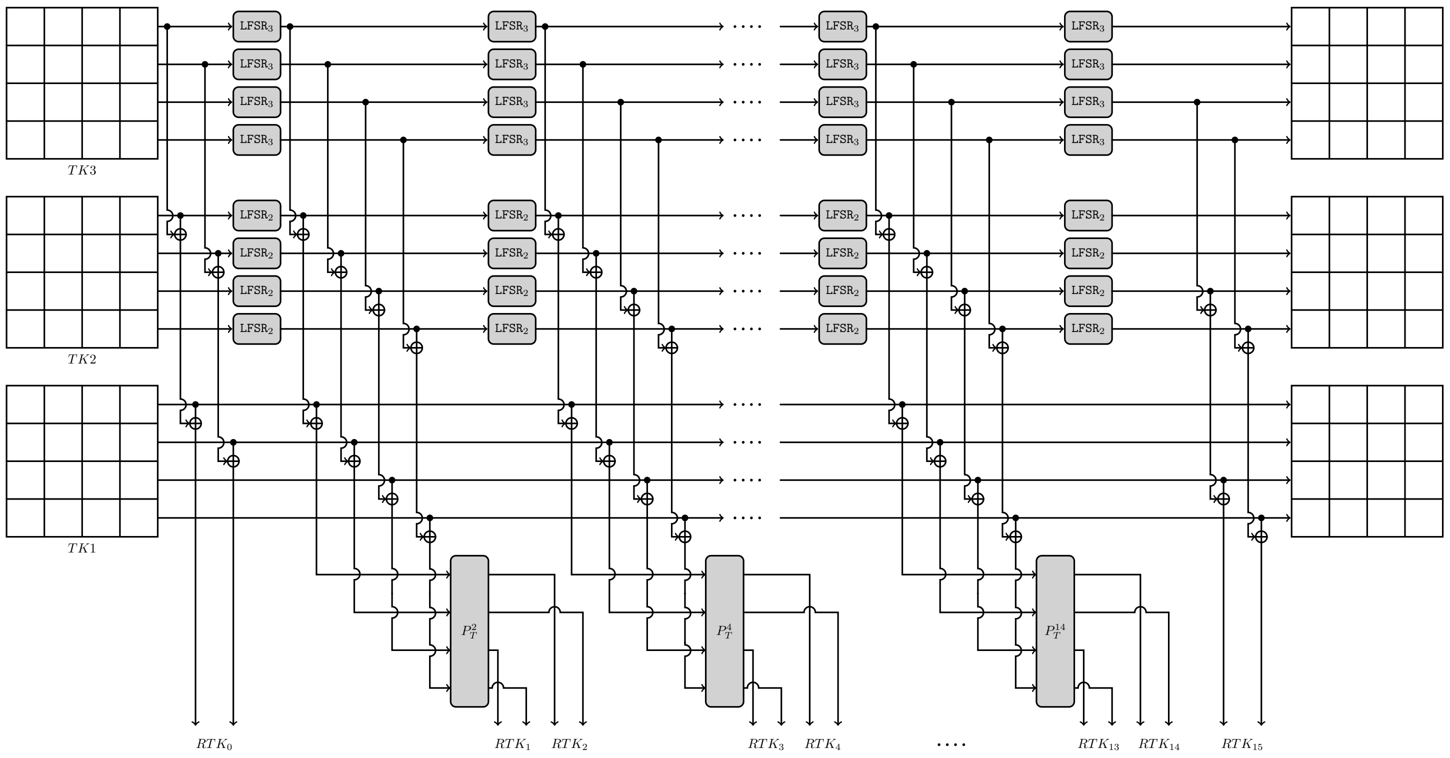


Fig. 2: Speed-optimized implementation of the Skinny tweakey schedule to derive 16 round tweakeys. Can be reiterated to derive more round tweakey material.

We opted for this approach to reach the best possible performance for the Skinny-based NIST LWC candidates we consider in our benchmark. Still, it has a considerable impact on code size as reported in Table 2. When those memory requirements are too high to be practical, we suggest to omit this implementation trick at the cost of a slower execution time. Our code is publicly available at `https://github.com/aadomn/skinny`.

| **Algorithm** | **Parallel Blocks** | **Speed** (cycles) M3 | **Speed** (cycles) M4 | **Code size** (bytes) | **RAM** (bytes) In/Output | **RAM** (bytes) Stack |
|---|---|---|---|---|---|---|
| Skinny-128-384 tweakey exp. | 1 | 3 252 | 3 266 | 5 754 | 896 | 60 |
| | 2 | 3 517 | 3 580 | 10 608 | 1 792 | 60 |
| Skinny-128-384+ tweakey exp. | 1 | 2 426 | 2 449 | 4 796 | 640 | 60 |
| | 2 | 2 588 | 2 605 | 8 620 | 1 280 | 60 |

Table 2: Implementation results on ARM Cortex-M for constant-time implementations of the Skinny-128-384 tweakey schedule. Speed refers to the entire tweakey expansion while the number of parallel blocks refer to the representation to match. In/Output refers to the amount of memory needed to store all the pre-computed round tweakeys.

## 3 Application to some NIST LWC round 2 candidates

### 3.1 GIFT-COFB

Implementation results for fixsliced GIFT-COFB have already been reported in [ANP20]. For our benchmark, we run measurements using the balanced implementation publicly available at `https://github.com/aadomn/gift`. This implementation aims at providing a trade-off between efficiency and compactness.

### 3.2 The Romulus family of lightweight AEAD algorithms

Romulus [IKMP20] consists of two families, a nonce-based and a nonce misuse-resistant authenticated encryption, namely Romulus-N and Romulus-M. In this document, we only consider the primary versions of each family, namely Romulus-N1 and Romulus-M1, as well as the new variants Romulus-N1+ and Romulus-M1+ based on Skinny-128-384+.

A feature shared between both families is that, when processing the input message, the tweakey is initialized as follows:

$$TK1 \leftarrow \texttt{lfsr}_{56}(D) \mid\mid B \mid\mid 0^{64}$$
$$TK2 \leftarrow \texttt{nonce}$$
$$TK3 \leftarrow \texttt{key}$$

where $\texttt{lfsr}_{56}(D)$ refers to a 56-bit counter and $B$ refers to a domain separation byte. Therefore, when multiple blocks of message have to be processed, the only part of the tweakey that will vary is $TK1$. In order to avoid useless computations, we suggest to precompute all the round tweakeys excluding $TK1$ at the beginning of the algorithm. Then, for each block, we increment $TK1$ before packing it in the fixsliced representation and running the tweakey schedule for this tweak only. Actually, since no LFSR is involved for this tweak and that $P_T^{16} = Id$, we can just compute the first 16 rounds. Moreover, since the last 64 bits of $TK1$ are always zero, we only need to store half of the outputs. Finally during the Skinny-128-384 execution we add the round tweakeys in two steps: first add the precomputed round tweakeys excluding $TK1$, and then add the round tweakey that exclusively results from $TK1$. Although it requires some additional exclusive-ORs and memory accesses during the AddRoundTweakey operation, it allows to save the recomputation of the entire tweakey schedule.

## 3.3 Skinny-AEAD

Skinny-128 also defines the underlying tweakable block ciphers used in Skinny-AEAD [BJK+20], another submission to the NIST LWC competition. In this document, we only consider the primary version Skinny-AEAD-M1 and the corresponding new variant Skinny-AEAD-M1+ based on Skinny-128-384+. As for Romulus, $TK1$ is the only tweakey part that varies through the entire algorithm execution since it is initialized as follows:

$$\begin{aligned} TK1 &\leftarrow \texttt{lfsr}_{64}(D) \,||\, 0^{56} \,||\, B \\ TK2 &\leftarrow \texttt{nonce} \\ TK3 &\leftarrow \texttt{key} \end{aligned}$$

where $\texttt{lfsr}_{64}(D)$ refers to a 64-bit counter and $B$ refers to a domain separation byte. Therefore, the above implementation trick used in Romulus also applies to Skinny-AEAD. Moreover, since Skinny-AEAD allows blocks to be processed in parallel, we can have a look at the benefits of the Skinny-128 implementation that processes two blocks at a time.

## 3.4 SAEAES

SAEAES is an instantiation of the SAEB mode of operation [NMSS18] with the AES block cipher. Hereafter we only consider the primary member SAEAES-128-64-128. Since the SAEB mode of operation does not allow to process several blocks in parallel, the underlying fixsliced AES-128 implementation process the same block twice. Therefore the AES-128 performance reported in Table 1 are actually reduced by a factor of 2. For the key expansion, we rely on a constant-time implementation of the AES-128 key schedule in order to exclusively consider constant-time implementations in our benchmark.

### 3.5 Implementation results

Hereafter we report implementation results on ARM Cortex-M3/4 based microcontrollers for the considered NIST LWC candidates mentioned above. For all the implementations, the internal primitive is written in assembly language while the mode is handled by C code. The code was compiled by `arm-none-eabi-gcc 9.2.1` using the flag `-O3` for optimized speed results. The Table 3 reports performance for small messages along with memory requirements whereas the Table 4 reports performance for payloads from 0 to 256 bytes. A graphical representation is provided in Figure 3 for greater clarity.

| **Algorithm** | **Parallel Blocks** | **Speed** (cycles) M3 | M4 | **Code size** (bytes) | **RAM** (bytes) In/Output | Stack |
|---|---|---|---|---|---|---|
| GIFT-COFB | 1 | 5 293 | 5 334 | 5 140 | 428 | 100 |
| Romulus-N1 | 1 | 13 352 | 13 511 | 9 132 | 1 092 | 96 |
| Romulus-N1+ | 1 | 10 041 | 10 178 | 8 032 | 836 | 96 |
| Romulus-M1 | 1 | 21 583 | 21 903 | 9 512 | 1 245 | 96 |
| Romulus-M1+ | 1 | 16 242 | 16 509 | 8 442 | 989 | 96 |
| Skinny-AEAD-M1 | 1 | 19 522 | 19 697 | 8 616 | 1 232 | 96 |
| Skinny-AEAD-M1+ | 1 | 14 762 | 14 910 | 7 564 | 976 | 96 |
| Skinny-AEAD-M1 | 2 | 21 138 | 21 334 | 15 302 | 2 384 | 96 |
| Skinny-AEAD-M1+ | 2 | 16 129 | 16 305 | 13 294 | 1 872 | 96 |
| SAEAES-128-64-128 | 1 | 18 496 | 18 632 | 5 030 | 452 | 148 |

Table 3: Implementation results on ARM Cortex-M3/4 for some NIST LWC round 2 candidates when processing 16 bytes of message along with 16 bytes of additional data. In/Output refers to the amount of memory needed to store the input and ouput plus the temporary variables (including the round keys).

As expected, GIFT-COFB is the candidate that shows the most outstanding results regarding both efficiency and compactness. For Skinny-based candidates, it results that Romulus-N is the fastest one for small messages and is only outperformed by Skinny-AEAD when 2 blocks are processed in parallel. However this parallelism requires twice the RAM to store the pre-computed round tweakeys which might be troublesome in practice for resource-constrained devices. Although SAEAES-128-64-128 is penalized by the fact that the AES-128 implementation processes a single block at a time instead of two, it shows decent results and is neck and neck with Skinny-AEAD for messages smaller than 64 bytes.

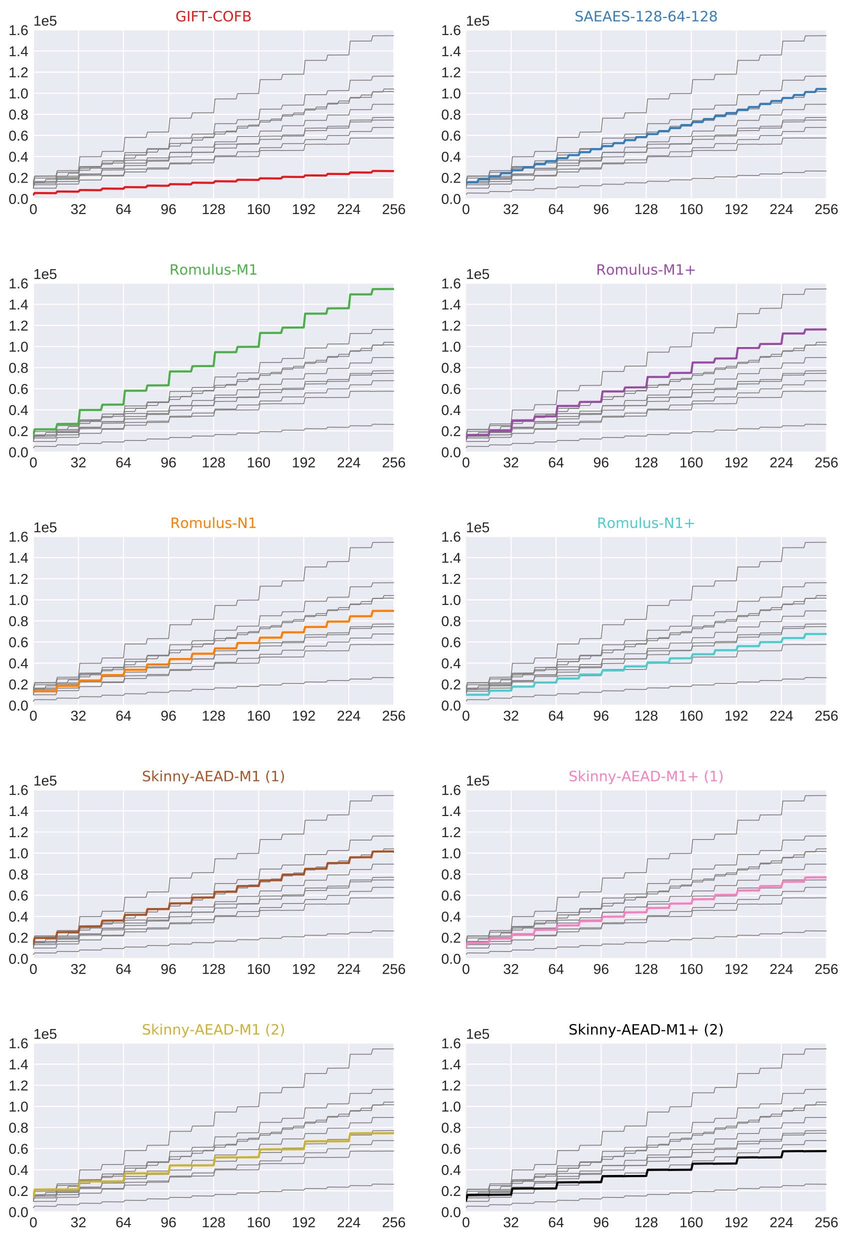


Fig. 3: Benchmark of some NIST LWC round 2 candidates on ARM Cortex-M3. The x-axis and y-axis refer to the message length (in bytes) and the corresponding execution time (in clock cycles), respectively. Note that 16 bytes of additional data are taken into account for all measurements. For Skinny-AEAD, the number enclosed in parantheses refer to the number of blocks processed in parallel.

| **Algorithm** | **Parallel Blocks** | Message size (bytes) | | | | |
|---|---|---|---|---|---|---|
| | | 16 | 32 | 64 | 128 | 256 |
| GIFT-COFB | 1 | 5 293 | 6 718 | 9 502 | 15 070 | 26 206 |
| Romulus-N1 | 1 | 13 352 | 18 475 | 28 625 | 48 925 | 89 525 |
| Romulus-N1+ | 1 | 10 041 | 13 918 | 21 586 | 36 922 | 67 594 |
| Romulus-M1 | 1 | 21 583 | 26 718 | 44 986 | 81 496 | 154 516 |
| Romulus-M1+ | 1 | 16 242 | 20 136 | 33 872 | 61 318 | 116 210 |
| Skinny-AEAD-M1 | 1 | 19 522 | 24 979 | 35 893 | 57 721 | 101 377 |
| Skinny-AEAD-M1+ | 1 | 14 762 | 18 902 | 27 182 | 43 742 | 76 862 |
| Skinny-AEAD-M1 | 2 | 21 138 | 21 388 | 28 992 | 44 200 | 74 616 |
| Skinny-AEAD-M1+ | 2 | 16 129 | 16 379 | 22 274 | 34 064 | 57 644 |
| SAEAES-128-64-128 | 1 | 18 496 | 24 198 | 35 602 | 58 410 | 104 026 |

Table 4: Benchmarking results on ARM Cortex-M3 for different message lengths along with 16 bytes of additional data.

## 4 Conclusion and perspectives

In this work, we reported fixsliced implementation results for some NIST LWC round 2 candidates. We also introduced a fast software implementation of the Skinny tweakey schedule when considering a bitsliced implementation. It is very likely that other internal primitives used in other candidates may benefit from the fixslicing implementation strategy and further examination on a case-by-case basis would be needed to get the whole picture in the context of the NIST LWC standardization process.

## References

ANP20. Alexandre Adomnicai, Zakaria Najm, and Thomas Peyrin. Fixslicing: A New GIFT Representation: Fast Constant-Time Implementations of GIFT and GIFT-COFB on ARM Cortex-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):402–427, Jun. 2020. https://tches.iacr.org/index.php/TCHES/article/view/8595.

AP20. Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like Ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. Cryptology ePrint Archive, Report 2020/1123, 2020. https://eprint.iacr.org/2020/1123.

BCI$^+$20. Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT-COFB. Cryptology ePrint Archive, Report 2020/738, 2020. https://eprint.iacr.org/2020/738.

BJK$^+$20. Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. SKINNY-AEAD and SKINNY-Hash. *IACR Transactions on Symmetric Cryptology*, 2020(S1):88–131, Jun. 2020. https://tosc.iacr.org/index.php/ToSC/article/view/8619.

BKL$^+$07. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, pages 450–466, 2007.

BPP$^+$17. Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A Small Present. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 321–345, Cham, 2017. Springer International Publishing.

IKMP20. Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. 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.

Jea16. Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2016.

JNP14. Jérémy Jean, Ivica Nikolic, and Thomas Peyrin. Tweaks and Keys for Block Ciphers: The TWEAKEY Framework. In *ASIACRYPT (2)*, volume 8874 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 2014.

NMSS18. Yusuke Naito, Mitsuru Matsui, Takeshi Sugawara, and Daisuke Suzuki. SAEB: A Lightweight Blockcipher-Based AEAD Mode of Operation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):192–217, May 2018. https://tches.iacr.org/index.php/TCHES/article/view/885, DOI=10.13154/tches.v2018.i2.192-217.

Pey20. Thomas Peyrin. New Romulus and SKINNY-AEAD variants. Announcement to the NIST lwc-forum mailing list, May 2020. https://groups.google.com/a/list.nist.gov/forum/#!forum/lwc-forum.

RAL17. Tiago B. S. Reis, Diego F. Aranha, and Julio López. PRESENT runs fast - efficient and secure implementation in software. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 644–664, 2017.

RPM20. Sebastian Renner, Enrico Pozzobon, and Jürgen Mottok. NIST LWC Software Performance Benchmarks on Microcontrollers, 2020. https://lwc.las3.de.

Wea20. Rhys Weatherley. Lightweight Cryptography Primitives, 2020. https://github.com/rweather/lightweight-crypto.