Rotom: A Microcontroller-Oriented Tweakable Block Cipher Using Modular Multiplication (Full Version)

Kento Morita[1], Kazuhiko Minematsu[2,4], Kosei Sakamoto[3,4], Kazuma Taka[5,4], Rei Ueno[6] and Takanori Isobe[4]

[1] University of Hyogo, Kobe, Japan, ad25i071@guh.u-hyogo.ac.jp
[2] NEC Corporation, Kawasaki, Japan, k-minematsu@nec.com
[3] Mitsubishi Electric Corporation, Kamakura, Japan, sakamoto.kosei@dc.mitsubishielectric.co.jp
[4] The University of Osaka, Suita, Japan, takanori.isobe@ist.osaka-u.ac.jp
[5] National Institute of Information and Communications Technology, Koganei, Japan, kazuma.taka@nict.go.jp
[6] Kyoto University, Kyoto, Japan, ueno.rei.2e@kyoto-u.ac.jp

Abstract. We propose Rotom, a family of 128-bit tweakable block ciphers designed for efficient software implementation on 32-bit microcontrollers. The central design idea is to use modular multiplication as a compact and software-efficient source of strong nonlinearity. Rotom instantiates this idea with a fixed 16-bit S-box built from odd constant multiplication modulo 2^{16} interleaved with lightweight linear maps, and combines it with a software-friendly recursive linear layer over four 32-bit words, a compact tweakey schedule, and modular-addition boundary key mixing. In contrast to typical ARX-based ciphers, Rotom provides explicit differential and linear characteristic bounds by design. These bounds are obtained by exhaustively computing the local transition probabilities of its fixed 16-bit S-box and combining them with active-S-box bounds induced by its recursive linear layer. Implementation results on ARM Cortex-M3 and Cortex-M4 show that Rotom achieves cycle counts close to Speck, substantially lower than those of MSX, and faster than representative fixsliced AES implementations, while providing tweak support and cipher-level differential and linear bounds. Overall, Rotom combines the efficiency of software-oriented arithmetic instructions with the explicit security evaluation enabled by a fixed-S-box structure.

Keywords: Tweakable Block Cipher · Constant Multiplication · Recursive Diffusion Layer · MCU

1 Introduction

Lightweight cryptography for microcontrollers is an important building block for embedded and massive IoT systems. Low-power wide-area networks such as LoRaWAN and Sigfox have been widely studied as connectivity technologies for massive numbers of low-power devices, where long-range communication is achieved under severe constraints on energy, data rate, and device cost [RKS17, HSM+20]. More recently, Ambient IoT has emerged in the 3GPP and 6G context as a candidate technology for extremely low-power or battery-less devices that harvest energy from their environment [BMP+23, ZXM+24]. These trends further increase the need for symmetric-key primitives that can run efficiently on constrained microcontrollers.

In such environments, encryption must be fast enough to meet latency and energy requirements, while its implementation must fit within limited ROM and RAM budgets. A common target for software implementations is the ARM Cortex-M class of microcontrollers, in particular Cortex-M3 and Cortex-M4. These platforms are widely used in embedded systems and provide efficient word-level arithmetic and bitwise operations, including integer multiplication, shifts, and rotations [Arm10a, Arm10b]. Therefore, primitives for this environment should be evaluated not only by their security margin, but also by execution cycles, code size, and RAM consumption on such microcontrollers.

For this reason, many software-oriented primitives have been designed or optimized for microcontrollers. Word-oriented designs such as Speck [BSS+15] and Chaskey [MMH+14] achieve high software efficiency by relying on simple arithmetic and bitwise operations. At the same time, optimized implementations of existing ciphers, such as fixsliced implementations of AES and SKINNY [AP21] and of GIFT [BPP+17, ANP20], have significantly improved the performance of S-box-based designs on small processors.

Among software-oriented designs, ARX-based ciphers, and Speck in particular, remain among the leading performance benchmarks for microcontrollers, offering exceptionally low cycle counts, code size, and RAM usage. However, their security evaluation is less direct than that of traditional S-box-based designs. For Speck, extensive automatic-search studies cover differential and linear trails [FWG+16, SHY16, SWW21, KSI25], but tight upper bounds on differential and linear characteristic probabilities are available only for a limited number of rounds. More recent ARX-oriented designs, such as SPARX and the Alzette-based CRAX/TRAX constructions, represent important steps toward more analyzable ARX designs through long-trail arguments or structured ARX-boxes [DPU+16, BBdS+20]. These designs target a different region of the design space, trading some performance for stronger design-time security arguments. Consequently, Speck remains the natural speed-oriented benchmark against which new software ciphers for microcontrollers are compared.

A recent proposal, MSX [MSR+25], addresses this gap from a different direction by using integer multiplication to obtain strong security arguments against differential and linear attacks. This gives a more structured and provable security argument than typical ARX designs. However, this additional assurance comes with a performance loss, and MSX is substantially slower than Speck on microcontrollers. Similarly, modern analyzable ARX constructions and highly optimized fixsliced implementations of AES, SKINNY, and GIFT do not improve on Speck as a pure software-speed baseline in many microcontroller settings.

ARX designs and MSX realize their nonlinear operations with arithmetic instructions rather than table lookups or bitwise Boolean functions, which keeps their implementation cost low on microcontrollers. A recent design that adopts such arithmetic-based nonlinearity is mid-pSquare [BGM+25]. Primarily targeting efficient prime-field masking, it is designed as a tweakable block cipher over the 31-bit Mersenne prime field for 32-bit microcontrollers. mid-pSquare is further discussed in Sections 3.3.1 and 5.3.

A tweakable block cipher (TBC) [LRW02] is an extended form of an ordinary block cipher so that it takes a public input called a tweak, in addition to the message and the key. A pair of a tweak and a key determines the permutation over the message space. A TBC is a powerful primitive. A tweak can be used to encode addresses, counters, domain identifiers, or protocol context for domain separation purposes. Moreover, TBCs efficiently provide strong message authentication or authenticated encryption (see Section 3).

Our goal is therefore not to outperform Speck as a pure cycle-count baseline. Rather, we aim to approach Speck-like software performance while adding two properties that Speck does not provide: native tweakability and explicit cipher-level differential and linear characteristic bounds.

Our Contribution. To achieve the above goal, we propose Rotom, a 128-bit TBC suitable for microcontroller-oriented software implementations. The main round components are built only from 16-bit modular multiplication by fixed odd constants, XORs, shifts, and rotations. Modular multiplication provides a compact and low-cost source of high nonlinearity on microcontrollers equipped with integer multiplication instructions. Rotations are also well suited to the target platforms, since they can be implemented using barrel-shift operations or rotated operands with little additional cost. Moreover, the use of fixed constants and a fixed sequence of arithmetic and logical operations enables constant-time software implementations on platforms with data-independent integer multiplication. The main features of Rotom are as follows.

- **A Multiplication-based 16-bit S-box.** We design a fixed 16-bit permutation S_{16} from odd constant multiplication modulo 2^{16} interleaved with lightweight bijective linear maps. Since S_{16} is public and fixed, its full differential and linear transition tables can be computed once. The 16-bit cell size also allows these local transitions to be incorporated into state-of-the-art SAT-based characteristic search methods [SWW21]. By combining the local S-box probabilities with active-S-box bounds, Rotom provides explicit cipher-level differential and linear characteristic bounds. Moreover, since the only non-linear operation is a constant multiplication (linear over the integer ring), S_{16} can be masked using linear gadgets together with arithmetic-to-Boolean (A2B) and Boolean-to-arithmetic (B2A) mask conversions but no masked field-multiplier is required.

- **A Software-Oriented Recursive Diffusion Layer.** We adapt recursive diffusion layers, which have mainly been studied as compact diffusion layers with hardware-oriented motivations [SDMS12, AF14], to a software-oriented setting. The resulting linear layer is an in-place word-level permutation over four 32-bit words. It consists of a small number of XOR-rotation updates, achieves high diffusion over the eight 16-bit S-box cells, and has an inverse with comparable cost. Thus, both encryption and decryption use the same lightweight word-level structure.

- **Native Tweak Support with Lightweight Boundary Hardening.** The tweakey schedule reuses the same nonlinear and linear layers as the data path, avoiding a separate heavy tweak-update primitive while supporting related-tweak analysis. At the input and output boundaries, Rotom uses word-wise modular key addition together with the linear layer $\mathcal{M}$. The modular additions introduce carry-dependent key terms, and the adjacent linear layer mixes these dependencies across the state. This makes extensions from reduced-round distinguishers to practical key-recovery attacks more difficult, especially for integral-type and outer-round extension attacks. Furthermore, adopting modular addition makes it possible to leverage the stronger security of arithmetic masking when the leakage follows a Hamming weight (HW) model.

- **Efficient ARM Cortex-M implementations.** We implement Rotom on ARM Cortex-M3 and Cortex-M4 and evaluate execution cycles, ROM size, and RAM consumption. In the fully unrolled setting with precomputed round tweakeys, the 128-bit-key variants encrypt one 128-bit block in 559/519 cycles for Rotom-128* and 651/603 cycles for Rotom-128 on Cortex-M3/M4, respectively. These results are close to Speck, substantially faster than MSX, and faster than representative fixsliced AES implementations, while retaining native tweak support and explicit cipher-level differential and linear bounds. The 256-bit-key variants are likewise substantially faster than the corresponding MSX variants and remain competitive with Speck-128/256 despite providing native tweak support.

Encryption and decryption use approximately 40 bytes of stack, and the precomputed round-tweakey buffers remain small. We also provide loop-based implementations that reduce ROM size with only moderate performance loss. These results show that the multiplication-based S-box, software-oriented recursive linear layer, and compact tweakey schedule translate into practical performance on 32-bit microcontrollers.

Furthermore, from the perspective of side-channel resistance, we show that Rotom admits constant-time implementations and that it can be masked using linear gadgets and A2B/B2A conversions without field-multiplier gadgets.

Together, these contributions give a compact TBC whose security evaluation and software cost are co-designed for 32-bit microcontrollers. The reference C and Cortex-M implementations, together with test vectors for Rotom, are available. [1]

2 Specification

Rotom is a family of 128-bit TBCs with a 128-bit tweak and a κ-bit key, where $\kappa \in \{128, 256\}$. It consists of four variants: Rotom-128 and Rotom-128* with 128-bit keys, and Rotom-256 and Rotom-256* with 256-bit keys. The starred variants are data-limited, round-reduced variants of the corresponding full variants, as specified in Section 2.4. The full variants Rotom-128 and Rotom-256 use 6 and 7 rounds, respectively, whereas Rotom-128* and Rotom-256* use 5 and 6 rounds, respectively.

Basic operations. For an n-bit string x and an integer $s \in \{0, \ldots, n-1\}$, let $x \ll s$ and $x \gg s$ denote the logical left and right shifts of x by s bits, respectively.

For an n-bit string $x = (x_0, x_1, \ldots, x_{n-1})$ and an integer $r \in \{1, \ldots, n-1\}$, let $x \lll_n r = (x_r, \ldots, x_{n-1}, x_0, \ldots, x_{r-1})$ and $x \ggg_n r = (x_{n-r}, \ldots, x_{n-1}, x_0, \ldots, x_{n-r-1})$ denote the left and right rotations of x by r bits, respectively. We define $x \lll_n 0 = x \ggg_n 0 = x$.

State Representation. A 128-bit value X is represented as an ordered sequence of bytes $(B_0, B_1, \ldots, B_{15})$, written as

$$X = B_0 \parallel B_1 \parallel \cdots \parallel B_{15},$$

where B_0 is the first byte in the input/output representation. Multi-byte values are parsed in little-endian order; thus, B_0 is the least significant byte of both the first 16-bit cell and the first 32-bit word.

The design uses two granularities. A 16-bit cell is the unit on which the nonlinear permutation is evaluated and active S-boxes are counted, whereas a 32-bit word is the unit used by the linear layer $\mathcal{M}$, whitening, and word-wise key operations.

Accordingly, the same 128-bit value is viewed either as eight 16-bit cells or as four 32-bit words:

$$\begin{aligned} &\langle X \rangle_{16} = (x_0, x_1, \ldots, x_7), && x_i = B_{2i+1} \parallel B_{2i} && (0 \le i < 8), \\ &\langle X \rangle_{32} = (X_0, X_1, X_2, X_3), && X_i = B_{4i+3} \parallel B_{4i+2} \parallel B_{4i+1} \parallel B_{4i} && (0 \le i < 4). \end{aligned}$$

Cells and words are treated as unsigned integers for arithmetic.

2.1 Round Components

We define two permutations $\mathcal{S}, \mathcal{M} : \{0,1\}^{128} \to \{0,1\}^{128}$, which are used as the basic components in the overall cipher. Here, $\mathcal{S}$ is a multiplication-based nonlinear layer, while $\mathcal{M}$ is a linear layer. We also define the round constants RC_r, which are injected only in the tweakey schedule.

[1] https://github.com/moritakento0515/Rotom_imp.git

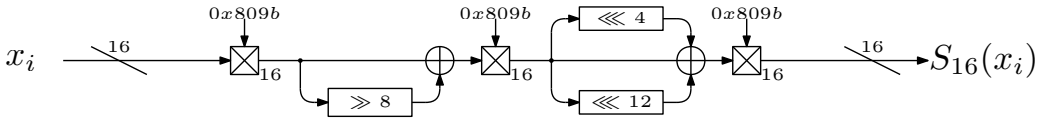


Figure 1: Structure of the 16-bit nonlinear permutation S_{16}.

Nonlinear Layer $\mathcal{S}$. For a 128-bit state X with $\langle X \rangle_{16} = (x_0, \ldots, x_7)$, we define

$$\big\langle \mathcal{S}(X) \big\rangle_{16} = \big(S_{16}(x_0), \ldots, S_{16}(x_7)\big), \text{ and } \big\langle \mathcal{S}^{-1}(X) \big\rangle_{16} = \big(S_{16}^{-1}(x_0), \ldots, S_{16}^{-1}(x_7)\big).$$

Let $c = \texttt{0x809b}$ and $c^{-1} = \texttt{0x6593}$, where $cc^{-1} \equiv 1 \pmod{2^{16}}$. We define

$$M_c(x) = cx \bmod 2^{16}, \quad M_{c^{-1}}(x) = c^{-1}x \bmod 2^{16},$$
$$L_{\mathrm{sh}}(x) = x \oplus (x \gg 8), \quad L_{\mathrm{rot}}(x) = x \oplus (x \lll_{16} 4) \oplus (x \lll_{16} 12).$$

The 16-bit nonlinear permutations are defined as

$$S_{16} = M_c \circ L_{\mathrm{rot}} \circ M_c \circ L_{\mathrm{sh}} \circ M_c,$$
$$S_{16}^{-1} = M_{c^{-1}} \circ L_{\mathrm{sh}} \circ M_{c^{-1}} \circ L_{\mathrm{rot}} \circ M_{c^{-1}}.$$

Since c is odd and both linear maps are self-inverse, S_{16}^{-1} is the inverse of S_{16}. Figure 1 illustrates S_{16}. A diagram of S_{16}^{-1} and pseudocode for both S_{16} and S_{16}^{-1} are provided in Figure 5 and Algorithm 2, respectively, in Supplementary Material B.

Linear Layer $\mathcal{M}$. For a 128-bit state X represented as four 32-bit words $\langle X \rangle_{32} = (X_0, X_1, X_2, X_3)$, the linear layer $\mathcal{M}$ consists of three in-place stages. In stage j, the four state words are updated sequentially in the following order:

$$X_0 \leftarrow X_0 \oplus (X_{p_0^{(j)}} \lll_{32} k_j),$$
$$X_1 \leftarrow X_1 \oplus (X_{p_1^{(j)}} \lll_{32} k_j),$$
$$X_2 \leftarrow X_2 \oplus (X_{p_2^{(j)}} \lll_{32} k_j),$$
$$X_3 \leftarrow X_3 \oplus (X_{p_3^{(j)}} \lll_{32} k_j).$$

Here, $p^{(j)} = \big(p_0^{(j)}, p_1^{(j)}, p_2^{(j)}, p_3^{(j)}\big)$ denotes the source pattern of stage j, and k_j denotes its rotation amount. Since the updates are performed in place, $X_{p_i^{(j)}}$ denotes the current value of the source word at the time of the corresponding assignment and may therefore have already been updated earlier in the same stage.

The parameters of the three stages are

j	$p^{(j)}$	k_j
0	$(3, 2, 1, 0)$	24
1	$(2, 3, 1, 0)$	0
2	$(2, 3, 0, 1)$	13

and the stages are applied in the order $j = 0, 1, 2$. The resulting state is denoted by $\mathcal{M}(X)$.

Figure 2 illustrates the procedure of $\mathcal{M}$.

The inverse permutation $\mathcal{M}^{-1}$ is obtained by executing the assignments in the reverse order. The explicit algorithms for $\mathcal{M}$ and $\mathcal{M}^{-1}$, together with a diagram of the inverse procedure, are provided in Figure 6 and Algorithm 3 of the Supplementary Material B.

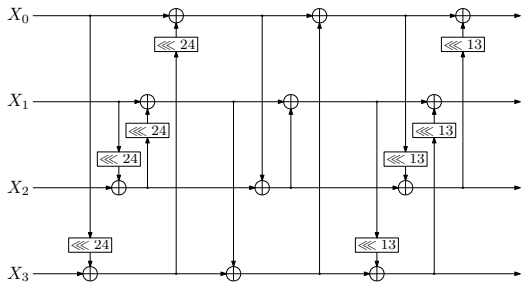


Figure 2: Linear layer $\mathcal{M}$.

Round Constants. The round constants are derived from the hexadecimal fractional part of

$$\pi = 3.\texttt{243f6a8885a308d313198a2e03707344}\ldots_{16}.$$

Let $C_0, \ldots, C_7$ be the first eight consecutive 32-bit words of this fractional part, as listed in Table 1. For $0 \le r \le 7$, the 128-bit round constant RC_r is defined by

$$\langle \mathrm{RC}_r \rangle_{32} = (C_r, 0, 0, 0).$$

Thus, RC_r is injected only into the first 32-bit word of the tweakey state.

Table 1: The 32-bit words C_r used to define the round constants RC_r.

r	C_r	r	C_r
0	0x243f6a88	4	0xa4093822
1	0x85a308d3	5	0x299f31d0
2	0x13198a2e	6	0x082efa98
3	0x03707344	7	0xec4e6c89

2.2 Overall Construction

Overview. We first specify the overall construction of the 128-bit-key variants. The modifications for the 256-bit-key variants are described in Section 2.3.

Encryption consists of three parts: the whitening part, the tweakey schedule, and the data-processing part. The tweakey schedule first generates the round tweakeys. The input-side whitening is then applied to the plaintext, followed by the data-processing part and the output-side whitening. Figure 3 illustrates the interaction among these three parts, while Algorithm 1 gives the complete encryption procedure and its two subroutines. Throughout the algorithm, R denotes the fixed number of rounds of the selected instance and is therefore omitted from the procedure inputs.

The tweakey schedule is unchanged in decryption and generates the same round tweakeys as in encryption. Decryption applies the inverse data-processing and whitening operations in the reverse order of encryption. Its complete procedure is illustrated in Figure 7 of the Supplementary Material B.

Whitening Part. As shown in Figure 3, the whitening part surrounds the data-processing part at the input and output boundaries. It consists of word-wise modular key addition together with the adjacent applications of the linear layer $\mathcal{M}$.

Let

$$\langle X \rangle_{32} = (X_0, X_1, X_2, X_3) \quad \text{and} \quad \langle K \rangle_{32} = (K_0, K_1, K_2, K_3).$$

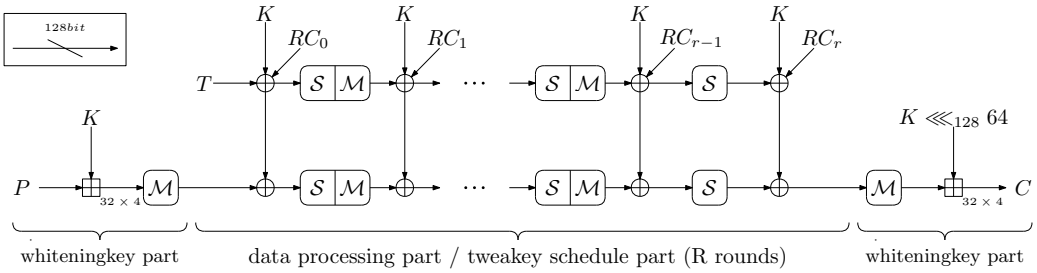


Figure 3: Overall encryption procedure. The data-processing part and the tweakey schedule use the same round components, and the final round of each branch omits $\mathcal{M}$.

We define word-wise addition and subtraction by

$$\langle X \boxplus_{32\times4} K \rangle_{32} = \big((X_0 + K_0) \bmod 2^{32}, \ldots, (X_3 + K_3) \bmod 2^{32}\big),$$
$$\langle X \boxminus_{32\times4} K \rangle_{32} = \big((X_0 - K_0) \bmod 2^{32}, \ldots, (X_3 - K_3) \bmod 2^{32}\big).$$

Thus, carries and borrows are confined to individual 32-bit words and do not propagate between words.

In encryption, the input-side whitening first adds the master key and then applies the linear layer:

$$X \leftarrow X \boxplus_{32\times4} K, \qquad X \leftarrow \mathcal{M}(X).$$

The output-side whitening applies the linear layer and then adds a 64-bit rotation of the master key:

$$X \leftarrow \mathcal{M}(X), \qquad X \leftarrow X \boxplus_{32\times4} \big(K \lll_{128} 64\big).$$

In decryption, these boundary operations are inverted by replacing word-wise addition with word-wise subtraction and $\mathcal{M}$ with $\mathcal{M}^{-1}$, while reversing their order.

Tweakey Schedule Part. The tweakey schedule generates the $R+1$ round tweakeys $TK[0], TK[1], \ldots, TK[R]$ from the 128-bit tweak T and the 128-bit master key K. It maintains a 128-bit internal state U, initialized by $U \leftarrow T$.

For each $r = 0, \ldots, R-1$, the schedule first generates the round tweakey $TK[r] = U \oplus K \oplus \mathrm{RC}_r$. The internal state is then updated from this round tweakey by applying the nonlinear layer: $U \leftarrow \mathcal{S}(TK[r])$. If $r < R-1$, the linear layer is subsequently applied as $U \leftarrow \mathcal{M}(U)$. Thus, the final iteration omits $\mathcal{M}$, as shown in Figure 3.

After the R iterations, the final round tweakey is generated from the resulting internal state as

$$TK[R] = U \oplus K \oplus \mathrm{RC}_R.$$

Data-Processing Part. The data-processing part updates a 128-bit data state X using the $R+1$ round tweakeys $TK[0], TK[1], \ldots, TK[R]$ generated by the tweakey schedule.

For each $r = 0, \ldots, R-1$, the state is first combined with the corresponding round tweakey and then passed through the nonlinear layer:

$$X \leftarrow X \oplus TK[r], \qquad X \leftarrow \mathcal{S}(X).$$

If $r < R-1$, the linear layer is subsequently applied as

$$X \leftarrow \mathcal{M}(X).$$

Thus, the final round omits $\mathcal{M}$, as shown in Figure 3.

Algorithm 1 Overall encryption procedure and its subroutines.

```
function ENCRYPT(P, T, K)
    TK[0...R] ← TKSCHEDULE(T, K)
    X ← P ⊞_{32×4} K
    X ← M(X)
    X ← DATAPROC(X, TK)
    X ← M(X)
    C ← X ⊞_{32×4} (K ⋘_128 64)
    return C
end function
```

```
function TKSCHEDULE(T, K)
    U ← T
    for r ← 0 to R − 1 do
        TK[r] ← U ⊕ K ⊕ RC_r
        U ← S(TK[r])
        if r < R − 1 then
            U ← M(U)
        end if
    end for
    TK[R] ← U ⊕ K ⊕ RC_R
    return TK[0...R]
end function
```

```
function DATAPROC(X, TK)
    for r ← 0 to R − 1 do
        X ← X ⊕ TK[r]
        X ← S(X)
        if r < R − 1 then
            X ← M(X)
        end if
    end for
    X ← X ⊕ TK[R]
    return X
end function
```

After the R rounds, the final round tweakey is added to the state:

$$X \leftarrow X \oplus TK[R].$$

The resulting state is then processed by the output-side whitening operations.

2.3 256-Bit-Key Variant

The 256-bit-key variants differ from the 128-bit-key variants only in the number of rounds, the tweakey schedule, and the whitening part. All round components and the data-processing procedure remain unchanged. The full and data-limited variants use seven and six rounds, respectively. The complete encryption and decryption procedures are illustrated in Figure 8 in Supplementary Material B.

Let the 256-bit master key be divided into two 128-bit halves as $K = K^{[0]} \parallel K^{[1]}$.

Whitening Part. The 256-bit-key variant uses four whitening keys derived by word-wise addition of the two 128-bit key halves:

$$\begin{aligned}
W_0 &= K^{[0]} \boxplus_{32\times4} K^{[1]},\\
W_1 &= K^{[0]} \boxplus_{32\times4} \left(K^{[1]} \lll_{128} 64\right),\\
W_2 &= \left(K^{[0]} \lll_{128} 32\right) \boxplus_{32\times4} K^{[1]},\\
W_3 &= \left(K^{[0]} \lll_{128} 32\right) \boxplus_{32\times4} \left(K^{[1]} \lll_{128} 64\right).
\end{aligned}$$

The input-side whitening is

$$\begin{aligned}
X &\leftarrow X \boxplus_{32\times4} W_0,\\
X &\leftarrow \mathcal{M}(X),\\
X &\leftarrow X \boxplus_{32\times4} W_1,\\
X &\leftarrow \mathcal{M}(X).
\end{aligned}$$

After the data-processing part, the output-side whitening applies these operations in the reverse order:

$$X \leftarrow \mathcal{M}(X),$$
$$X \leftarrow X \boxplus_{32\times 4} W_2,$$
$$X \leftarrow \mathcal{M}(X),$$
$$X \leftarrow X \boxplus_{32\times 4} W_3.$$

Decryption applies the inverse whitening operations in reverse order, replacing word-wise additions with word-wise subtractions and $\mathcal{M}$ with $\mathcal{M}^{-1}$.

Tweakey Schedule. The tweakey schedule is obtained from that of the 128-bit-key variant by replacing the master key K with the round-dependent key half $K^{(r)}$ when generating $TK[r]$, where

$$K^{(r)} = \begin{cases} K^{[0]}, & r \equiv 0 \pmod 2, \\ K^{[1]}, & r \equiv 1 \pmod 2. \end{cases}$$

Thus, $TK[r] = U \oplus K^{(r)} \oplus RC_r$ for $0 \le r \le R$, where the update rule for U is unchanged from the 128-bit-key variant.

2.4 Security Claims

We instantiate the generic construction Rotom as four variants: Rotom-128, Rotom-128*, Rotom-256, and Rotom-256*. The variants Rotom-128 and Rotom-128* use a 128-bit key, whereas Rotom-256 and Rotom-256* use a 256-bit key. The asterisk denotes a data-limited variant with one fewer round than the corresponding full variant.

The claims are stated in terms of time and data complexity. Following the security claims of QARMAv2 [ABD+23], we use a slack parameter ε to account for simple attack optimizations and constant-factor improvements. We set $\varepsilon = 16$.

128-bit key. The full 128-bit-key variant, Rotom-128, uses $R = 6$ and claims security up to time complexity

$$2^{128-\varepsilon} = 2^{112}$$

in both the single-tweak and related-tweak settings, for data complexity up to 2^{128} blocks per master key. The data-limited variant, Rotom-128*, uses $R = 5$ and makes the same time-complexity claim when the data complexity is limited to 2^{80} blocks per master key.

256-bit key. The full 256-bit-key variant, Rotom-256, uses $R = 7$ and claims security up to time complexity

$$2^{256-\varepsilon} = 2^{240}$$

in both the single-tweak and related-tweak settings, for data complexity up to 2^{128} blocks per master key. The data-limited variant, Rotom-256*, uses $R = 6$ and makes the same time-complexity claim when the data complexity is limited to 2^{80} blocks per master key.

For all four variants, we make no security claim in the related-key, known-key, or chosen-key settings. This scope is common among lightweight block-cipher designs [MSR+25, BBI+15, Ava17, BII+25, ABD+23].

3 Design Rationale

This section explains the design choices behind our TBC. We first state the design goals, and then describe the main design approach: fixed multiplication-based S-boxes, a software-oriented recursive linear layer, the tweakey schedule, and modular-addition whitening.

3.1 Design Goals

We target a 128-bit TBC with a 128/256-bit key and a 128-bit tweak. The design follows three requirements.

(1) **High performance on 32-bit microcontrollers.** The primary software targets are ARM Cortex-M3 and Cortex-M4. These cores are representative low-cost 32-bit microcontroller platforms for embedded and IoT devices. They dominate the 32-bit microcontroller market (e.g. [MCU]). They provide efficient word-level integer operations, including XORs, rotations, shifts, and ordinary integer multiplication, while not relying on large memory, lookup tables, SIMD instructions, or high-end processor features [Arm10a, Arm10b].

(2) **Explicit differential and linear bounds.** To provide strong security assurance, the design should provide explicit upper bounds on differential characteristic probabilities (DCPs) and linear characteristic probabilities (LCPs) at the cipher level, while maintaining high software efficiency on 32-bit microcontrollers.

(3) **Being tweakable.** A TBC [LRW02] is widely acknowledged as a powerful primitive, as it enables high quantitative security (also known as Beyond-birthday-bound security) and high efficiency, as exemplified by state-of-the-art TBC modes, e.g., Deoxys II (a winner of CAESAR competition) [JNPS21], ΘCB [KR11], and ZMAC [IMPS17]. Having a 128-bit block and tweak implies that we can implement standard block-cipher modes such as GCM or OCB by fixing a tweak, while the aforementioned TBC modes can also be implemented, showing versatility.

3.2 Approach

Existing Design and Limitations. Efficient block-cipher design for 32-bit microcontrollers typically follows two directions: ARX-style designs and S-box-based designs. ARX-style designs use additions, rotations, and XORs that map directly to word-level instructions [BSS+15, DPU+16, BBdS+20]. This makes designs such as Speck highly competitive in cycles, code size, and RAM usage. Their limitation is analysis: the same arithmetic structure makes tight DCP/LCP bounds difficult to obtain beyond a limited number of rounds [FWG+16, SHY16, SWW21, KSI25]. Structured ARX designs such as SPARX and the Alzette-based CRAX/TRAX constructions [DPU+16, BBdS+20] improve design-time security arguments, but the additional structure comes with extra implementation cost. As a result, they do not replace Speck as the main software-performance baseline on microcontrollers.

S-box-based designs provide a clearer analysis framework. Local DCPs and LCPs can be computed for the S-box and then combined with multi-round diffusion arguments or automated search. Their implementation challenge is that constant-time software cannot rely on ordinary lookup tables. Efficient table-free techniques such as bitslicing and fixslicing avoid this problem and have led to highly optimized software implementations [BPP+17, ANP20, AP21]. However, on small microcontrollers, they still require additional Boolean operations and data-layout overhead compared with ARX-style designs whose operations directly match word-level instructions. As a result, S-box-based ciphers often do not reach the performance of ARX-style designs such as Speck.

MSX [MSR+25] tackles this trade-off by implementing a universal hash function based on integer arithmetic and obtains provable differential and linear bounds, backed by the decorrelation theory [Vau03]. This demonstrates that multiplication is a useful source of nonlinearity for microcontroller-oriented ciphers. However, detailed security analysis beyond differential and linear attacks is difficult because of its unconventional design. Moreover, a large number of subkeys for the universal hash introduces lots of memory operations, which degrades the performance compared with Speck.

Our Design Overview. Rotom addresses this gap by turning modular multiplication into a fixed, analyzable S-box component and co-designing it with a software-oriented diffusion layer. At a high level, the design is based on the following three choices.

- Instead of using multiplication as a large keyed word-level nonlinear operation, Rotom uses odd-constant multiplication modulo 2^{16} interleaved with low-cost bijective linear maps. The resulting 16-bit S-box is public, fixed, and key-independent. Its DDT and LAT can therefore be exhaustively computed once and used as local bounds in the multi-round differential and linear analysis.

- The cipher-level design jointly selects this S-box and the software-oriented recursive linear layer $\mathcal{M}$. The search balances three quantities under a microcontroller-oriented cycle budget: the local DCP/LCP strength of the S-box, the active-cell diffusion provided by $\mathcal{M}$, and the cycle cost on ARM Cortex-M3 and Cortex-M4. Thus, the main design contribution is the combination of a fixed multiplication-based 16-bit S-box with an efficient word-level recursive diffusion layer, rather than the use of multiplication alone.

- The remaining components are chosen to add tweak support and boundary hardening without increasing the number of data-processing rounds. The tweakey schedule reuses $\mathcal{S}$ and $\mathcal{M}$, avoiding a separate heavy tweak-update primitive while supporting the related-tweak setting. At the input and output boundaries, lightweight modular-addition key mixing is used instead of additional S-box layers. Its carry propagation hardens reduced-round key-recovery extensions and is also useful against integral-type attacks, as discussed later.

The following subsections explain these design choices in detail.

3.3 Multiplication-Based Nonlinear Layer

This subsection explains the design rationale of the nonlinear layer $\mathcal{S}$. We first justify odd-constant multiplication modulo 2^n as the basic nonlinear operation, and then introduce the multiplication–linear construction used for the final 16-bit S-box.

3.3.1 Choice of Multiplicative Operation

The nonlinear operation should be efficient on 32-bit microcontrollers, table-free, key-independent, invertible as a fixed component, and suitable for systematic DCP/LCP evaluation. Multiplication modulo a power of two matches these requirements.

On ARM Cortex-M3 and Cortex-M4, the ordinary `MUL` instruction computes the lower 32 bits of a 32×32 product with robust one-cycle behavior, whereas full-product multiplication instructions do not necessarily have the same latency and timing properties [MSR+25, Por18]. Since the lower n bits of an integer product give multiplication modulo 2^n, fixed constant multiplication modulo 2^n can be implemented without an explicit reduction step. This makes it a natural low-latency operation for software implementations on 32-bit microcontrollers.

Candidate Forms of Multiplication. Multiplication has appeared in several symmetric-key designs. For example, IDEA uses multiplication modulo $2^{16}+1$, with data words multiplied by subkeys, while XMX and MultiSwap use modular multiplication in firmware-oriented designs [LM91, MNSV97, Scr01]. These examples motivate multiplication as a nonlinear arithmetic operation, but they do not directly match our goal of a fixed, key-independent, efficiently invertible component. For our purpose, we compare three choices: prime-field

multiplication, variable-operand integer multiplication, and fixed constant multiplication modulo 2^n.

Prime-field multiplication does not fit our low-latency target because it requires an additional reduction step. Even when the prime is close to a power of two, the reduction adds cost on small embedded processors. Recent prime-field designs reduce this cost by using a Mersenne prime. For example, mid-pSquare [BGM+25] is a TBC over the 31-bit Mersenne prime field that uses squaring as its nonlinear operation and is designed primarily for efficient prime-field masking. With $p = 2^{31} - 1$, the reduction can be implemented in constant time with a few additional instructions on Cortex-M4. However, the product of two 31-bit field elements has up to 62 bits and thus requires a full-product instruction such as `UMULL` instead of the one-cycle `MUL`. On Cortex-M3, this instruction takes multiple cycles and terminates early depending on the operand values [Por18], so constant-time prime-field multiplication becomes costly on Cortex-M3, one of our target platforms.

Variable-operand integer multiplication uses the hardware multiplier directly, but natural forms such as a $32 \times 32 \to 32$ lower-half product or a $16 \times 16 \to 32$ full product are not fixed n-bit permutations by themselves. They therefore require additional structure when used as an invertible nonlinear component. Moreover, their differential behavior depends on the other operand; for example, a difference in one operand can be annihilated when the other operand is zero. If one operand is key-dependent, the quality of the multiplication layer can also depend on the selected key value, leading to possible weak-multiplier or weak-key classes.

Under these requirements, fixed constant multiplication modulo 2^n is the most suitable basic operation. For a word size n and an odd constant c, the map

$$M_c^{(n)} : x \mapsto cx \bmod 2^n$$

is a permutation on n-bit words. Its inverse is multiplication by $c^{-1} \bmod 2^n$, so it is suitable for an invertible S-box construction and efficient decryption. Since c is a public design parameter chosen by search, the nonlinear component is fixed and independent of the key. This avoids multiplier-induced weak-key classes and allows its DCPs and LCPs to be evaluated once for all keys. We therefore use fixed odd constant multiplication modulo 2^n as the basic arithmetic operation for the nonlinear layer. This choice has also been used in MOE [BFL+21]. The concrete word size and the additional bit-mixing operations used to construct the final S-box are discussed next.

Masked S-box. S-box only utilizes a multiplication with a constant as a non-linear operation (over binary field) in combination with bit shifts and bit-wise XORs. As a constant multiplication is a linear operation over integer ring, the masked multiplication can be easily realized by applying the constant multiplication to each share independently. Let x be the input and x_i be its i-th arithmetic share in the d-th order arithmetic masking ($0 \leq i \leq d$). To realize masked $f(x) = cx$ with a constant c, we compute cx_i for each share x_i as $cx = c(x_0 + x_1 + \ldots x_d)$ in d-th order arithmetic masking. The bit shift and bit-wise XOR are linear operation over binary field, so they can be also masked with linear gadgets. Thus, the masked S-box of Rotom can be implemented with linear gadgets and arithmetic-to-Boolean (A2B) and Boolean-to-arithmetic (B2A) mask conversions, but no masked field-multiplication are required.

3.3.2 Design of Multiplication-Based S-box

Fixed odd multiplication modulo 2^n is table-free and invertible, but a single multiplication has structural weaknesses as a stand-alone S-box. We first explain these limitations and then introduce the multiplication–linear construction used to build our S-box.

Limitation of a Single Odd Multiplication Let $M_c(x) = cx \bmod 2^n$ for an odd constant c. Then M_c is a permutation on n-bit words. However, as a Boolean mapping, it is triangular. Writing

$$x = \sum_{i=0}^{n-1} x_i 2^i, \qquad M_c(x) = \sum_{i=0}^{n-1} y_i 2^i,$$

we have $y_i = x_i \oplus f_i(x_0, \ldots, x_{i-1})$ for some Boolean function f_i. Thus, the i-th output bit depends only on $x_0, \ldots, x_i$.

This triangular structure gives deterministic weak directions. For every t,

$$x \equiv x' \pmod{2^t} \quad \Longrightarrow \quad M_c(x) \equiv M_c(x') \pmod{2^t}.$$

Hence, high-bit differences cannot create lower-bit differences. In particular,

$$M_c(x \oplus 2^{n-1}) = M_c(x) \oplus 2^{n-1},$$

so the MSB difference propagates with probability one. Similarly, since c is odd,

$$M_c(x) \bmod 2 = x \bmod 2,$$

and the LSB mask is perfectly correlated through M_c. The probability-one propagation and related structural weaknesses of odd modular multiplication were also pointed out in the analysis of MOE [BFL+21].

Repeating odd multiplications alone does not remove this weakness. For odd constants a and b,

$$M_b \circ M_a = M_{ab \bmod 2^n},$$

which is again an odd constant multiplication and remains triangular. Therefore, a multiplication-based S-box must interleave odd multiplications with bit-mixing linear maps that move high-bit differences to lower positions and low-bit masks to higher positions. MOE [BFL+21] also considered placing a binary linear map between two modular multiplications, but did not instantiate this approach because of the additional sharing and recombination cost in its target setting. We adopt and extend this approach for software-oriented 16-bit S-boxes. This leads to the multiplication–linear construction described next.

Multiplication–Linear S-boxes A general candidate has the form

$$S_n = M_c \circ L_t \circ M_c \circ \cdots \circ L_1 \circ M_c,$$

where $M_c(x) = cx \bmod 2^n$ for an odd constant c, and each L_i is an invertible linear map over n-bit words. We call this family multiplication–linear S-boxes. The main instances in our search are $M_c \circ L_1 \circ M_c$ and $M_c \circ L_2 \circ M_c \circ L_1 \circ M_c$, which we refer to as the MLM and MLMLM forms, respectively. Longer multiplication–linear constructions are possible, but each additional multiplication or linear map increases the per-cell S-box cost, and this increase is multiplied by the eight cells in every nonlinear layer.

The internal linear maps break the alignment between weak directions of consecutive multiplications. They move high-bit differences, which would otherwise remain in the high-bit region, to positions where the next multiplication creates carry interaction. They also move low-bit masks away from positions with strong linear correlation. Thus, the maps L_i make consecutive multiplications interact through different bit positions.

$$S_n^{-1} = M_c^{-1} \circ L_1^{-1} \circ M_c^{-1} \circ \cdots \circ L_t^{-1} \circ M_c^{-1},$$

Hence, if the linear maps have inverses of the same cost, the inverse S-box has essentially the same cost as the forward S-box.

This construction strengthens the nonlinear component without changing the word size. For a fixed n, increasing the number of multiplication and linear layers enlarges the design space and can improve the best achievable local DCP/LCP behavior, while keeping the DDT/LAT dimension unchanged.

Cell Size. We instantiate the construction with $n = 16$. An 8-bit S-box is easier to analyze, but a 128-bit state then contains sixteen cells, increasing extraction, alignment, masking, and packing overhead. A 32-bit S-box matches the machine word size, but computing DCP/LCP bounds for a 32-bit fixed permutation is too expensive for our design search. Moreover, a larger word size does not remove the triangular weakness of odd multiplication.

The 16-bit choice gives a middle point between implementation cost and analysis cost. It gives enough DCP/LCP weight per active S-box while still allowing exhaustive offline evaluation of the fixed permutation. For DCP/LCP evaluation, the 128-bit state is viewed as eight 16-bit cells: $X = (x_0, x_1, \ldots, x_7)$. For implementation, the same state is packed into four 32-bit words: $X_i = x_{2i} + 2^{16} x_{2i+1}$ for $0 \leq i < 4$.

The nonlinear layer $\mathcal{S}$ extracts the two 16-bit cells from each 32-bit word, applies S_{16} to both cells, and packs the outputs back into one word. Thus, one application of $\mathcal{S}$ consists of eight applications of S_{16} and the corresponding packing overhead. The rest of the round, including $\mathcal{M}$, whitening, and word-wise key additions, operates directly on the packed 32-bit representation

Requirements for Linear Maps. The internal linear maps L_i must compensate for the directional weaknesses of odd multiplication while remaining efficient and invertible. We therefore restrict the search to 16-bit instruction-level permutations that satisfy three requirements:

- low implementation cost on Cortex-M processors;
- bijectivity with an inverse of comparable cost;
- movement of differences and masks away from the weak bit positions of odd multiplication.

The first requirement limits the operations to rotations, shifts, and XORs with shifted or rotated operands, which map efficiently to Cortex-M barrel-shift instructions.

Table 2 defines the resulting search space. The cost is the design-stage instruction count, including any preprocessing needed to represent a 16-bit value in a 32-bit register. The search enumerates all parameter tuples shown in the table and retains only maps that are bijective over 16-bit words. Equivalent parameterizations are not deduplicated, since they do not affect the maximum score obtained within each cost class. Counting parameter tuples without removing equivalent maps, the ROT, SRxor, 2ROTxor, 2SRxor, and 2SALTxor families contain 16, 8, 16, 64, and 112 candidates, respectively.

Standalone maps of the form $x \mapsto x \oplus (x \ll s)$ are excluded. Such maps move information only from lower bit positions to higher bit positions, and therefore do not compensate for the triangular weakness of odd multiplication, where high-bit differences do not naturally diffuse to lower positions. Left shifts are used only in mixed two-shift families such as 2SALTxor, where they are combined with shifts in the opposite direction.

The parameter restrictions in Table 2 enforce bijectivity and keep the inverse cost comparable. For example, $s \geq 8$ makes $x \mapsto x \oplus (x \gg s)$ involutive on 16-bit words, because the second shift vanishes. For mixed left/right shift families, the condition $s \neq t$ excludes the non-bijective cases. The two-rotation form uses rotations separated by 8 bits as a cheap bijective byte-mixing template. The final parameters are selected by the search below and specified in Section 2.1.

Table 2: Candidate families for the internal linear maps L_i. Only parameter choices that define bijections over 16-bit words are admissible. Cost values are cycle counts that already account for the Cortex-M barrel shifter, plus one additional instruction beyond the arithmetic operations themselves, required to correctly operate on 16-bit units within a 32-bit register, as detailed in Section 5.2.

Family	Cost	Admissible parameters	Form
ROT	2	$0 \le r < 16$	$x \mapsto x \lll_{16} r$
SRxor	2	$8 \le s < 16$	$x \mapsto x \oplus (x \gg s)$
2ROTxor	3	$0 \le r < 16$	$x \mapsto x \oplus (x \lll_{16} r) \oplus (\ x \lll_{16} (r+8))$
2SRxor	3	$8 \le s, s' < 16$	$x \mapsto x \oplus (x \gg s) \oplus (x \gg s')$
2SALTxor	3	$8 \le s, t < 16,\ s \ne t$	$x \mapsto x \oplus (x \ll s) \oplus (x \gg t)$, $x \mapsto x \oplus (x \gg s) \oplus (x \ll t)$

3.3.3 Cost-Aware Search for S-box Representatives

The previous subsection fixed the S-box design space: the cell size is 16 bits, the nonlinear operation is fixed odd multiplication modulo 2^{16}, and the internal linear maps are restricted to the low-cost families in Table 2. We now enumerate this space and select one representative for each implementation-cost class.

Keeping the cost classes separate is necessary because the locally strongest S-box is not necessarily the best cipher-level choice. A stronger local DCP/LCP score may require a more expensive S-box, whose cost is paid for all eight cells in every nonlinear layer. The final choice also depends on the cost and branch number of the diffusion layer and on the resulting number of rounds. The selected representatives are therefore passed to the active-S-box filtering and the subsequent linear-layer search.

Search Domain. We consider S-box candidates of the following two forms:

$$\text{MLM} : M_c \circ L_1 \circ M_c, \qquad \text{MLMLM} : M_c \circ L_2 \circ M_c \circ L_1 \circ M_c.$$

Here, for an odd constant c, $M_c(x) = cx \bmod 2^{16}$. The constant c is exhaustively enumerated over all odd 16-bit values, namely $c \in \{1, 3, \ldots, 2^{16} - 1\}$, which gives 2^{15} choices. All multiplication layers in the same candidate use the same constant c.

Each internal linear map L_i is selected from Table 2, and all admissible parameter choices are enumerated. For MLMLM candidates, (L_1, L_2) is an ordered pair.

Cost Classes. We classify candidates using the design-stage cycle model. One odd constant multiplication costs one cycle, and each internal linear map has the cost given in Table 2. The search space contains 24 (= 8 +16) parameter choices of cost 2 and 192 (= 16 + 64 + 112) parameter choices of cost 3, including duplicate parameterizations.

These choices give the five cycle classes in Table 3. The rightmost column reports the number of candidates enumerated in each class.

Table 3: Cycle classes and search-space sizes of multiplication–linear S-box candidates.

Structure	Linear-map costs	S-box cost	Number of candidates
MLM	2	4 cycles	786,432
MLM	3	5 cycles	6,291,456
MLMLM	$2+2$	7 cycles	18,874,368
MLMLM	$2+3$ or $3+2$	8 cycles	301,989,888
MLMLM	$3+3$	9 cycles	1,207,959,552

Table 4: Best multiplication–linear S-box representatives for each cycle class. Values are truncated to two decimal places.

Cost	Score $w(S)$	AS_{128}	Structure and parameters
4	4.10	31.21	$M_{\texttt{0x7a5b}} \circ \mathrm{SRxor}_{10} \circ M_{\texttt{0x7a5b}}$
5	5.35	23.92	$M_{\texttt{0x62ad}} \circ \mathrm{2ROTxor}(3,11) \circ M_{\texttt{0x62ad}}$
7	7.92	16.16	$M_{\texttt{0x2167}} \circ \mathrm{SRxor}_{10} \circ M_{\texttt{0x2167}} \circ \mathrm{SRxor}_{9} \circ M_{\texttt{0x2167}}$
8	9.17	13.95	$M_{\texttt{0x809b}} \circ \mathrm{2ROTxor}(4,12) \circ M_{\texttt{0x809b}} \circ \mathrm{SRxor}_{8} \circ M_{\texttt{0x809b}}$
9	10.30	12.42	$M_{\texttt{0xf693}} \circ \mathrm{2ROTxor}(3,11) \circ M_{\texttt{0xf693}} \circ \mathrm{2ROTxor}(3,11) \circ M_{\texttt{0xf693}}$

Selection Method. For every candidate S, we compute the differential weight $w_D(S)$, the linear weight $w_L(S)$, and the combined score $w(S)$, as defined in

Supplementary Material A, Section A.1.

Since the nonlinear layer must support both differential and linear bounds, we rank candidates by the combined score $w(S)$. Let C_τ be the set of candidates with estimated cost τ cycles. For each cost class, we select

$$S^\star_\tau = \arg\max_{S \in C_\tau} w(S).$$

Thus, the output of this search is not a single final S-box, but one best representative for each implementation-cost class.

Results. Table 4 reports the selected representatives. The value AS_{128} estimates the number of active S-boxes required to reach characteristic weight 128 using only the single-S-box score:

$$AS_{128}(S) = \frac{128}{w(S)}.$$

The values in the table are truncated to two decimal places.

These representatives are local optima within their cost classes, not the final S-box choice. The next step compares them at the cipher level using the diffusion-layer cost and branch number, the required number of rounds, and the resulting multi-round DCP/LCP bounds.

3.3.4 Filtering by Active-S-box Bounds

The cost-class search in the previous subsection gives one representative S-box for each implementation-cost class, namely five representatives $S_\tau, \tau \in \{4,5,7,8,9\}$. The final S-box cannot yet be selected because the required number of rounds depends on the branch number B of the linear layer, while the total cost also depends on its implementation cost C_M. We therefore filter the five representatives by identifying the S-box classes that can remain optimal for each assumed branch number. The surviving classes are then combined with concrete linear-layer candidates to determine the final S-box.

Filtering Method. For each representative S_τ, let $w_\tau = w(S_\tau)$ be its score. Using only the single-S-box score, the number of active S-boxes required to reach the target characteristic weight $W = 128$ is

$$N_\tau = N_{128}(S_\tau) = \left\lceil \frac{128}{w_\tau} \right\rceil.$$

We treat the branch number of the linear layer as a design parameter: $B \in \{1,\ldots,9\}$. For R rounds, the number of active S-boxes is estimated using

$$A_B(R) = \left\lfloor \frac{R}{2} \right\rfloor B + (R \bmod 2).$$

Table 5: Input values for active-S-box filtering.

S-box cost τ	Score w_τ	Required AS N_τ	Per-round base cost $C_{\text{base}}(\tau)$
4	4.10	32	49
5	5.35	24	57
7	7.92	17	73
8	9.17	14	81
9	10.30	13	89

This gives at least B active S-boxes for every two rounds and one additional active S-box when an odd number of rounds leaves one nonlinear layer unpaired. The minimum number of rounds required for each pair (B, τ) is therefore

$$R_{B,\tau} = \min\{R \mid A_B(R) \geq N_\tau\}.$$

We next estimate the part of the implementation cost that is independent of the concrete linear layer. The nonlinear layer $\mathcal{S}$ evaluates eight 16-bit S-boxes and incurs 8 cycles of packing overhead:

$$C_{\mathcal{S}}(\tau) = 8\tau + 8 = 8(\tau + 1).$$

AddRoundTweakey costs

$$C_{\text{ART}} = 5 + 4 = 9.$$

Thus, the per-round base cost is

$$C_{\text{base}}(\tau) = C_{\mathcal{S}}(\tau) + C_{\text{ART}} = 8(\tau + 1) + 9.$$

The base cost required to reach weight 128 is

$$C^{\text{base}}_{B,\tau} = R_{B,\tau} \cdot C_{\text{base}}(\tau).$$

Once the linear-layer cost is included, the total cost has the form

$$C^{\text{total}}_{B,\tau} = C^{\text{base}}_{B,\tau} + (R_{B,\tau} - 1)C_M + C_{\text{fixed}},$$

where C_{fixed} does not affect comparisons among the S-box classes. The minimum-base-cost candidate is not necessarily optimal because a candidate with a slightly larger base cost may require fewer applications of $\mathcal{M}$.

Therefore, for each B, we discard a candidate τ_1 if there exists another candidate τ_2 such that

$$C^{\text{base}}_{B,\tau_2} \leq C^{\text{base}}_{B,\tau_1} \qquad \text{and} \qquad R_{B,\tau_2} \leq R_{B,\tau_1},$$

with at least one strict inequality. A discarded candidate cannot become optimal for any nonnegative value of C_M.

Results. Table 5 summarizes the input values used for the filtering. Table 6 reports $(R_{B,\tau}, C^{\text{base}}_{B,\tau})$ for each assumed branch number B and S-box class τ. Each entry is written as $R_{B,\tau}/C^{\text{base}}_{B,\tau}$, and bold entries indicate the minimum base cost for each B.

Applying the dominance rule gives the non-dominated candidates in Table 7.

For $B = 1, 2, 6$, two candidates remain because the candidate with the larger base cost requires fewer rounds and may become preferable once C_M is fixed. For the high branch numbers relevant to our design, the result simplifies to

$$B = 7, 8 \Rightarrow \tau = 8, \qquad B = 9 \Rightarrow \tau = 7.$$

The subsequent linear-layer search selects a candidate with branch number $B = 7$. The filtering therefore selects the 8-cycle representative $S_8^\star$, namely the multiplication-based S-box shown in Figure 1.

Table 6: Filtering results for each assumed branch number. Each entry is $R_{B,\tau}/C^{\text{base}}_{B,\tau}$.

B	$\tau = 4$	$\tau = 5$	$\tau = 7$	$\tau = 8$	$\tau = 9$
1	63/3087	47/2679	33/2409	**27/2187**	25/2225
2	32/1568	24/1368	17/1241	**14/1134**	13/1157
3	22/1078	16/912	12/876	10/810	**9/801**
4	16/784	12/684	9/657	8/648	**7/623**
5	14/686	10/570	8/584	**6/486**	6/534
6	12/588	8/456	**6/438**	6/486	5/445
7	10/490	8/456	6/438	**4/324**	4/356
8	8/392	6/342	5/365	**4/324**	4/356
9	8/392	6/342	**4/292**	4/324	4/356

Table 7: Non-dominated S-box classes after active-S-box filtering.

B	Base-cost minimum	Non-dominated candidates
1	$\tau = 8$	$\tau = 8$ $(R = 27, C = 2187)$, $\tau = 9$ $(R = 25, C = 2225)$
2	$\tau = 8$	$\tau = 8$ $(R = 14, C = 1134)$, $\tau = 9$ $(R = 13, C = 1157)$
3	$\tau = 9$	$\tau = 9$ $(R = 9, C = 801)$
4	$\tau = 9$	$\tau = 9$ $(R = 7, C = 623)$
5	$\tau = 8$	$\tau = 8$ $(R = 6, C = 486)$
6	$\tau = 7$	$\tau = 7$ $(R = 6, C = 438)$, $\tau = 9$ $(R = 5, C = 445)$
7	$\tau = 8$	$\tau = 8$ $(R = 4, C = 324)$
8	$\tau = 8$	$\tau = 8$ $(R = 4, C = 324)$
9	$\tau = 7$	$\tau = 7$ $(R = 4, C = 292)$

3.4 Software-Oriented Recursive Linear Layer

The linear layer $\mathcal{M}$ must satisfy two requirements. First, it should provide strong diffusion over the eight 16-bit S-box cells. The active-S-box filtering in the previous subsection indicates that branch numbers around 7 and 8 are the relevant targets. Second, it should operate efficiently on the packed state representation, where the 128-bit state is stored as four 32-bit words.

A dense linear layer over eight 16-bit cells would match the branch-number metric directly. However, it would require extracting, aligning, masking, and repacking the 16-bit cells, introducing additional overhead and register pressure. We therefore construct $\mathcal{M}$ directly on the four packed 32-bit words.

Our construction adapts the principle of recursive diffusion layers [SDMS12, AF14] to a software-oriented setting. Instead of evaluating a dense cell-level matrix, $\mathcal{M}$ applies a sequence of low-cost in-place word updates. Since an updated word can be reused as an input to subsequent updates, this sequence realizes recursive diffusion with a small number of elementary operations.

3.4.1 RotXor Updates as Elementary Linear Maps

Let the state be represented by a tuple of four 32-bit words $X = (X_0, X_1, X_2, X_3)$. The elementary operation used in $\mathcal{M}$ is

$$X_d \leftarrow X_d \oplus (X_s \lll_{32} k), \qquad d \neq s.$$

We call this operation a RotXor update.

RotXor updates are well suited to Cortex-M3 and Cortex-M4. An XOR with a rotated source operand can be implemented as a single word-level instruction, and the update is performed in place, allowing the state to remain in four registers.

Moreover, each RotXor update is self-inverse. Since the source word X_s is unchanged by the update, applying it twice restores the destination word:

$$\big(X_d \oplus (X_s \lll_{32} k)\big) \oplus (X_s \lll_{32} k) = X_d.$$

Thus, each RotXor update is an involution. Consequently, the inverse of a sequence of RotXor updates is obtained by applying the same updates in reverse order, with the same number of operations.

We use rotations rather than shifts because rotations preserve all source bits while changing their positions. Shift-XOR updates are also inexpensive, but discard source bits and are therefore less suitable for diffusing the full 128-bit state.

For search and verification, a RotXor update is viewed as an elementary linear map. Let ρ^k denote the 32-bit rotation by k bits, and let e_i be the i-th unit vector of length four. The update

$$X_d \leftarrow X_d \oplus (X_s \lll_{32} k)$$

corresponds to

$$E_{d,s,k} = I_4 + e_d e_s^\top \rho^k.$$

A sequence of m RotXor updates defines the word-level matrix

$$\mathcal{M} = E_{d_m,s_m,k_m} \cdots E_{d_1,s_1,k_1}.$$

Its coefficients are linear operators on 32-bit words, such as

$$0, \quad 1, \quad \rho^k, \quad \rho^{k_1} \oplus \rho^{k_2} \oplus \cdots .$$

Thus, $\mathcal{M}$ can be interpreted as a recursive 4×4 word-level matrix whose coefficients are rotations and XOR sums of rotations.

Branch Number over 16-bit Cells. Although $\mathcal{M}$ operates on four 32-bit words, its diffusion is evaluated over the eight 16-bit cells used by $\mathcal{S}$. We use the cell weight and branch number B_{16}. Since the state contains eight cells, the maximum possible branch number is $B_{16} = 9$.

The active-S-box filtering in the previous subsection identifies branch numbers 7 and 8 as the main targets. The linear-layer search therefore aims to achieve a high value of B_{16} with as few RotXor updates as possible.

Naive Search Space. A single RotXor update is specified by the destination word d, the source word s, and the rotation amount k. There are four choices for d, three choices for $s \neq d$, and 32 choices for k. Thus, one update has $4 \cdot 3 \cdot 32 = 384$ possibilities, and a naive exhaustive search over m updates has size 384^m. Already for $m = 6$, this gives

$$384^6 \approx 3.2 \times 10^{15},$$

which is too large for a direct exhaustive search without additional restrictions.

Moreover, in our preliminary search over unrestricted sequences of up to six RotXor updates, the best branch number found is only $B_{16} = 4$. This is well below the target values 7 and 8, motivating the structured search space introduced next.

3.4.2 Stage-Restricted Search

To make the search tractable while preserving implementation regularity, we restrict the search to stage-based update sequences. One stage consists of four sequential RotXor updates:

$$X_{d_0} \leftarrow X_{d_0} \oplus (X_{s_0} \lll_{32} k), \; X_{d_1} \leftarrow X_{d_1} \oplus (X_{s_1} \lll_{32} k),$$
$$X_{d_2} \leftarrow X_{d_2} \oplus (X_{s_2} \lll_{32} k), \; X_{d_3} \leftarrow X_{d_3} \oplus (X_{s_3} \lll_{32} k).$$

Each stage satisfies the following constraints:

- each word appears exactly once as a destination;

Table 8: Best branch numbers found in the linear-layer search.

Search space	Number of updates	Number of stages	Best B_{16}
Unconstrained short search	6	–	4
Stage-restricted search	8	2	6
Stage-restricted search	12	3	7

- each word appears exactly once as a source;
- each update satisfies $d_i \neq s_i$;
- all four updates use the same rotation amount k.

In the search, the destination order is fixed to $(0, 1, 2, 3)$. Then the source pattern is a derangement of four elements. We denote the number of derangements of n elements by D_n, also written as $!n$, and use the standard formula

$$D_n = !n = n! \sum_{i=0}^{n} \frac{(-1)^i}{i!}.$$

Hence, there are $!4 = 9$ possibilities. Since the rotation amount has 32 choices, one stage has

$$9 \cdot 32 = 288$$

possibilities, and an r-stage search space has size

$$288^r.$$

For three stages, corresponding to 12 RotXor updates, this gives

$$288^3 = 23{,}887{,}872$$

candidates, which is feasible for exhaustive search.

The stage restriction also gives a balanced recursive structure. Every word is updated once and used once as a source within each stage, so the updates do not concentrate on a small subset of words. Since the updates are applied sequentially, a word updated early in a stage can be reused as a source later in the same stage, producing recursive diffusion beyond a parallel sparse matrix.

Search Results. We exhaustively search stage-restricted candidates with up to three stages, corresponding to at most 12 RotXor updates. Table 8 summarizes the best branch numbers.

The three-stage search finds 12-update candidates with $B_{16} = 7$. No candidate with $B_{16} = 8$ or 9 is found in the stage-restricted search with up to 12 updates.

We next combine this result with the active-S-box filtering from Section 3.3.4. For the 12-update candidate with $B_{16} = 7$, the relevant parameters are

$$B = 7, \qquad C_M = 12, \qquad \tau = 8, \qquad R = 4.$$

For $\tau = 8$, the per-round base cost is $C_{\text{base}}(8) = 81$. Since the final round omits the linear layer $\mathcal{M}$, the total estimated cost is

$$C^{\text{total}} = 4 \cdot 81 + 3 \cdot 12 = 360.$$

This result excludes all branch numbers $B \leq 6$ from cycle minimization under our cost model. The minimum base costs obtained from the active-S-box filtering are

B	1	2	3	4	5	6
$\min_\tau C^{\text{base}}_{B,\tau}$	2187	1134	801	623	486	438.

All these values exceed 360 even before adding the linear-layer cost.

For $B = 8$, the filtering selects $\tau = 8$ and $R = 4$, giving the same base cost $4 \cdot 81 = 324$. A $B_{16} = 8$ candidate with linear-layer cost C_M would therefore cost

$$324 + 3C_M.$$

To outperform the selected $B_{16} = 7$ class, it must satisfy

$$324 + 3C_M < 360,$$

or equivalently $C_M < 12$. Equality requires $C_M = 12$. Since no $B_{16} = 8$ candidate is found with up to 12 updates, no such candidate in the searched space can match or improve the selected cost.

No $B_{16} = 9$ candidate is found in the same search space. We do not claim non-existence outside this space. Consequently, among the candidates obtained by our search, the 12-update class with $B_{16} = 7$ gives the lowest estimated cipher-level cost. We next select a concrete representative from this class.

Representative Selection. The three-stage search finds 9,700 candidates with 12 updates and $B_{16} = 7$. All these candidates have the same primary diffusion metric and implementation cost, so we use the stage rotation amounts as a secondary implementation-oriented criterion.

Although all rotation amounts have the same cost in our Cortex-M3/M4 cycle model, a zero rotation avoids a nontrivial rotation, while byte-aligned rotations provide a regular structure useful for portability. We first maximize the number of stages with rotation amount zero. Among the $B_{16} = 7$ candidates, the maximum is one. We then require one additional nonzero byte-aligned rotation, $k \in 8, 16, 24$. This leaves 176 candidates with one zero rotation, one nonzero byte-aligned rotation, and one unrestricted rotation.

No further security or performance criterion distinguishes these candidates in our evaluation. We therefore select a representative with stage rotation amounts

$$(24, 0, 13).$$

Accordingly, we adopt the 12-update linear layer with $B_{16} = 7$ shown in Figure 2 as the final linear layer $\mathcal{M}$ of Rotom. Its exact instruction sequence is specified in Section 2.

3.5 Tweakey Schedule

The tweakey schedule [JNP14] supports the fixed-key related-tweak setting without increasing the number of data-processing rounds. If the tweak state were updated only linearly, a tweak difference could propagate through the schedule without activating a nonlinear component. Rotom therefore updates the tweak state using the same layers S and M as the data-processing part. Consequently, tweak differences are charged to active S-boxes, allowing the same local DCP bounds to be used in the related-tweak analysis. For a fixed tweak, the round tweakeys are constants and the single-tweak analysis is unchanged.

For the 256-bit-key variants, the two halves $K^{[0]}$ and $K^{[1]}$ of $K = K^{[0]} \parallel K^{[1]}$ are used alternately, as specified in Section 2. This requires only switching the referenced key half and prevents the same partial-key information from being directly reused across adjacent rounds.

Finally, reusing S and M in both key-size variants keeps the implementation compact, since no separate nonlinear tweak-update primitive is required.

3.6 Modular-Addition Whitening and Boundary Key Hardening

The whitening part surrounds the data-processing part at the input and output boundaries. At each boundary, word-wise modular key addition is combined with the adjacent linear layer $\mathcal{M}$.

The main purpose of the modular-addition boundary is to make key-recovery extensions from reduced-round distinguishers more difficult. With XOR whitening, the first linear layer could be absorbed into an equivalent XOR subkey:

$$\mathcal{M}(P \oplus K) \oplus TK[0] = \mathcal{M}(P) \oplus \big(\mathcal{M}(K) \oplus TK[0]\big).$$

Word-wise modular addition makes this absorption more difficult because the carries depend jointly on the plaintext word and the key word. In particular, determining an output bit of a word addition generally requires information about lower plaintext and key bits through the carry chain. The adjacent linear layer M then spreads these word-local, carry-dependent terms across the state before the first nonlinear layer S.

In this sense, the whitening aims to play an FX/DESX-like boundary-hardening role. It places key-dependent operations outside the data-processing part to make it more difficult for an attacker to extend a reduced-round distinguisher merely by peeling off an equivalent XOR subkey.

A direct alternative would be to add another nonlinear layer at each boundary. This would require eight additional 16-bit S-box evaluations per boundary, or sixteen evaluations in total, together with the corresponding unpacking and packing overhead. By contrast, the modular-addition part requires four 32-bit additions per boundary during encryption and four 32-bit subtractions per boundary during decryption.

This choice is also motivated by recent work on modular-addition FX constructions, where word-wise modular-addition whitening is shown to strengthen resistance to integral attacks by accelerating key-dependent algebraic propagation [BHLP25]. Our setting is not identical to that framework, because the whitening values and the round tweakeys are derived from the same master key and the whitening values are mutually related. We therefore use this result as design motivation rather than as a direct security proof. The concrete impact of the boundary construction is analyzed in Section 4.3.

128-Bit-Key Variants. For the 128-bit-key variants, the state entering the first nonlinear layer is

$$\mathcal{M}\big(P \boxplus_{32\times 4} K\big) \oplus TK[0],$$

where $\boxplus_{32\times 4}$ denotes componentwise addition modulo 2^{32} over four 32-bit words. If Y denotes the output of the data-processing part, the output boundary computes

$$C = \mathcal{M}(Y) \boxplus_{32\times 4} \big(K \lll_{128} 64\big).$$

Thus, the same master key is used under different 32-bit-word alignments at the two boundaries, avoiding repeated use of the identical word-wise whitening relation. Since the 64-bit rotation only permutes the four 32-bit words, it can be implemented by register renaming without additional instructions.

256-Bit-Key Variants. Let the 256-bit master key be divided into two 128-bit halves $K^{[0]}$ and $K^{[1]}$. A straightforward extension would repeat the 128-bit-key whitening structure twice, using the two key halves separately. However, the two halves would then be injected at different depths relative to the adjacent applications of $\mathcal{M}$, resulting in asymmetric treatment.

Instead, every whitening value combines both halves by word-wise modular addition. As specified in Section 2.3, $W_0, \ldots, W_3$ use the four combinations obtained by using $K^{[0]}$

either directly or after a 32-bit rotation and $K^{[1]}$ either directly or after a 64-bit rotation. Consequently, both halves occur at every whitening position, while the rotations change their word-wise pairing across the four injections. These rotations only permute the four 32-bit words and therefore require no additional instructions. The values $W_0, \ldots, W_3$ remain related functions of the same master key and are not regarded as independent whitening keys.

Side-channel aspect. The whitening uses a 32-bit-wise arithmetic addition, differently from the round key addition realized by bit-wise XOR. Side-channel attacks on ciphers with whitening (e.g., PRINCE) typically estimate the sum of whitening and first-round keys [YMUM+21]. However, such a strategy is inapplicable to Rotom, as the whitening uses different addition from round key addition and diffuses the key at the subsequent linear layer. This implies that side-channel attack must recover the whitening key using the leakage of arithmetic addition before recovering the round key. This is actually feasible as evaluated by simulation in Section 5.4.2. However, as a countermeasure, the whitening naturally employs an arithmetic masking, which would be more secure than Boolean masking because of the relationship between its algebraic structure and Hamming weight leakage [UIM+25]. Thus, thanks to the whitening by arithmetic addition, masked Rotom would achieve higher side-channel security by adoption of arithmetic masking, when the leakage follows a Hamming weight (HW) model.

4 Security Analysis

In this section, we analyze the security of Rotom against a wide range of cryptanalytic techniques, including differential and linear cryptanalysis, impossible differential attacks, and integral attacks. We also discuss how many rounds can be covered by key-recovery attacks obtained by extending the corresponding distinguishers.

4.1 Differential and Linear Cryptanalysis

This section evaluates differential and linear cryptanalysis at the single-characteristic level. The analysis has three steps. First, we derive characteristic bounds by combining the local DDT/LAT properties of S_{16} with lower bounds on the number of active S-boxes. Second, we enumerate concrete differences and masks for the minimum-active cases and report the best characteristics found in this evaluation. Finally, we compare these results with the security thresholds of the four variants.

Characteristic bounds. The nonlinear layer $\mathcal{S}$ consists of eight parallel applications of the 16-bit S-box S_{16}. We computed the full differential distribution table (DDT) and linear approximation table (LAT) of S_{16}. Let p_S be the maximum differential probability and q_S be the maximum squared correlation. The maximum DDT entry is 84, and the maximum absolute LAT entry is 2728. Thus,

$$p_S = 84 \cdot 2^{-16} \approx 2^{-9.61}, \qquad q_S = (2728 \cdot 2^{-16})^2 \approx 2^{-9.17}.$$

We combine these local bounds with lower bounds on the number of active S-boxes. The active-S-box bounds are obtained with a bit-level model of the round function, where S_{16} is encoded as a circuit. For related-tweak differentials, active S-boxes are counted both in the data processing part and in the tweakey schedule. For linear characteristics, the model excludes local transitions with zero LAT entry. Table 9 reports the resulting bounds. The same bounds apply to the 256-bit-key variants. Under a fixed master key, the alternating key halves act only as round-dependent constants and therefore do not affect difference or mask propagation.

Table 9: Minimum number of active S-boxes. Here, SK, TK, and SK Lin denote single-tweak differentials, related-tweak differentials, and single-tweak linear characteristics, respectively.

Model	1	2	3	4	5	6
SK	1	7	9	14	17	21
TK	1	7	9	14	17	21
SK Lin	1	7	9	14	17	21

A differential characteristic with A active S-boxes has probability at most p_S^A, and a linear characteristic with A active S-boxes has squared correlation at most q_S^A. These give conservative characteristic-level bounds.

Using the active-S-box bounds in Table 9, the single- and related-tweak differential weights for one to six rounds are

$$9.61,\ 67.25,\ 86.47,\ 134.51,\ 163.33,\ 201.76.$$

The corresponding single-tweak linear weights are

$$9.17,\ 64.21,\ 82.55,\ 128.42,\ 155.94,\ 192.63.$$

In particular, three rounds already exceed the 80-bit limit, while four rounds exceed the 112-bit time threshold and the bounds imposed by a 2^{128} data limit.

Concrete Characteristic Search. The active-S-box bounds use only worst-case local DDT/LAT values. To check whether such worst-case local transitions can actually be combined through the linear layer, we also enumerate concrete characteristics for the minimum-active cases.

For differential characteristics, we enumerate the word-level activity patterns with the minimum number of active S-boxes and then enumerate concrete 16-bit differences satisfying the linear-layer constraints. For each candidate, we check whether all active S-box transitions have nonzero DDT entries and compute the product of their DDT probabilities. The largest differential characteristic probability over 3 rounds found in this evaluation is

$$p \approx 2^{-124.51}.$$

For linear characteristics, we propagate masks through the transpose linear layer and enumerate concrete 16-bit masks satisfying the corresponding constraints. For each candidate, we check whether all active S-box transitions have nonzero LAT entries and compute the product of their squared correlations. The largest squared correlation over 3 rounds found in this evaluation is

$$c^2 \approx 2^{-122.84}.$$

These concrete searches show that the minimum-active patterns do not directly realize the worst-case local DDT/LAT bounds.

Security Implication. The active-S-box bounds are conservative because they use the worst-case local DDT/LAT values independently at every active S-box. The concrete three-round searches show a large gap from these bounds: with nine active S-boxes, the best differential and linear characteristics found have weights 124.51 and 122.84, compared with the conservative weights 86.47 and 82.55. This indicates that worst-case local transitions do not align through the linear layer in the minimum-active cases.

Even under the conservative bounds, three rounds exceed the 80-bit limit of Rotom-128* and Rotom-256*. Four rounds exceed the 112-bit time threshold of Rotom-128 and the bounds imposed by the 2^{128} data limit of Rotom-256. Consequently, Rotom-128* and Rotom-128 retain a three-round characteristic-level margin, while Rotom-256* and Rotom-256 retain a four-round margin. We therefore conclude that all four variants have sufficient characteristic-level margin against differential and linear attacks.

4.2 Summary of Other Attacks

Due to space limitations, we give only a summary of our evaluation against other standard cryptanalytic techniques. Detailed analyses are provided in Supplementary Material C.

We evaluate boomerang and differential-linear attacks, multiplicative differential attacks, algebraic and integral attacks, impossible differential attacks, rotational and slide attacks, and meet-in-the-middle attacks. For boomerang and differential-linear attacks [Wag99, LH94, BDK02], the reduced-round differential and linear bounds from Section 4.1 provide sufficient margin for all relevant splits. In particular, the bounds cover the 5-round Rotom-128* and the 6-round Rotom-256* under the 2^{80} data limit, as well as the 6-round Rotom-128 and the 7-round Rotom-256 under their corresponding time- and data-complexity claims.

For multiplicative differential and rotational attacks [BCJW02, Fur01, MNSV97, Scr01, KN10], we explicitly evaluate the relevant local relations of S_{16}. Although some relations hold with non-negligible probability for a single cell, they are not preserved by the internal linear maps, the linear layer $\mathcal{M}$, or the tweakey schedule. The round constants and nonlinear tweakey update also break simple slide symmetries [BW99].

For algebraic attacks [BC13, CP02] and integral attacks [KW02, Tod15, BC16], the high algebraic degree of S_{16} and the rapid growth of the degree upper bound in the data-processing part make low-degree structures difficult to maintain. Moreover, the modular-addition key mixing at the input and output boundaries introduces carry-dependent key terms, strengthening resistance against integral properties and making key-recovery extensions harder.

For impossible differential attacks [BBS99, KHL10] and meet-in-the-middle attacks [AS09, SA09], the 1.5-round full-diffusion property limits standard structural cores to at most three and five rounds, respectively, within the active-S-box-based analysis and the standard MitM framework [AS09].

In addition, the modular-addition boundary key mixing and the dependencies introduced by the tweakey schedule make it difficult to attach outer rounds to such cores for practical key recovery.

Overall, we do not identify an attack that threatens any of the four variants. Together with the differential and linear analysis in Section 4.1, these evaluations support their claimed security margins.

4.3 Difficulty of Key-Recovery Extensions

We finally discuss the difficulty of extending reduced-round distinguishers to key-recovery attacks. In our search, the best differential and linear distinguishers we found cover three rounds of the data processing part. A standard way to turn such distinguishers into key-recovery attacks is to add outer rounds and partially encrypt or decrypt them with guessed key material. In our construction, however, this extension becomes difficult once the whitening part is included.

The input and output whitening use word-wise modular key addition, and the adjacent applications of the linear layer $\mathcal{M}$ immediately mix the resulting carry-dependent key terms across the state. Consequently, the attacker cannot isolate a small independent subset of key bits for the outer rounds. Instead, even a short extension around the distinguisher

involves full 32-bit whitening-key words together with the corresponding round tweakey material.

Moreover, the same master key is used in the tweakey schedule, so the whitening-key guesses are not independent of the round tweakeys appearing inside the distinguisher. The nonlinear update of the tweakey state further restricts the possible order of key guesses. Therefore, although 3-round differential and linear distinguishers exist, attaching practical key-recovery rounds to them is difficult once the whitening part is included. We do not identify a key-recovery attack on the proposed 5-round ,6-round and 7-round variants that improves over exhaustive search. More detailed analysis is given in the remainder of this subsection.

4.3.1 Estimated Number of Rounds Covered by Key-Recovery Attacks

We focus on differential and linear attacks in the single-key setting, and estimate the maximum number of rounds that might be covered after adding a key-recovery phase. Let $\tau = \kappa - \varepsilon$ denote the exponent of the claimed time complexity; thus, $\tau = 112$ for $\kappa = 128$ and $\tau = 240$ for $\kappa = 256$. Let $D_{\max}$ denote the maximum allowed data complexity. Let $w_D(r)$ and $w_L(r)$ denote the weights of an r-round differential characteristic and an r-round linear characteristic, respectively. Here, the linear weight is defined using squared correlation.

The required number of data differs between linear and differential attacks. A linear characteristic of weight $w_L(r)$ requires approximately $2^{w_L(r)}$ data. We therefore define

$$B_L := \min\{\tau, \log_2 D_{\max}\},$$

where B_L is the maximum linear weight that is not excluded by the limits of time and data complexities.

A differential characteristic of weight $w_D(r)$ requires approximately $2^{w_D(r)}$ pairs. Among $D_{\max}$ data, at most $\lfloor D_{\max}/2 \rfloor$ pairs can satisfy one exact difference at the input of the distinguisher under the correct key. Therefore, we define

$$B_D := \min\left\{\tau, \log_2 \left\lfloor \frac{D_{\max}}{2} \right\rfloor\right\},$$

where B_D is the maximum differential weight that is not excluded by the limits of time and data complexities.

Let

$$r_L := \max\{r : w_L(r) < B_L\}, \qquad r_D := \max\{r : w_D(r) < B_D\},$$

and define

$$r_{\text{dist}} := \max\{r_L, r_D\},$$

where, r_{dist} denotes the maximum number of rounds covered by an evaluated differential or linear characteristic that is not excluded by the corresponding security claim.

The weights of the differential and linear characteristics over 2, 3, and 4 rounds derived from the number of active S-boxes are

$$(w_D(r), w_L(r)) = (67.25, 64.21), \quad (86.47, 82.55), \quad (134.51, 128.42),$$

respectively.

For Rotom-128* and Rotom-256*, $D_{\max} = 2^{80}$ gives

$$B_D = 79, \qquad B_L = 80,$$

and therefore

$$r_{\mathrm{dist}} = 2.$$

For Rotom-128, $\tau = 112$ and $D_{max} = 2^{128}$ give

$$B_D = B_L = 112,$$

and hence

$$r_{dist} = 3.$$

For Rotom-256, $\tau = 240$ and $D_{max} = 2^{128}$ give

$$B_D = 127, B_L = 128.$$

Since the 4-round differential and linear weights are 134.51 and 128.42, respectively, we again obtain

$$r_{dist} = 3.$$

We emphasize that the concrete 3-round searches give larger weights, 124.51 and 122.84. Therefore, using the smaller active-S-box-based weights in the above estimate is conservative.

A structure of D plaintexts contains approximately $D^2/2$ candidate pairs. This does not give $D^2/2$ right pairs for one fixed input difference under the correct key. However, multiple-differential or truncated-differential attacks may exploit pairs corresponding to several differences. Such aggregation is outside the single-characteristic-based estimation above. In our estimation, we employ conservative bound. Therefore, we expect that these attacks cannot become much more powerful compared to the differential attacks based on the single characteristic in our estimation.

Adding Key-Recovery Rounds. For the 128-bit-key variants, exact output-side partial decryption starts with

$$Y = \mathcal{M}^{-1}(C \boxminus_{32\times 4} (K \lll_{128} 64)),$$

where Y denotes the output of the data-processing part. Thus, an output-side extension must be evaluated together with the word-wise modular subtraction and the adjacent linear layer.

For the 256-bit-key variants, the output-side inverse computation is

$$Y = \mathcal{M}^{-1}\left(\mathcal{M}^{-1}(C \boxminus_{32\times 4} W_3) \boxminus_{32\times 4} W_2\right).$$

The corresponding input-side extension must account for W_0, W_1, and the two adjacent applications of $\mathcal{M}$.

The linear layer $\mathcal{M}$ adjacent to the whitening operation mixes the four 32-bit words. In addition, the word-wise modular key addition introduces data-dependent carries within each word, so multiple whitening-key bits jointly affect the state entering or leaving the data-processing part. Consequently, the attacker cannot directly apply the standard outer-round extension based on guessing a small independent S-box subkey. Instead, the whitening-key value and the corresponding round tweakeys must be considered jointly. Under a straightforward exact partial-encryption or partial-decryption approach, this is expected to require guessing a substantially larger portion of the whitening-key value than in a conventional S-box-based outer-round extension.

Table 10: Estimated number of attacked rounds for differential and linear attacks.

Variant	$D_{\max}/2^{\tau}$	r_{dist}	Security margin
Rotom-128*	$2^{80}/2^{112}$	2	3
Rotom-128	$2^{128}/2^{112}$	3	3
Rotom-256*	$2^{80}/2^{240}$	2	4
Rotom-256	$2^{128}/2^{240}$	3	4

Under a straightforward partial-encryption or partial-decryption approach, computing the whitening part is expected to require guessing large and correlated parts of the whitening-key bits and round-tweakey bits. This cost could consume a substantial part of the claimed time complexity. In particular, as mentioned above, whitening mixes all words through modular addition and $\mathcal{M}$. Consequently, an attacker needs to guess most of the whitening key, leaving an insufficient time complexity to extend the key-recovery rounds. Hence, in this evaluation model, we do not automatically grant the attacker additional key-recovery rounds[2]. Under this model, the total number of attacked rounds is equal to the number of rounds covered by the distinguisher. Table 10 summarizes the resulting estimations.

More Conservative Evaluation. We also consider a more conservative model in which one key-recovery round is appended at each end of the distinguisher, namely, on the encryption and decryption sides, without accounting for any additional cost. This situation could arise if the whitening-key bits that need to be guessed coincide with the partial-key bits required for the additional key-recovery rounds. However, the whitening procedure applies modular addition and rotation to the key. These operations make such a coincidence less likely and generally require the attacker to guess additional partial-key bits. We therefore regard this model as a conservative evaluation.

Under this evaluation, 2 key-recovery rounds are added to the overall attack. Consequently, the estimated security margin of each variant is reduced by 2 rounds. We emphasize that this is a conservative estimation and does not imply the existence of concrete attacks covering these additional rounds.

5 Implementation Evaluation

This section evaluates the performance and side-channel aspects of Rotom implementations on 32-bit ARM Cortex-M microcontrollers. We first describe the evaluation setting and implementation techniques, then present performance comparisons, and finally discuss timing and power/EM side-channel attacks and countermeasures.

5.1 Evaluation Setting

Benchmark Methodology. We evaluate execution cycles per 128-bit block, ROM code size, and RAM usage. Since Rotom is a TBC, we measure the tweakey schedule, encryption, and decryption separately. The encryption and decryption measurements assume precomputed round tweakeys, corresponding to the precomputed round-key setting used for ordinary block ciphers. The separate tweakey-schedule measurement also allows us to estimate the cost of on-the-fly round-tweakey generation.

We use the FELICS benchmark framework for the top-level interface [DCK+15, DCK+19]. This methodology follows previous microcontroller evaluations [ANP20, AP21,

[2]We also consider a more conservative model in which the attacker is granted one outer key-recovery round at each end without accounting for the cost of guessing its partial keys.

MSR+25, BBdS+20].

Comparison Targets. We mainly compare Rotom with Speck-128/128 and Speck-128/256, which are highly efficient on microcontrollers, and with MSX-128-128 and MSX-128-256, which are microcontroller-oriented multiplication-based block ciphers.

As a broader reference, we also report fixslicing-based constant-time implementations of AES-128, AES-256, GIFT-128, SKINNY-128-128 and SKINNY-128-256 on ARM Cortex-M processors. In addition, we compare Rotom-128 with mid-pSquare [BGM+25], a TBC for 32-bit microcontrollers.

Evaluation Environment. The proposed scheme Rotom is implemented in handwritten assembly optimized for the target platforms. Speck and MSX are implemented in C, and both are measured together with Rotom on our target platforms. The C source code for Speck and MSX is taken from the official MSX repository [mra]. The Speck implementation is derived from the official repository [Nat21] used in SUPERCOP [BL18], with minor modifications to match the byte-oriented input format used by MSX and Rotom. For the ARM environment, we use STM32CubeIDE[3] as the development environment. For C implementations, we use `arm-none-eabi-gcc` (GNU Tools for STM32 13.3.rel1, GCC 13.3.1) with the optimization options `-O2` and `-O3`.

The results for the other comparison targets are taken from the corresponding papers rather than measured in our evaluation environment. The fixslicing-based implementations are written in handwritten assembly [ANP20, AP21], whereas mid-pSquare is evaluated with a constant-time C implementation on Cortex-M4. The mid-pSquare implementation is compiled with Arm Compiler 6 using `-O3`, and its results include the key and tweak schedules.

Fully-unrolled and Non-unrolled Implementations. To evaluate both peak performance and the performance achievable with a practical ROM footprint, we prepare two implementation profiles for each target platform. A similar strategy is used in previous work [AP21].

The first is a *fully-unrolled implementation*, where the whole computation is unrolled to minimize execution cycles by removing loop overhead and branches. This profile is used for speed-oriented comparison with previous work.

The second is a *non-unrolled implementation*, where repeated groups of one or more rounds are executed inside a loop, depending on the variant and operation. This profile represents an implementation with a practical ROM footprint and is used to evaluate how much performance is retained after reducing the code size.

Remarks. These comparisons are not under identical functionality and implementation conditions. Most comparison targets are not TBCs. Moreover, Speck , MSX and mid-pSquare are implemented in C, whereas Rotom and the fixslicing-based implementations use handwritten assembly. Some reported AES and SKINNY results also process two blocks in parallel. Therefore, the comparison is intended to show the performance range of Rotom on 32-bit ARM Cortex-M microcontrollers, rather than to claim a strictly equivalent comparison.

5.2 Implementation Techniques

This section describes the implementation of Rotom on ARM Cortex-M3 and Cortex-M4, focusing on the packed-state handling required by Nonlinear Layer $\mathcal{S}$. The 128-bit state is stored in four 32-bit registers, each containing two 16-bit cells. This representation allows

[3] https://www.st.com/en/development-tools/stm32cubeide.html

MatrixLayer, whitening, and AddRoundTweakey to operate directly on 32-bit words. In contrast, Nonlinear Layer $\mathcal{S}$ applies S_{16} independently to each 16-bit cell, so each 32-bit word must be unpacked into two halfwords and repacked after the two S-box evaluations.

The unpacking and repacking cost is included in the design-stage cost model used in the S-box search. However, a direct C implementation does not necessarily generate the instruction sequence assumed by this model. We therefore use handwritten assembly for Cortex-M3 and Cortex-M4, both to match the design-stage cost model on real devices and to measure the cost of each operation accurately.

We first describe the Cortex-M3 implementation as the baseline, and then explain the additional Cortex-M4 optimizations.

Cortex-M3 implementation. In the Cortex-M3 implementation, each 32-bit word $W = [x_1 \mid x_0]$ contains two 16-bit cells. The lower cell x_0 is used directly, while the upper cell x_1 is extracted by `LSR #16`. The same S_{16} macro is then applied to both cells, and the result $S_{16}(x_1)$ is inserted into the upper halfword by `BFI`. Thus, one word is transformed as

$$[x_1 \mid x_0] \longmapsto [S_{16}(x_1) \mid S_{16}(x_0)].$$

This unpack/repack step costs two instructions, `LSR` and `BFI`, per 32-bit word, or eight instructions for the full Nonlinear Layer $\mathcal{S}$.

The S_{16} macro requires two implementation details. First, before each shift, the result of the multiplication is truncated to 16 bits by `UXTH`. This prevents upper-halfword bits from affecting the subsequent shift-XOR map

$$x \mapsto x \oplus (x \gg 8),$$

which is implemented using a 32-bit shift instruction.

Second, for the rotation-XOR map

$$x \mapsto x \oplus (x \lll_{16} 4) \oplus (x \lll_{16} 12),$$

we duplicate the 16-bit value into both halfwords, forming $[x \mid x]$. Then a 32-bit `ROR` realizes the corresponding 16-bit rotations without additional masking. This implementation matches the design-stage cost model: the cost of each 16-bit S-box evaluation and the unpack/repack overhead of the packed state are both reflected in the measured Cortex-M3 code.

Cortex-M4 implementation. The Cortex-M4 implementation uses the same packed 32-bit state representation as the Cortex-M3 implementation, but exploits halfword-oriented instructions to optimize Nonlinear Layer $\mathcal{S}$. Instead of fully separating the two halfwords and applying the scalar S_{16} macro twice, the multiplication and shift-XOR steps are partly processed in packed form.

For multiplication, `SMULBB` and `SMULTB` multiply the lower and upper halfwords of a packed word by the constant, respectively. Although these instructions perform signed halfword multiplication, only the lower 16 bits of the product are used. Thus, they give the same result as multiplication modulo 2^{16}. The two 16-bit products are then packed into one 32-bit word by `PKHBT`, giving the conceptual transformation

$$[x_1 \mid x_0] \longmapsto [cx_1 \mid cx_0].$$

For the shift-XOR map, `UXTB16` combined with `ROR #8` extracts the upper byte of each halfword in parallel. Therefore, the map

$$x \mapsto x \oplus (x \gg 8)$$

can be applied to both halfwords with one `EOR`. For the rotation-XOR map, we still duplicate each 16-bit value into $[x \mid x]$ and use a 32-bit `ROR`, as in the Cortex-M3 implementation, to avoid mixing bits across halfword boundaries.

With these Cortex-M4-specific instructions, the two S_{16} applications inside one 32-bit word are processed in 16 instructions. Over the full Nonlinear Layer $\mathcal{S}$, this reduces the cost by 8 cycles compared with the Cortex-M3 implementation. This optimization is not included in the S-box search cost model, which uses Cortex-M3 as the baseline. Thus, the Cortex-M4 results reflect both the design-stage cost model and the additional speedup obtained from M4 halfword-oriented instructions.

5.3 Results

The comparison results for Rotom, Speck, and MSX are shown in Table 11. As a reference comparison with representative S-box-based 128-bit block ciphers, Table 12 also shows the values for fixslicing-based implementations of AES-128, AES-256, GIFT-128, and SKINNY-128, for which fast constant-time implementations on ARM Cortex-M processors have been reported. In addition, we provide a comparison with mid-pSquare.

Furthermore, we build a simple AE based on Rotom-128, using ΘCB [KR11] mode, dubbed Rotom-128-ΘCB3, and compare it with Ascon-AEAD128 [TMC+25] and ChaCha20-Poly1305 [NL18]. Our benchmark results on Cortex-M3/M4 show that Rotom-128-ΘCB3 is about twice as fast as ChaCha20-Poly1305 and is comparable to Ascon-AEAD128. The details are given in Supplementary Material D.

Cycle Counts. Table 11 shows that Rotom is substantially faster than MSX and competitive with Speck. For the 128-bit-key variants, fully unrolled encryption of Rotom-128 takes 651 cycles on Cortex-M3 and 603 cycles on Cortex-M4, compared with 1034 and 973 cycles for MSX-128-128. It is comparable to Speck-128/128, and is faster on Cortex-M4. The reduced-round variant Rotom-128* gives the smallest encryption and decryption cycle counts among the 128-bit-key variants.

For the 256-bit-key variants, Rotom-256* and Rotom-256 are faster than MSX-128-256. Compared with Speck-128/256, Rotom-256* is slightly slower on both Cortex-M3 and Cortex-M4, while Rotom-256 has a larger overhead because it uses one additional round and a heavier whitening part. The tweakey schedule is also cheaper than MSX-128-128 for the 128-bit-key variants. For the 256-bit-key variants, the tweakey schedule must handle both 128-bit key halves. In the fully unrolled implementations, Rotom-256* remains faster than MSX-128-256, whereas Rotom-256 is moderately slower because it uses one additional round. In the non-unrolled implementations, however, the performance degradation of Rotom is small, while MSX incurs a substantially larger overhead; consequently, both Rotom-256* and Rotom-256 are faster than MSX-128-256.

Table 12 shows that Rotom is also faster than representative fixslicing-based implementations. For example, Rotom-128 is about 1.96 times faster than fully fixsliced unrolled AES-128 on Cortex-M3 and 2.11 times faster on Cortex-M4. For Rotom-128*, the speedups are about 2.28 and 2.46 times, respectively.

RAM Usage. The stack usage of Rotom encryption and decryption is approximately 40 bytes for all variants. The additional buffer required in the precomputed round-tweakey setting is also small. For Rotom, the parenthesized I/O values include both the precomputed round-tweakey buffer and the master key used in the whitening part. For the 128-bit-key variants, Rotom-128* requires 16(+112) bytes and Rotom-128 requires 16(+128) bytes. These values are smaller than those of Speck-128/128, MSX-128-128, and SKINNY-128-128 in Tables 11 and 12. For the 256-bit-key variants, Rotom-256* requires 16(+144) bytes and Rotom-256 requires 16(+160) bytes. These values are also smaller than those of the

Table 11: Implementation results of Rotom, Speck, and MSX on ARM Cortex-M microcontrollers. Time is given in cycles per 128-bit block, and ROM and RAM are given in bytes. RAM is divided into I/O buffer and Stack. The values in parentheses in the I/O column for encryption and decryption denote additional inputs such as precomputed round keys, round tweakeys, or key inputs.

		Cortex-M3				Cortex-M4			
Cipher	Operation	Time	ROM	RAM		Time	ROM	RAM	
				I/O	Stack			I/O	Stack
Fully-unrolled implementation									
128-bit-key variants									
Rotom-128*	Tweakey sched.	530	1819	128	36	490	1699	128	36
	Encryption	559	1885	16(+112)	40	519	1765	16(+112)	40
	Decryption	568	1889	16(+112)	40	528	1769	16(+112)	40
Rotom-128	Tweakey sched.	625	2175	144	36	577	2031	144	36
	Encryption	651	2233	16(+128)	40	603	2089	16(+128)	40
	Decryption	663	2237	16(+128)	40	615	2093	16(+128)	40
Speck-128/128	Key sched.	530	1479	272	16	523	1387	272	12
	Encryption	635	1795	16(+256)	28	623	1811	16(+256)	12
	Decryption	648	1959	16(+256)	32	631	1843	16(+256)	12
MSX-128-128	Key sched.	706	1576	880	100	660	1488	880	104
	Encryption	1034	2620	16(+864)	44	973	2596	16(+864)	36
	Decryption	1043	2612	16(+864)	44	962	2560	16(+864)	36
256-bit-key variants									
Rotom-256*	Tweakey sched.	677	2247	160	40	629	2103	160	40
	Encryption	725	2461	16(+144)	40	677	2317	16(+144)	40
	Decryption	745	2469	16(+144)	40	697	2325	16(+144)	40
Rotom-256	Tweakey sched.	782	2615	176	40	726	2447	176	40
	Encryption	818	2809	16(+160)	40	762	2641	16(+160)	40
	Decryption	839	2817	16(+160)	40	783	2649	16(+160)	40
Speck-128/256	Key sched.	578	1711	304	32	569	1623	304	28
	Encryption	671	1907	16(+272)	28	659	1923	16(+272)	12
	Decryption	684	2079	16(+272)	32	667	1955	16(+272)	12
MSX-128-256	Key sched.	714	1668	896	108	678	1620	896	108
	Encryption	1034	2620	16(+864)	44	973	2596	16(+864)	36
	Decryption	1043	2612	16(+864)	44	966	2560	16(+864)	36
Non-unrolled implementation									
128-bit-key variants									
Rotom-128*	Tweakey sched.	558	451	128	36	523	427	128	36
	Encryption	591	557	16(+112)	40	548	525	16(+112)	40
	Decryption	598	565	16(+112)	40	555	533	16(+112)	40
Rotom-128	Tweakey sched.	659	451	144	36	617	427	144	36
	Encryption	686	549	16(+128)	40	638	525	16(+128)	40
	Decryption	695	557	16(+128)	40	647	533	16(+128)	40
Speck-128/128	Key sched.	655	511	272	28	654	511	272	28
	Encryption	645	519	16(+256)	24	640	515	16(+256)	24
	Decryption	675	555	16(+256)	32	651	563	16(+256)	24
MSX-128-128	Key sched.	1606	1612	880	212	1457	1524	880	208
	Encryption	1088	616	16(+864)	56	1037	624	16(+864)	56
	Decryption	1123	660	16(+864)	56	1073	664	16(+864)	56
256-bit-key variants									
Rotom-256*	Tweakey sched.	723	479	160	40	675	455	160	40
	Encryption	789	703	16(+144)	40	741	679	16(+144)	40
	Decryption	807	715	16(+144)	40	759	691	16(+144)	40
Rotom-256	Tweakey sched.	836	483	176	40	780	459	176	40
	Encryption	888	703	16(+160)	40	832	679	16(+160)	40
	Decryption	908	715	16(+160)	40	852	691	16(+160)	40
Speck-128/256	Key sched.	716	639	304	36	712	639	304	36
	Encryption	691	559	16(+272)	28	684	555	16(+272)	28
	Decryption	697	615	16(+272)	32	700	599	16(+272)	20
MSX-128-256	Key sched.	1614	1621	896	212	1467	1552	896	208
	Encryption	1088	624	16(+864)	56	1035	624	16(+864)	56
	Decryption	1124	664	16(+864)	56	1073	664	16(+864)	56

Table 12: Comparison with existing fixslicing-based S-box-based 128-bit block ciphers, where SFS denotes semi-fixsliced and FFS denotes fully-fixsliced implementations. Time is given in cycles per 128-bit block, and ROM and RAM are given in bytes. For SKINNY, since ROM is also not separated into Code and Data, Data is shown as "–". For the encryption I/O of GIFT-128, in accordance with the in-place setting of this paper, the 16 bytes for the output block are excluded and the 320-byte round-key buffer is added in parentheses.

				Time		ROM		RAM	
Cipher	Operation	Representation	Blocks	M3	M4	Code	Data	I/O	Stack
Unrolled implementations									
128-bit key/tweakey variants									
AES-128	Key sched.	SFS(LUT)	1	1158	1235	3768	1024	368	196
	Key sched.	FFS(LUT)	1	1178	1255	3848	1024	368	196
	Enc.	SFS	2	1312	1314	9504	0	32(+352)	112
	Enc.	FFS	2	1273	1275	9184	0	32(+352)	112
GIFT-128	Key sched.	Fix	–	966	969	3510	0	336	48
	Enc.	Fix	1	1316	1319	4868	0	32(+320)	56
SKINNY-128-128	Enc.	FFS	2	1862	1872	1620	–	32(+1280)	60
256-bit key/tweakey variants									
AES-256	Enc.	SFS	2	1804	1806	13136	0	32(+480)	112
	Enc.	FFS	2	1745	1747	12656	0	32(+480)	112
SKINNY-128-256	Enc.	FFS	2	2214	2224	1628	–	32(+1536)	60
Non-unrolled implementations									
128-bit key variants									
AES-128	Key sched.	SFS(LUT)	1	1461	1538	784	256	368	212
	Key sched.	FFS(LUT)	1	1481	1561	936	256	368	212
	Enc.	SFS	2	1394	1415	1820	0	32(+352)	116
	Enc.	FFS	2	1348	1369	2360	0	32(+352)	116
GIFT-128	Key sched.	Fix	–	1813	1812	1100	0	336	56
	Enc.	Fix	1	1444	1427	1332	160	32(+320)	64
256-bit key variants									
AES-256	Enc.	SFS	2	1918	1945	1860	0	32(+480)	116
	Enc.	FFS	2	1849	1874	2392	0	32(+480)	116

corresponding Speck, MSX, AES, and SKINNY variants. Thus, even though Rotom is a TBC, it maintains a small RAM footprint in the precomputed round-tweakey setting.

Code Size. The non-unrolled implementations show that Rotom can keep a practical code size while retaining competitive performance. The ROM sizes in Table 11 are reported separately for the tweakey schedule, encryption, and decryption. We do not report the ROM size of a fully integrated implementation including all three components.

Therefore, the code-size discussion is based on the measurements of each individual operation. Since several functions are shared among the tweakey schedule, encryption, and decryption, an integrated implementation may further reduce the total ROM size by sharing common routines. We leave this optimization as future work.

Comparison with mid-pSquare. Among the variants of mid-pSquare, we choose the one with a 124-bit tweak ($\tau = 1$), whose tweak length equals its block length, as in Rotom-128. We use its unprotected implementation, since we do not provide masked implementations of Rotom, although mid-pSquare is primarily designed for masked implementations. In this setting, mid-pSquare requires about 5.1k cycles per encryption on Cortex-M4 [BGM+25]. Since this value includes the key and tweak schedules, we compare it with the sum of the tweakey-schedule and encryption cycles of Rotom-128, which is 1180 cycles (577 + 603) in the fully unrolled implementation and 1255 cycles (617 + 638) in the non-unrolled

implementation on Cortex-M4.

Summary. The results show that Rotom achieves cycle counts close to Speck on ARM Cortex-M3/M4 while providing native tweak support, and runs with substantially fewer cycles than MSX. It also keeps RAM usage small even in the precomputed round-tweakey setting. These results show that the main design choices—a fixed multiplication-based S-box, a word-level recursive linear layer, and a compact tweakey schedule—translate into efficient software implementation on 32-bit ARM Cortex-M microcontrollers.

5.4 Side-channel

5.4.1 Timing Attacks

On ARM Cortex-M3 and Cortex-M4, Rotom can be implemented in constant time using fixed-latency instructions and fixed memory-access patterns, without secret-dependent branches or table lookups. On other platforms, the timing behavior of the employed instructions, particularly multiplication, must be verified when implementing Rotom.

5.4.2 Power/Electromagnetic (EM) Side-channel Attacks

As the Rotom family is designed for microcontroller implementation, it would be important to consider its vulnerability/resistance to power/electromagnetic (EM) side-channel attacks. A side-channel adversary may target the key whitening and first/last round S-box as in existing attacks. Rotom family employs the arithmetic addition for whitening key and multiplication for S-box, which would be a major target as in side-channel attacks on conventional ciphers.

Target. We perform the correlation power analysis (CPA) [BCO04] on these components by simulation. We suppose that the attacker first recovers the whitening key by CPA on arithmetic addition, then recovers the first-round key by CPA on the multiplication in S-box. Although the targets are 32-bit-wise operations, a 32-bit key guess is computationally heavy. Hence, in the CPA, we used an eight-bit key guess; that is, the selection functions are initially given by $\mathrm{LSB}_8(T + K)$ and $\mathrm{LSB}_8((T \oplus K) \times c)$, where LSB_8 is an eight-bit truncation and T is a byte of plaintext or intermediate value calculated by plaintext and recovered whitening key. We suppose that the attacker recovers the upper bits by CPA with recovered key chunks.

Result. Figure 4 shows the result of CPAs on the eight-bit LSB, where the horizontal and vertical axes denote the number of traces and the success rate (SR), respectively. The simulated leakage X is given as $X = Z + N$, where Z is the target value (i.e., $T + K$ and $(T \oplus K) \times c$) and N is a zero-mean Gaussian noise with $\sigma^2 = 10$. For the SR evaluation, we performed CPA 100 times with random keys for each number of traces and counted the number of successful attacks. Although the hypothetical values guessed by CPA include a secret-dependent algorithmic noise due to the eight-bit truncation and structure of arithmetic addition/multiplication, the result shows that we could correctly estimate the secret key within 500 traces. We need a countermeasure against side-channel attacker in some application scenarios where power/EM side-channel attacks are feasible.

Countermeasure. Masking is a major countermeasure against these side-channel attacks. As Rotom family utilizes both arithmetic and Boolean operations, arithmetic-to-Boolean (A2B) and Boolean-to-arithmetic (B2A) mask conversion [Cor17] enables an effective masking as in addition-rotation-XOR (ARX) ciphers. As mentioned in Section 3.3.1, the S-box can be masked without field-multiplication gadgets. The other operations in Rotom

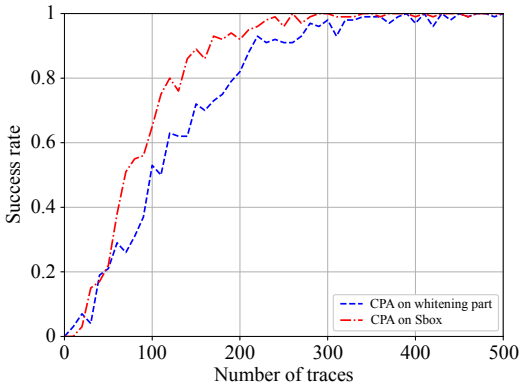


Figure 4: Simulation result of CPA.

family are linear over the field/ring. Hence, we can mask the whole of Rotom without masked field-multiplier but with linear gadgets and A2B/B2A conversion. Moreover, the side-channel attacker must recover the whitening key at first using the leakage from arithmetic addition, while key recovery from arithmetic masking is more difficult than that from Boolean masking if the leakage follows an HW model [UIM+25]. Since it has been shown that the HW leakage model would fit some microcontrollers [MOP07], the compatibility of Rotom with arithmetic masking would make a sense as a microcontroller-oriented cipher. We leave evaluation of masked Rotom implementation as future work.

6 Conclusion

We proposed Rotom, a 128-bit tweakable block cipher for efficient software implementation on 32-bit microcontrollers. The design combines a fixed multiplication-based 16-bit S-box, a word-level recursive linear layer, a compact tweakey schedule, and modular-addition boundary key mixing.

A key advantage of Rotom over typical ARX-based designs is that its fixed nonlinear layer enables explicit differential and linear characteristic bounds. The local DCP/LCP properties of the 16-bit S-box can be computed once and combined with active-S-box bounds at the cipher level.

Implementation results on ARM Cortex-M3 and Cortex-M4 show that these components translate into efficient software performance. Rotom achieves cycle counts close to Speck under the reported implementation while supporting a public tweak, and runs substantially faster than MSX. The non-unrolled implementations retain most of this performance while reducing the ROM requirement to a practical level. Even in the precomputed round-tweakey setting, where all round tweakeys are stored in RAM, Rotom maintains a small memory footprint. From a side-channel perspective, Rotom admits constant-time implementations on Cortex-M3 and Cortex-M4 and is amenable to masking using linear gadgets together with A2B and B2A conversions.

Overall, Rotom provides a practical design point for microcontroller-oriented implementations between highly efficient ARX-based ciphers and more directly analyzable S-box-based designs.

Acknowledgments

This work was supported by JST CRONOS, Japan, Grant Number JPMJCS25N1 and JSPS KAKENHI, Grant Number JP24H00696.

References

[ABD+23] Roberto Avanzi, Subhadeep Banik, Orr Dunkelman, Maria Eichlseder, Shibam Ghosh, Marcel Nageler, and Francesco Regazzoni. The QARMAv2 family of tweakable block ciphers. *IACR Transactions on Symmetric Cryptology*, 2023(3):25–73, 2023.

[AF14] Daniel Augot and Matthieu Finiasz. Direct construction of recursive MDS diffusion layers using shortened BCH codes. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2014.

[ANP20] Alexandre Adomnicai, Zakaria Najm, and Thomas Peyrin. Fixslicing: A new GIFT representation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):402–427, 2020.

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):402–425, 2021.

[Arm10a] Arm Limited. *ARMv7-M Architecture Reference Manual*, 2010. Architecture reference manual for Cortex-M3 class processors.

[Arm10b] Arm Limited. *ARMv7E-M Architecture Reference Manual*, 2010. Architecture reference manual for Cortex-M4 class processors.

[AS09] Kazumaro Aoki and Yu Sasaki. Preimage attacks on one-block MD4, 63-step MD5 and more. In *Selected Areas in Cryptography, 15th International Workshop, SAC 2008, Sackville, New Brunswick, Canada, August 14–15, 2008, Revised Selected Papers*, volume 5381 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2009.

[Ava17] Roberto Avanzi. The QARMA block cipher family: Almost MDS matrices over rings with zero divisors, nearly symmetric Even-Mansour constructions with non-involutory central rounds, and search heuristics for low-latency S-boxes. *IACR Transactions on Symmetric Cryptology*, 2017(1):4–44, March 2017.

[BBdS+20] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Alzette: A 64-bit arx-box - (feat. CRAX and TRAX). In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 419–448. Springer, 2020.

[BBI+15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015.

[BBS99] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. In *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 12–23. Springer, 1999.

[BC13] Christina Boura and Anne Canteaut. On the influence of the algebraic degree of f^{-1} on the algebraic degree of $g \circ f$. *IEEE Transactions on Information Theory*, 59(1):691–702, 2013.

[BC16] Christina Boura and Anne Canteaut. Another view of the division property. In *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 654–682. Springer, 2016.

[BCJW02] Nikita Borisov, Monica Chew, Robert Johnson, and David A. Wagner. Multiplicative differentials. In *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4–6, 2002, Revised Papers*, volume 2365 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2002.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

[BDK02] Eli Biham, Orr Dunkelman, and Nathan Keller. Enhancing differential-linear cryptanalysis. In Yuliang Zheng, editor, *Advances in Cryptology – ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1–5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 254–266. Springer, 2002.

[BFL+21] Olivier Bronchain, Sebastian Faust, Virginie Lallemand, Gregor Leander, Léo Perrin, and François-Xavier Standaert. MOE: Multiplication operated encryption with trojan resilience. *IACR Transactions on Symmetric Cryptology*, 2021(1):78–129, 2021.

[BGM+25] Brieuc Balon, Lorenzo Grassi, Pierrick Méaux, Thorben Moos, François-Xavier Standaert, and Matthias Johann Steiner. mid-pSquare: Leveraging the strong side-channel security of prime-field masking in software. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(4):486–519, 2025.

[BHLP25] Christof Beierle, Phil Hebborn, Gregor Leander, and Yevhen Perehuda. Integral resistance of block ciphers with key whitening by modular addition. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2025, Proceedings, Part V*, volume 16004 of *Lecture Notes in Computer Science*, pages 497–529. Springer, 2025.

[BII+25] Subhadeep Banik, Tatsuya Ishikawa, Takanori Isobe, Ryoma Ito, Kazuhiko Minematsu, Kazuma Nakata, Mostafizar Rahman, and Kosei Sakamoto. Dialga: A family of low-latency tweakable block ciphers using multiple linear layers. *IACR Trans. Symmetric Cryptol.*, 2025(4):70–124, 2025.

[BL18] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to, 2018.

[BMP+23] M. Majid Butt, Nitin R. Mangalvedhe, Nuno K. Pratas, Johannes Harrebek, John Kimionis, Muhammad Tayyab, Oana-Elena Barbu, Rapeepat Ratasuk, and Benny Vejlgaard. Ambient IoT: A missing link in 3GPP IoT devices landscape. arXiv:2312.06569, 2023.

[BODKW19] Achiya Bar-On, Orr Dunkelman, Nathan Keller, and Ariel Weizman. DLCT: A new tool for differential-linear cryptanalysis. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 313–342. Springer, 2019.

[BPP+17] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A small present – towards reaching the limit of lightweight encryption. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 321–345. Springer, 2017.

[BSS+15] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 175:1–175:6. ACM, 2015.

[BW99] Alex Biryukov and David Wagner. Slide attacks. In *Fast Software Encryption, 6th International Workshop, FSE'99, Rome, Italy, March 24–26, 1999, Proceedings*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 1999.

[CHP+18] Carlos Cid, Tao Huang, Thomas Peyrin, Yu Sasaki, and Ling Song. Boomerang connectivity table: A new cryptanalysis tool. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29–May 3, 2018, Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 683–714. Springer, 2018.

[Cor17] Jean-Sébastien Coron. High-order conversion from boolean to arithmetic masking. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 93–114, Cham, 2017. Springer International Publishing.

[CP02] Nicolas T. Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *Advances in Cryptology – ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002.

[DCK+15] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. FELICS - fair evaluation of LIghtweight cryptographic systems. https://www.cryptolux.org/index.php/FELICS, 2015.

[DCK+19] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of lightweight block ciphers for the internet of things. *Journal of Cryptographic Engineering*, 9(3):283–302, 2019.

[DPU+16] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design strategies for ARX with provable bounds: Sparx and LAX. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 484–513, 2016.

[Fur01] Vladimir Furman. Differential cryptanalysis of nimbus. In *Fast Software Encryption, 8th International Workshop, FSE 2001, Yokohama, Japan, April 2–4, 2001, Revised Papers*, volume 2355 of *Lecture Notes in Computer Science*, pages 187–195. Springer, 2001.

[FWG+16] Kai Fu, Meiqin Wang, Yinghua Guo, Siwei Sun, and Lei Hu. MILP-based automatic search algorithms for differential and linear trails for Speck. In *Fast Software Encryption – FSE 2016*, volume 9783 of *Lecture Notes in Computer Science*, pages 268–288. Springer, 2016.

[HSM+20] Arliones Hoeller, Jean Sant'Ana, Juho Markkula, Konstantin Mikhaylov, Richard Demo Souza, and Hirley Alves. Beyond 5g low-power wide-area networks: A LoRaWAN suitability study. *Sensors*, 20(8):2254, 2020.

[IMPS17] Tetsu Iwata, Kazuhiko Minematsu, Thomas Peyrin, and Yannick Seurin. ZMAC: A fast tweakable block cipher mode for highly secure message authentication. In *CRYPTO (3)*, volume 10403 of *Lecture Notes in Computer Science*, pages 34–65. Springer, 2017.

[JNP14] Jérémy Jean, Ivica Nikolic, and Thomas Peyrin. Tweaks and keys for block ciphers: The TWEAKEY framework. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 2014.

[JNPS21] Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. The deoxys AEAD family. *J. Cryptol.*, 34(3):31, 2021.

[KHL10] Jongsung Kim, Seokhie Hong, and Jongin Lim. Impossible differential cryptanalysis using matrix method. *Discrete Mathematics*, 310(5):988–1002, 2010.

[KN10] Dmitry Khovratovich and Ivica Nikolić. Rotational cryptanalysis of ARX. In *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7–10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*, pages 333–346. Springer, 2010.

[KR11] Ted Krovetz and Phillip Rogaway. The software performance of authenticated-encryption modes. In *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 306–327. Springer, 2011.

[KSI25] Rikuto Kurahara, Kosei Sakamoto, and Takanori Isobe. Finding new differential characteristics on SPECK family. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 108(5):760–763, 2025.

[KW02] Lars R. Knudsen and David Wagner. Integral cryptanalysis. In *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4–6, 2002, Revised Papers*, volume 2365 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2002.

[LH94] Susan K. Langford and Martin E. Hellman. Differential-linear cryptanalysis. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21–25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 17–25. Springer, 1994.

[LM91] Xuejia Lai and James L. Massey. A proposal for a new block encryption standard. In *Advances in Cryptology – EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer, 1991.

[LRW02] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2002.

[MCU] 32 bit mcu market overview. https://www.futuremarketreport.com/industry-report/32-bit-mcu-market.

[MMH+14] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: An efficient MAC algorithm for 32-bit microcontrollers. In *Selected Areas in Cryptography – SAC 2014*, volume 8781 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2014.

[MNSV97] David M'Raïhi, David Naccache, Jacques Stern, and Serge Vaudenay. XMX: A firmware-oriented block cipher based on modular multiplications. In *Fast Software Encryption, 4th International Workshop, FSE '97*, volume 1267 of *Lecture Notes in Computer Science*, pages 166–171. Springer, 1997.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer New York, 2007.

[mra] mrahman454. Block-cipher-MSX: Reference implementation of MSX, Speck, and AES for microcontrollers. https://github.com/mrahman454/Block-Cipher-MSX.

[MSR+25] Kazuhiko Minematsu, Tomoyasu Suzaki, Mostafizar Rahman, Sahiba Suryawanshi, and Takanori Isobe. MSX: Lightweight block ciphers for microcontrollers with high-assurance against differential and linear attacks. *IACR Communications in Cryptology*, 2(4):1–32, 2025.

[Nat21] National Security Agency. Simon and speck: SUPERCOP implementations. https://github.com/nsacyber/simon-speck-supercop/tree/master/crypto_stream, 2021.

[NL18] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF protocols. RFC 8439, June 2018.

[Por18] Thomas Pornin. Constant-time crypto. BearSSL documentation, 2018. Accessed: 2026-06-29.

[RKS17] Usman Raza, Parag Kulkarni, and Mahesh Sooriyabandara. Low power wide area networks: An overview. *IEEE Communications Surveys & Tutorials*, 19(2):855–873, 2017.

[SA09] Yu Sasaki and Kazumaro Aoki. Finding preimages in full MD5 faster than exhaustive search. In *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 134–152. Springer, 2009.

[Scr01] Beale Screamer. Microsoft's digital rights management scheme—technical details. http://cryptome.org/ms-drm.htm, October 2001. Accessed: 2025-07-03.

[SDMS12] Mahdi Sajadieh, Mohammad Dakhilalian, Hamid Mala, and Pouyan Sepehrdad. Recursive diffusion layers for block ciphers and hash functions. In *Fast Software Encryption, 19th International Workshop, FSE 2012, Washington, DC, USA, March 19–21, 2012, Revised Selected Papers*, volume 7549 of *Lecture Notes in Computer Science*, pages 385–401. Springer, 2012.

[SHY16] Ling Song, Zhangjie Huang, and Qianqian Yang. Automatic differential analysis of ARX block ciphers with application to SPECK and LEA. In *Information Security and Privacy – ACISP 2016*, volume 9723 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 2016.

[SWW21] Ling Sun, Wei Wang, and Meiqin Wang. Accelerating the search of differential and linear characteristics with the SAT method. *IACR Transactions on Symmetric Cryptology*, 2021(1):269–315, 2021.

[TMC+25] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Jinkeon Kang, and John Kelsey. Ascon-based lightweight cryptography standards for constrained devices: Authenticated encryption, hash, and extendable output functions. NIST Special Publication 800-232, National Institute of Standards and Technology, August 2025.

[Tod15] Yosuke Todo. Structural evaluation by generalized integral property. In *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 287–314. Springer, 2015.

[UIM+25] Rei Ueno, Akiko Inoue, Kazuhiko Minematsu, Akira Ito, and Naofumi Homma. A formal security proof of masking: Reduction from strong noisy leakage to probing model without random probing and application to LR primitive. Cryptology ePrint Archive, Paper 2025/2199, 2025.

[Vau03] Serge Vaudenay. Decorrelation: A theory for block cipher security. *J. Cryptol.*, 16(4):249–286, 2003.

[Wag99] David A. Wagner. The boomerang attack. In Lars R. Knudsen, editor, *Fast Software Encryption, 6th International Workshop, FSE '99, Rome, Italy, March 24–26, 1999, Proceedings*, volume 1636 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 1999.

[YMUM+21] Ville Yli-Mäyry, Rei Ueno, Noriyuki Miura, Makoto Nagata, Shivam Bhasin, Yves Mathieu, Tarik Graba, Jean-Luc Danger, and Naofumi Homma. Diffusional side-channel leakage from unrolled lightweight block ciphers: A case study of power analysis on prince. *IEEE Transactions on Information Forensics and Security*, 16:1351–1364, 2021.

[ZXM+24] Kan Zheng, Rongtao Xu, Jie Mei, Haojun Yang, Lei Lei, and Xianbin Wang. Ambient IoT towards 6G: Standardization, potentials, and challenges. arXiv:2412.12519, 2024.

A Analysis Conventions

A.1 Differential and linear probabilities

For a mapping $S : \mathbb{F}_2^n \to \mathbb{F}_2^n$, the differential probability for input difference Δx and output difference Δy is

$$\mathrm{DP}_S(\Delta x, \Delta y) = 2^{-n} \left|\{x \in \mathbb{F}_2^n \mid S(x) \oplus S(x \oplus \Delta x) = \Delta y\}\right|.$$

The maximum differential probability and the corresponding differential weight are

$$\mathrm{MDP}(S) = \max_{\Delta x \neq 0, \Delta y} \mathrm{DP}_S(\Delta x, \Delta y), \qquad w_D(S) = -\log_2 \mathrm{MDP}(S).$$

For masks $\alpha, \beta \in \mathbb{F}_2^n$, let $\alpha \cdot x$ denote the standard inner product over $\mathbb{F}_2$. The correlation of the linear approximation (α, β) is

$$\mathrm{Cor}_S(\alpha, \beta) = 2^{-n} \sum_{x \in \mathbb{F}_2^n} (-1)^{\alpha \cdot x \oplus \beta \cdot S(x)}.$$

We use the squared-correlation convention:

$$\mathrm{MLP}(S) = \max_{\alpha \neq 0, \beta \neq 0} \mathrm{Cor}_S(\alpha, \beta)^2, \qquad w_L(S) = -\log_2 \mathrm{MLP}(S).$$

The single-S-box score used in the design rationale is

$$w(S) = \min\{w_D(S), w_L(S)\}.$$

For a target characteristic weight W, we define

$$\mathrm{AS}_W(S) = \frac{W}{w(S)}, \qquad N_W(S) = \left\lceil \frac{W}{w(S)} \right\rceil.$$

A.2 Active cells and branch numbers

For a state represented as eight 16-bit cells $X = (x_0, \ldots, x_7)$, define

$$\mathrm{wt}_{16}(X) = |\{i \mid x_i \neq 0\}|.$$

An active S-box is a 16-bit S-box whose input difference or input mask is nonzero, depending on whether differential or linear propagation is considered.

For a linear map L on the 128-bit state, its differential branch number over 16-bit cells is

$$B_{16}^{\mathrm{diff}}(L) = \min_{X \neq 0} \left(\mathrm{wt}_{16}(X) + \mathrm{wt}_{16}(L(X))\right).$$

Since the state has eight 16-bit cells, the maximum possible value is 9. For linear-mask propagation, we use

$$B_{16}^{\mathrm{lin}}(L) = B_{16}^{\mathrm{diff}}(L^{-T}).$$

When a common 16-bit branch number is reported for a linear-layer candidate, it means a lower bound that holds for both $B_{16}^{\mathrm{diff}}(L)$ and $B_{16}^{\mathrm{lin}}(L)$, unless stated otherwise.

B Round Components

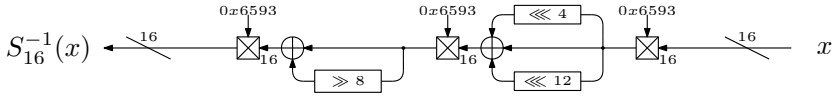


Figure 5: Structure of the inverse 16-bit nonlinear permutation S_{16}^{-1}.

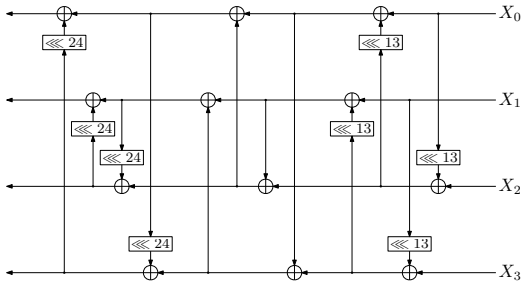


Figure 6: Inverse linear layer $\mathcal{M}^{-1}$.

Algorithm 2 The 16-bit nonlinear permutation S_{16} and its inverse.

function $S_{16}(x)$
  $x \leftarrow (c \cdot x) \bmod 2^{16}$
  $x \leftarrow x \oplus (x \gg 8)$
  $x \leftarrow (c \cdot x) \bmod 2^{16}$
  $x \leftarrow x \oplus (x \lll_{16} 4) \oplus (x \lll_{16} 12)$
  $x \leftarrow (c \cdot x) \bmod 2^{16}$
  return x
end function

function $S_{16}^{-1}(x)$
  $x \leftarrow (c^{-1} \cdot x) \bmod 2^{16}$
  $x \leftarrow x \oplus (x \lll_{16} 4) \oplus (x \lll_{16} 12)$
  $x \leftarrow (c^{-1} \cdot x) \bmod 2^{16}$
  $x \leftarrow x \oplus (x \gg 8)$
  $x \leftarrow (c^{-1} \cdot x) \bmod 2^{16}$
  return x
end function

Algorithm 3 The linear layer $\mathcal{M}$ and its inverse $\mathcal{M}^{-1}$.

function $\mathcal{M}(X_0, X_1, X_2, X_3)$
Stage 0
  $X_0 \leftarrow X_0 \oplus (X_3 \lll_{32} 24)$
  $X_1 \leftarrow X_1 \oplus (X_2 \lll_{32} 24)$
  $X_2 \leftarrow X_2 \oplus (X_1 \lll_{32} 24)$
  $X_3 \leftarrow X_3 \oplus (X_0 \lll_{32} 24)$
Stage 1
  $X_0 \leftarrow X_0 \oplus X_2$
  $X_1 \leftarrow X_1 \oplus X_3$
  $X_2 \leftarrow X_2 \oplus X_1$
  $X_3 \leftarrow X_3 \oplus X_0$
Stage 2
  $X_0 \leftarrow X_0 \oplus (X_2 \lll_{32} 13)$
  $X_1 \leftarrow X_1 \oplus (X_3 \lll_{32} 13)$
  $X_2 \leftarrow X_2 \oplus (X_0 \lll_{32} 13)$
  $X_3 \leftarrow X_3 \oplus (X_1 \lll_{32} 13)$
  return (X_0, X_1, X_2, X_3)
end function

function $\mathcal{M}^{-1}(X_0, X_1, X_2, X_3)$
Inverse of Stage 2
  $X_3 \leftarrow X_3 \oplus (X_1 \lll_{32} 13)$
  $X_2 \leftarrow X_2 \oplus (X_0 \lll_{32} 13)$
  $X_1 \leftarrow X_1 \oplus (X_3 \lll_{32} 13)$
  $X_0 \leftarrow X_0 \oplus (X_2 \lll_{32} 13)$
Inverse of Stage 1
  $X_3 \leftarrow X_3 \oplus X_0$
  $X_2 \leftarrow X_2 \oplus X_1$
  $X_1 \leftarrow X_1 \oplus X_3$
  $X_0 \leftarrow X_0 \oplus X_2$
Inverse of Stage 0
  $X_3 \leftarrow X_3 \oplus (X_0 \lll_{32} 24)$
  $X_2 \leftarrow X_2 \oplus (X_1 \lll_{32} 24)$
  $X_1 \leftarrow X_1 \oplus (X_2 \lll_{32} 24)$
  $X_0 \leftarrow X_0 \oplus (X_3 \lll_{32} 24)$
  return (X_0, X_1, X_2, X_3)
end function

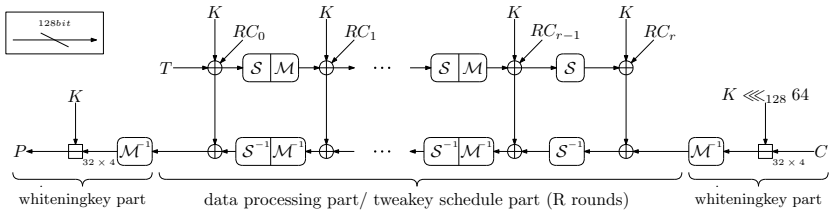
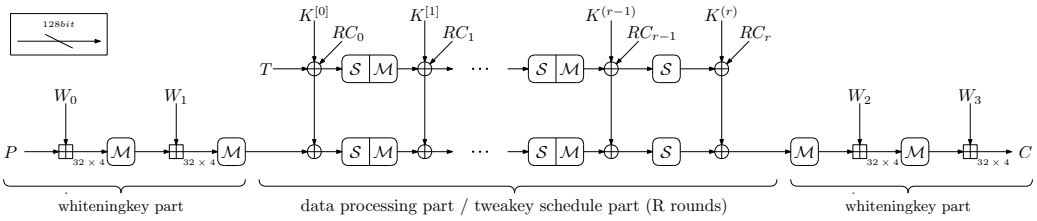
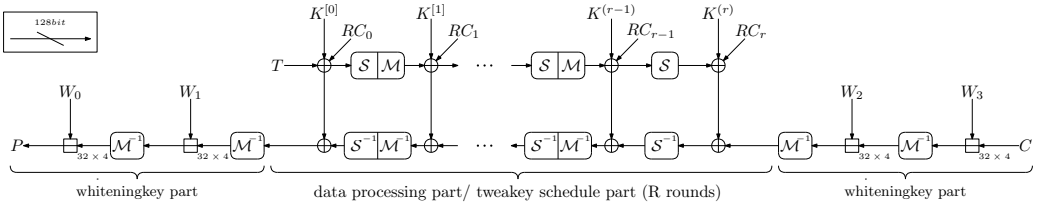


Figure 7: Overall decryption procedure for the 128-bit-key variants. The tweakey schedule is identical to that used for encryption, whereas the data-processing branch applies the inverse round components in reverse order. The final tweakey-schedule iteration omits $\mathcal{M}$, and the first inverse data-processing round encountered from the ciphertext side omits $\mathcal{M}^{-1}$. Decryption proceeds from C to P, from right to left in the figure.



(a) Encryption



(b) Decryption

Figure 8: Overall encryption and decryption procedures for the 256-bit-key variants. The tweakey schedule is identical in both directions, whereas decryption applies the inverse data-processing and whitening operations in reverse order. The final tweakey-schedule iteration omits $\mathcal{M}$, and the first inverse data-processing round encountered from the ciphertext side omits $\mathcal{M}^{-1}$. In panel (b), decryption proceeds from C to P, from right to left.

C Details of Security Analysis

C.1 Boomerang and Differential-Linear Attacks

Boomerang attacks combine two differential characteristics, typically one for the upper part and one for the lower part of the cipher [Wag99]. Differential-linear attacks combine a differential characteristic for one part with a linear approximation for the remaining part [LH94, BDK02]. In this section, we estimate such attacks from the single-characteristic bounds obtained in Section 4.1.

128-Bit-Key Variants. For boomerang attacks, we use the standard characteristic-level estimate that combines the two differential parts with probability p^2q^2. Hence, if one of the two differential parts already has weight w, then the boomerang weight contains at least $2w$. From Table 9, three rounds already have related-tweak differential weight 86.47. Thus, any 5-round boomerang split contains a differential part of at least three rounds and already contributes at least

$$2 \cdot 86.47 = 172.94$$

bits of boomerang weight from that part alone. This exceeds the 80-bit threshold of Rotom-128*. For Rotom-128, any 6-round split either contains a 4-round differential part, whose weight is 134.51, or is split as $3 + 3$ rounds. Both cases exceed the 128-bit threshold by a large margin under the same characteristic-level estimate.

For differential-linear attacks, we combine the differential weight of the differential part and the squared-correlation weight of the linear part. From Table 9, the 3-round related-tweak differential weight is 86.47. The 3-round single-tweak linear weight is 82.55, and the 4-round linear weight is 128.42. Therefore, any 5-round differential-linear split contains either a 3-round differential part or a 3-round linear part, already exceeding the 80-bit threshold of Rotom-128*. For Rotom-128, a $3 + 3$ split gives weight at least

$$86.47 + 82.55 = 169.02,$$

while a split containing a 4-round part already exceeds the 128-bit threshold on the differential or linear side. Thus, Rotom-128 also has sufficient margin against characteristic-level differential-linear attacks.

These estimates do not constitute full boomerang-hull or differential-linear-hull bounds. In particular, we do not evaluate boomerang connectivity tables or differential-linear connectivity tables [CHP+18, BODKW19]. Nevertheless, the available reduced-round bounds already exceed the claimed security thresholds with margin. Together with the modular-addition whitening and adjacent linear layers, which make key-recovery extensions harder than peeling off XOR subkeys, these results support the claimed resistance of both 128-bit-key variants against boomerang and differential-linear attacks.

256-Bit-Key Variants. The same characteristic-level bounds apply to the 256-bit-key variants, since the alternating key halves act only as round-dependent constants under a fixed master key and do not affect difference or mask propagation. Thus, the 6-round variant Rotom-256* inherits the estimates for Rotom-128, which already exceed its 80-bit threshold.

For the 7-round variant Rotom-256, any boomerang split contains a differential part of at least four rounds. Its contribution alone has boomerang weight at least

$$2 \cdot 134.51 = 269.02,$$

which exceeds the claimed 240-bit time-complexity threshold. Similarly, any 7-round differential-linear split contains either a differential part or a linear part of at least four

rounds. The corresponding weights are at least 134.51 and 128.42, respectively, and therefore exceed the 128-bit data-complexity threshold of Rotom-256.

C.2 Multiplicative Differential Attacks

Multiplicative differential attacks exploit input-output relations that are compatible with modular multiplication, such as complement-type pairs or doubling pairs [BCJW02, Fur01, MNSV97, Scr01]. We evaluate such relations for the 16-bit S-box S_{16}, where the nonlinearity of the data processing part is concentrated.

For the complement-type relation, we exhaustively checked all 2^{16} inputs and obtained

$$2^{-16}\,|\{x : S_{16}(x \oplus \texttt{0xffff}) = S_{16}(x) \oplus \texttt{0xffff}\}| = 2^{-15}.$$

For the doubling relation, which is a typical relation considered in multiplicative differential attacks, we similarly evaluated

$$2^{-16}\,|\{x : S_{16}(2x) = 2S_{16}(x)\}| = \frac{4108}{2^{16}} \approx 2^{-4.00},$$

where the multiplication by 2 is taken modulo 2^{16}. We obtained the same values for S_{16}^{-1}.

Although the constant multiplication M_c commutes with the doubling relation, the shift-xor and rotation-xor maps inside S_{16} do not preserve it deterministically. If an attacker tries to maintain a cell-wise doubling relation through a fully active S-box layer, then each active 16-bit cell must satisfy the local compatibility condition $S_{16}(2x) = 2S_{16}(x)$. For one cell, this condition holds with probability $4108/2^{16}$. However, the linear layer $\mathcal{M}$ does not preserve such a cell-wise multiplicative relation, since it mixes words by XORs and rotations. After 1.5 rounds, any active relation is diffused to all eight 16-bit S-boxes. Therefore, multiplicative differential relations cannot be propagated deterministically across rounds. We do not identify a useful multiplicative differential attack on the proposed variants.

C.3 Algebraic Attacks

We evaluate algebraic attacks [BC13, CP02] by tracking the algebraic degree of the round function over $\mathbb{F}_2$. The XOR operations and the linear layer $\mathcal{M}$ are linear and therefore do not increase the algebraic degree. Degree growth is caused by the 16-bit nonlinear permutation S_{16} and the modular additions in the whitening part.

For S_{16}, we computed the ANF of all Boolean components. For every nonzero mask $v \in \mathbb{F}_2^{16}$, the component function $v \cdot S_{16}$ has algebraic degree 15. The same property holds for S_{16}^{-1}. Thus, one active 16-bit cell already reaches the maximum possible local degree after one application of S_{16}.

We then propagate this local degree information through the round function. After one round, an active cell has degree 15. After 1.5 rounds, the full-diffusion property ensures that the activity reaches all eight 16-bit S-boxes. After two rounds, our degree-propagation analysis yields an upper bound of degree 127, which is the highest possible algebraic degree for a component of a 128-bit permutation. Additional rounds preserve this upper bound.

The same analysis applies to the 256-bit-key variants, since they use the same data-processing part, while the alternating key halves are injected only by XOR under a fixed master key.

Therefore, an algebraic description of any of the four variants involves high-degree equations over the full 128-bit state, together with the dependencies introduced by the tweakey schedule and whitening part. We do not identify a direct algebraic shortcut, including Gröbner-basis or SAT-based approaches, that is more efficient than exhaustive search.

C.4 Integral Attacks

Integral attacks exploit subsets of plaintexts whose sums remain independent of the secret key [KW02]. Such distinguishers are closely related to algebraic-degree growth and division-property propagation [Tod15, BC16]. As discussed in the algebraic analysis, every nonzero component function of the 16-bit S-box S_{16} has algebraic degree 15, and our degree-propagation analysis yields an upper bound of degree 127 after two rounds, which is the highest possible degree for a component of a 128-bit permutation. Although this upper bound does not imply that the actual degree attains 127, its rapid growth suggests that low-degree integral structures are difficult to maintain through the data-processing part.

In addition, the whitening part uses word-wise modular key addition instead of XOR whitening. This design choice is motivated by the recent framework for integral resistance of block ciphers with modular-addition key whitening [BHLP25]. In all variants, the 128-bit state is partitioned into four 32-bit words, and the input and output whitening use $\boxplus_{32\times 4}$ with whitening values derived from the master key. For the 128-bit-key variants, these values are obtained from the master key and its 64-bit rotation, whereas those of the 256-bit-key variants combine the two 128-bit key halves. Thus, the whitening part can be viewed as word-wise modular pre- and post-whitening around the data-processing part. Carry propagation within each 32-bit word introduces nonlinear key-dependent terms in the ANF, unlike XOR whitening, which adds the key linearly.

The framework of modular key whitening shows that, for a t-word-wise pre- and post-whitened construction with independent whitening keys, integral resistance can be reduced to checking the independence of selected high-degree ANF coefficients. In our case, the relevant word partition is $t = 4$ and $s = 32$. Although the formal result does not directly apply because the whitening values and round tweakeys are derived from the same master key, the same mechanism motivates the use of modular key addition in the whitening part.

Together, the rapid growth of the degree upper bound in the data-processing part and the nonlinear key dependencies introduced by modular-addition whitening make it difficult to maintain a key-independent integral property through the full construction. We therefore do not identify an integral distinguisher or integral key-recovery attack against any of the four variants.

C.5 Impossible Differential Attacks

We evaluate impossible differential attacks based on the standard miss-in-the-middle approach [BBS99, KHL10]. Since our construction achieves full diffusion in 1.5 rounds in both the forward and backward directions, any nonzero difference reaches all eight 16-bit S-boxes after 1.5 rounds.

In a miss-in-the-middle distinguisher, the forward and backward truncated differential trails must impose incompatible conditions at the middle state. In a standard active-S-box-based analysis, such a contradiction is established before both trails become fully diffused. Therefore, the number of rounds covered by this truncated impossible-differential core is

$$1.5 + 1.5 = 3.$$

Thus, within the standard active-S-box-based analysis, the miss-in-the-middle technique can cover at most three rounds.

Extending this core to a key-recovery attack requires additional outer rounds and guesses of the corresponding round-tweakey and whitening-key material. Such extensions are further constrained by the modular-addition whitening and the dependencies introduced by the tweakey schedule. In particular, attacking Rotom-128* requires adding two outer rounds, while attacking Rotom-128 and Rotom-256* requires three, and attacking Rotom-256 requires four. We do not identify an efficient key-recovery attack obtained by extending this three-round core against any of the four variants.

C.6 Rotational Analysis

Rotational cryptanalysis studies whether a cipher preserves relations between rotated inputs and outputs [KN10]. In our construction, XORs and fixed rotations may preserve some word-wise rotational relations. Hence, the relevant question is whether the nonlinear layer and the tweakey schedule preserve such relations.

For the 16-bit S-box, we evaluated

$$p_r = 2^{-16} \left|\{x : S_{16}(\mathrm{ROTL}_{16}(x, r)) = \mathrm{ROTL}_{16}(S_{16}(x), r)\}\right|$$

for all nonzero rotations $1 \le r < 16$. For the specified S_{16}, the maximum value is

$$\max_{1 \le r < 16} p_r = \frac{1046}{2^{16}} \approx 2^{-5.97}.$$

Thus, S_{16} does not preserve any nontrivial rotational relation deterministically. This is mainly due to the modular multiplications by the fixed odd constant and the shift-xor map $x \mapsto x \oplus (x \gg 8)$, which is not rotation-equivariant.

The tweakey schedule further breaks rotational symmetry. A rotational pair would require the round tweakeys to satisfy compatible rotated relations. However, the schedule uses the same nonlinear layer and injects round-dependent constants

$$RC_r = (C_r, 0, 0, 0),$$

which are not invariant under nonzero rotations. Therefore, we do not identify a standard rotational attack or a simple related-tweak rotational attack against any of the four variants.

C.7 Slide Attacks

Slide attacks exploit self-similarity between round functions [BW99]. In our construction, this symmetry is broken by the round constants injected into the tweakey schedule:

$$TK[r] = U_r \oplus K^{(r)} \oplus RC_r, \qquad RC_r = (C_r, 0, 0, 0),$$

where $K^{(r)} = K$ for the 128-bit-key variants and $K^{(r)}$ alternates between the two key halves for the 256-bit-key variants.

Since the constants C_r are round-dependent, a shifted alignment of round tweakeys would require $RC_r = RC_{r+s}$ for some nonzero shift s over the aligned rounds, which does not hold for the specified constants. For the 256-bit-key variants, such an alignment must additionally respect the alternating key-half pattern. Moreover, the tweakey state is updated through $\mathcal{S}$ and $\mathcal{M}$, making related-tweak slid pairs subject to nonlinear schedule constraints. Therefore, we do not identify a standard slide attack or a simple related-tweak slide attack against any of the four variants.

C.8 Meet-in-the-Middle Attacks

We evaluate meet-in-the-middle attacks following the standard framework based on partial matching [AS09] and initial structures, also known as independent bicliques [SA09]. The 1.5-round full-diffusion property implies that a nonzero state difference induced by varying key material activates all eight 16-bit S-boxes within 1.5 rounds in both the forward and backward directions.

Therefore, the number of rounds available for partial matching is estimated to be up to

$$(1.5 - 1) + (1.5 - 1) + 1 = 2.$$

An initial structure requires the forward and backward key-differential trails not to share active nonlinear components. Since both trails become fully active within 1.5 rounds, an initial structure can cover at most 1.5 rounds at the half-round granularity.

Even assuming that splice-and-cut [AS09] adds another 1.5-round diffusion window in the most favorable case for the attacker, the total structural coverage is estimated as

$$1.5 + 1.5 + 2 = 5$$

rounds within this standard framework.

This estimate ignores the cost of attaching outer operations to the MitM core. At both boundaries, word-wise modular key additions introduce carry-dependent key dependencies, which are immediately mixed by the adjacent linear layers. Moreover, the same master key determines both the whitening values and the round tweakeys. Consequently, outer-round computations cannot be separated into guesses of small independent subkeys, but require correlated whitening-key and round-tweakey material.

Therefore, although the structural estimate reaches the five rounds of Rotom-128*, we do not identify a practical MitM key-recovery attack against it. The 6-round variants Rotom-128 and Rotom-256*, and the 7-round variant Rotom-256, are outside this structural estimate.

D AEAD-Level Comparison

To evaluate Rotom in a practical authenticated-encryption setting, we compare a Rotom-based AEAD scheme with Ascon-AEAD128 and ChaCha20-Poly1305 on Cortex-M3 and Cortex-M4. We first describe the evaluation setting, and then discuss the results.

D.1 Evaluation Setting

Instantiation. We instantiate a variant of ΘCB3 [KR11] with Rotom-128 as the underlying TBC, denoted Rotom-128-ΘCB3. In more detail, a TBC call to process i-th message block in Rotom-128-ΘCB3 takes the nonce N as a tweak. It also derives an offset, i.e., input and output mask, as a function of i and an encryption of a constant, using a Gray code in the same manner as OCB. Rotom-128-ΘCB3 is a mixture of OCB and ΘCB3, which is to ensure fast operation (by eliminating the tweakey computation per block) and sufficiently high security. The security is 128-bit for messages of $\ll 2^{64}$ blocks, which we think is reasonable for microcontrollers.

Comparison Targets. For Ascon-AEAD128 [TMC+25], we use the official C implementation with the bi32_armv7m backend.[4] For ChaCha20-Poly1305 [NL18], we use Weatherley's Arduino Cryptography Library implementation,[5] which is also referenced in NIST IR 8454, Section 4.1.3. The three schemes differ in their parameters: Ascon-AEAD128 uses a 128-bit key and a 128-bit nonce, ChaCha20-Poly1305 uses a 256-bit key and a 96-bit nonce, and Rotom-128-ΘCB3 uses a 128-bit key and a 128-bit nonce.

Measurement Setup. We use the same Cortex-M3 and Cortex-M4 platforms, toolchain, and measurement methodology as in Section 5.1. The mode layer of Rotom-128-ΘCB3 is implemented in C, while each TBC call invokes the non-unrolled assembly implementation of Rotom-128 described in Section 5.1. Each measurement covers from key setup to tag output. We use empty associated data, a 16-byte tag, and message lengths of 16, 128, and 1024 bytes.

D.2 Results

Table 13 reports the execution time in cycles per byte together with the ROM size, and Table 14 reports RAM usage.

Although Rotom-128-ΘCB3 is a generic mode built on a TBC, its performance approaches that of the dedicated AEAD scheme Ascon-AEAD128. Compared with ChaCha20-Poly1305, Rotom-128-ΘCB3 requires 1.6–2.0× fewer cycles for all message lengths on both platforms. Moreover, Rotom-128-ΘCB3 has the smallest ROM footprint of the three schemes. For short messages, it requires more RAM than the other two schemes, whereas its RAM usage grows more slowly with the message length.

[4] https://github.com/ascon/ascon-c
[5] https://github.com/rweather/arduinolibs

Table 13: AEAD encryption performance on Cortex-M3 and Cortex-M4 (empty AD, 16-byte tag). Execution time is given in cycles per byte for each message length, where the cycle count includes key setup, initialization, message processing, and tag generation. ROM is given in bytes.

			Cycles per byte		
Platform	Scheme	ROM	16 B	128 B	1024 B
Cortex-M3	Ascon-AEAD128	13,404	152.1	55.1	43.0
	ChaCha20-Poly1305	3,564	555.1	139.4	99.3
	Rotom-128-ΘCB3	2,748	274.6	77.9	53.3
Cortex-M4	Ascon-AEAD128	13,392	150.8	54.5	42.5
	ChaCha20-Poly1305	3,508	528.0	125.6	86.5
	Rotom-128-ΘCB3	2,716	274.1	77.5	52.9

Table 14: RAM usage (static data and stack) in bytes for AEAD encryption.

	Cortex-M3			Cortex-M4		
Scheme	16 B	128 B	1024 B	16 B	128 B	1024 B
Ascon-AEAD128	284	508	2,300	276	500	2,292
ChaCha20-Poly1305	564	788	2,580	572	796	2,588
Rotom-128-ΘCB3	956	1,068	1,964	956	1,068	1,964