

APEX: AFS-based Permutation family for Efficiency and eXtensibility (feat. the APEX Suite)

Zhiguang Yan  and Yongzhuang Wei 

Guangxi Key Laboratory of Cryptography and Information Security, Guilin University of Electronic Technology, Guilin, 541004, China.

E-mail(s): zhiguang_yan@163.com; walker_wyz@guet.edu.cn;

Abstract

ARX-based cryptographic primitives have received considerable attention for their efficient software implementations. For a long time, however, constructing ARX primitives with provable resistance against single-trail differential and linear cryptanalysis remained an open problem. Dinu et al. addressed this problem by introducing the Long Trail Strategy (LTS), the first general design strategy for establishing such bounds for ARX symmetric-key primitives. A remaining challenge in applying LTS is the systematic design and analysis of large-state S-boxes that combine efficient implementation with strong multi-iteration differential and linear bounds. More recently, Yan et al. introduced a general framework for designing and analyzing such S-boxes, with the AFS family serving as a concrete instantiation. Given their excellent multi-iteration security bounds and outstanding software implementation efficiency, AFS boxes offer a viable solution to the core challenging problem in ARX cryptography—systematic design, accurate analysis, and the pursuit of an extreme and compact balance between cryptographic security and implementation performance. We thus argue that the potential of AFS boxes as nonlinear components in LTS-based primitives has long been undervalued. To demonstrate this potential and address the broader design challenge, we use AFS-64 to construct APEX, an efficient and extensible family of cryptographic permutations spanning state widths from 64 to 1536 bits.

We then instantiate APEX in a broad range of symmetric primitives to demonstrate its extensibility and translate the security and implementation advantages of AFS into complete cryptographic designs. These include small-state hash functions and extendable-output functions, authenticated encryption with associated data (AEAD) schemes, an ultralightweight block cipher, large-state block and tweakable block ciphers, and large-state hash functions based on the Sponge-F mode and designed for China’s Next-Generation Commercial Cryptographic Algorithms program. We derive differential and linear long-trail bounds for the underlying permutations and adapt the long-trail analysis to rate-restricted, same-capacity, and related-tweak settings. Together with analyses of other major attack classes, these results support the selected step counts and the stated security claims.

Optimized implementations on 8-bit AVR, 32-bit ARMv7-M, and x86-64 demonstrate the practical software efficiency of APEX across diverse processor architectures. For 64-byte (resp. 1536-byte) messages, the small-state APEX-HASH functions achieve $1.14\text{--}1.37\times$ (resp. $1.14\text{--}1.17\times$) and $1.15\text{--}1.16\times$ (resp. $1.18\text{--}1.20\times$) the throughput of the corresponding Esch instances on AVR and ARM, respectively. The small-state APEX-AEAD schemes similarly achieve $1.17\text{--}1.48\times$ (resp. $1.14\text{--}1.19\times$) and $1.15\text{--}1.19\times$ (resp. $1.12\text{--}1.15\times$) the throughput of the corresponding Schwaemm instances. For the large-state hash functions targeting the NGCC program, APEX₁₅₃₆¹²-HASH-F-512 achieves $2.09\times$ and $2.88\times$ the throughput of the fastest listed SHA3-512 implementations on x86-64 and ARM, respectively, for a 1-MiB message. The higher-security APEX₁₅₃₆¹⁶-HASH-F-768 and APEX₁₅₃₆²⁰-HASH-F-1024 profiles achieve 6.02 and 11.82 cycles/byte on x86-64, and 123.72 and 239.63 cycles/byte on ARM, respectively. For block-cipher applications, the ultralightweight APEX₆₄⁸-BC-128 achieves encryption and decryption speedups of $1.18\text{--}1.21\times$ over CRAX-S across AVR and ARM, while the large-state APEX₂₅₆¹⁵-BC-256 achieves $1.77\times$ and $1.65\times$ speedups over SATURNIN-256/256 for encryption and decryption on ARM, respectively. The tweakable block cipher APEX₂₅₆¹⁵-TBC-256/128 achieves $1.38\times$ and $1.39\times$ speedups over TRAX-L for encryption and decryption, respectively. Taken together, these results show that APEX combines extensibility across state sizes and primitive classes with efficient software implementations on markedly different processor architectures.

Keywords: AFS-64; APEX; cryptographic permutations; Long Trail Strategy; hash functions; authenticated encryption with associated data; lightweight block ciphers; tweakable block ciphers; Next-Generation Commercial Cryptographic Algorithms

Contents

1	Introduction	3
2	Notation and Conventions	5
3	The APEX Permutation Family	6
3.1	Specification	6
3.2	Design Rationale	9
3.3	Security Analysis of the APEX Permutation Family	13
3.4	Implementation Considerations	18
4	Naming Convention for APEX-Based Constructions	19
5	Small-State Hashing and AEAD	19
5.1	Specification	19
5.2	Design Rationale	32
5.3	Security Analysis	36
5.4	Implementation	39
6	Large-State Hash Functions for the Next-Generation Commercial Cryptographic Algorithms Program	44
6.1	Specification	44
6.2	Design Rationale and Security Analysis	47
6.3	Implementation	50
7	Block Ciphers and Tweakable Block Ciphers	51
7.1	Specification	51
7.2	Design Rationale and Security Analysis	52
7.3	Implementation	54
8	Conclusion	55
A	Details of integral cryptanalysis of the APEX permutations	56
B	Details of impossible-differential and zero-correlation linear cryptanalysis of the APEX permutations	57
B.1	Impossible differentials	57
B.2	Zero-correlation linear cryptanalysis	57

1 Introduction

Cryptographic primitives based on the ARX (addition–rotation–XOR) paradigm have been extensively studied because of their software efficiency. By relying primarily on modular addition, bit rotation, and XOR, ARX designs can be implemented using simple word-level instructions that map naturally to general-purpose processors. This makes them particularly attractive for software-oriented cryptographic applications.

The appeal of this approach predates the modern use of the term ARX. Early ARX-like designs, such as TEA [1] and XTEA [2], proposed by Wheeler and Needham, explored compact software implementations of block ciphers using modular addition, shifts, and XOR. ARX structures were subsequently adopted in stream ciphers. Salsa20 [3], proposed by Bernstein for the eSTREAM project, provided a high-speed software-oriented design built entirely from ARX operations. ChaCha [4], introduced as a variant of Salsa20, improved its diffusion while retaining excellent software performance. The resulting ChaCha family has since been widely deployed, most notably through ChaCha20-Poly1305, which combines ChaCha20 [4] with the Poly1305 authenticator [5] in an AEAD construction specified by the Crypto Forum Research Group (CFRG) in RFC 8439 [6] and adopted by the Internet Engineering Task Force (IETF) for use in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) [7], TLS 1.3 [8], QUIC [9], and Internet Key Exchange version 2 (IKEv2)/Internet Protocol Security (IPsec) [10].

The same implementation advantages have also made ARX structures prominent in lightweight cryptography. Introduced by the U.S. National Security Agency (NSA), the SPECK family [11] comprises lightweight ARX block ciphers with a range of block and key sizes designed for efficient software implementation on resource-constrained platforms. LEA [12] was designed as a high-speed software-oriented ARX block cipher and was subsequently standardized as Korean Industrial Standard KS X 3246 [13]. More recently, Ballet [14], a lightweight block cipher based on a simplified Lai–Massey [15] structure and an ARX round function, was selected as one of the final winners in the block-cipher category of the 2018–2020 Cryptographic Algorithm Design Competition organized by the Chinese Association for Cryptologic Research (CACR) [16].

The use of ARX is not limited to block and stream ciphers. ARX operations have also been employed in hash functions and large-state cryptographic constructions. BLAKE [17], whose compression function uses a ChaCha-inspired ARX core, advanced to the final round of the SHA-3 competition [18]. Its successor BLAKE2 [19] further simplified and optimized the design for software implementation and has been widely deployed in cryptographic libraries. The Skein hash family [20] is built around Threefish, a large-state tweakable block cipher whose MIX operation combines modular addition, rotation, and XOR. These designs illustrate that the software advantages of ARX extend naturally from compact ciphers to large-state and tweakable constructions.

This implementation simplicity, however, does not translate directly into simple security arguments. Carry propagation in modular addition couples different bit positions, and a large word-oriented ARX component does not naturally decompose into small S-boxes whose local differential and linear properties can be combined with lower bounds on the number of active S-boxes, as in the Wide Trail Strategy (WTS) [21]. Consequently, differential and linear evaluations of ARX designs have historically relied heavily on computer-assisted searches for reduced-round characteristics. Deriving strong and provable single-trail differential and linear bounds for a complete ARX construction remains considerably more difficult. The Long Trail Strategy (LTS), introduced by Dinu et al. [22], was developed to address this difficulty.

The original LTS paper considered two candidate ARX-boxes, SPECKEY, adapted from the SPECK-32 [11] round function, and MARX-2, built from the Skein [20] MIX operation. Motivated by SPECKEY’s lower operation count and better implementation properties, particularly on 16-bit platforms, the authors selected it as the large nonlinear component and used LTS to construct SPARX, the first lightweight ARX block-cipher family with provable bounds against single-trail differential and linear cryptanalysis [22]. MARX-2 was later used as the nonlinear component of LiARX [23], a lightweight block cipher designed according to the same LTS principle. The same general design idea also appears in the sLiSCP [24] and sLiSCP-light [25] permutation families, whose nonlinear component is a reduced-round Simeck [26] instance. In particular, sLiSCP-light underlies SPIX [27] and SPOC [28], two candidates in the Lightweight Cryptography standardization process of the U.S. National Institute of Standards and Technology (NIST), both of which advanced to the second round [29].

Building on the LTS framework, Beierle et al. introduced the compact 64-bit ARX-box Alzette [30] at CRYPTO 2020. Its structure is SPECK-like, but it uses different rotation constants and places the rotations differently. These changes strengthen its multi-iteration bounds without giving up efficient software implementation. Alzette is used directly in the CRAX [30] block cipher and the TRAX [30] tweakable block cipher, and it is the nonlinear component of the SPARKLE [31] permutation family. SPARKLE in turn underlies the Schwaemm [31] family of AEAD schemes and

the Esch [31] family of hash functions. NIST advanced the submission SPARKLE (SCHWAEMM and ESCH) to the final round of its Lightweight Cryptography standardization process [29]. Although NIST ultimately selected the Ascon family [32] for standardization, the SPARKLE submission demonstrated highly competitive software performance. NIST’s final report [33] notes that, in Weatherley’s size-optimized microcontroller benchmarks, SPARKLE achieved the greatest speedups over ChaCha20-Poly1305 on the 8-bit AVR platform. On the 32-bit ESP32, it was the only finalist faster than ChaCha20-Poly1305 in all four tested AEAD cases, while on the 32-bit ARM Cortex-M3, SPARKLE and Xoodyak [34] were the only finalist hash functions faster than BLAKE2s [19].

In 2023, Fan et al. [35] proposed another 64-bit ARX-based large-state S-box. However, its reported multi-iteration differential and linear bounds were weaker than those of Alzette, illustrating the difficulty of designing large ARX-boxes with strong security bounds across multiple iterations.

Despite these developments, large-state S-boxes for LTS were still designed and analyzed largely on an instance-by-instance basis. Yan et al. [36] recently proposed a systematic framework for designing and analyzing such S-boxes, which is not restricted to ARX constructions. The framework is accompanied by a bit-level model for optimal long-trail decomposition and the SMCS search technique for deriving tighter multi-iteration differential and linear bounds. Applied to existing ARX designs, these tools improve the known bounds for Alzette and yield tighter bounds for SPARKLE.

As one concrete instantiation of this general framework, Yan et al. introduced the ARX-Feistel Structure (AFS) [36] family of large-state ARX-boxes. The AFS instances combine inexpensive software operations with strong multi-iteration differential and linear properties.

Among these instances, AFS-64 is a natural building block for an extensible software-oriented design. It uses only 32-bit modular additions, XORs, and fixed rotations, while providing stronger trail bounds than Alzette. For one iteration, its maximum expected differential characteristic probability (MEDCP) and maximum expected linear characteristic correlation (MELCC) are bounded by 2^{-10} and 2^{-6} , respectively; the corresponding bounds for Alzette are 2^{-6} and 2^{-2} . Over two iterations, the bounds become 2^{-37} and 2^{-20} for AFS-64, versus 2^{-34} and 2^{-17} for Alzette [36].

We believe that the value of the AFS family for LTS-based design has not yet been fully recognized by the cryptographic community, and that its usefulness as a building block for efficient symmetric primitives has consequently been underestimated. In this work, we demonstrate this potential using AFS-64, the 64-bit instance of the family. Combining AFS-64 with LTS, we construct APEX, an efficient and extensible family of cryptographic permutations. To demonstrate this extensibility, we use APEX to construct a broad range of symmetric primitives spanning different applications and security requirements.

Our Contributions

Our contributions are summarized as follows.

- An efficient and extensible family of cryptographic permutations, APEX, is introduced based on the large-state ARX-box AFS-64. Guided by the Long Trail Strategy, APEX combines parallel AFS-64 layers with a width-extensible sparse diffusion layer and supports state widths from 64 to 1536 bits. Its common structure provides a unified foundation for symmetric primitives with different security and performance requirements. The strong multi-iteration differential and linear bounds of AFS-64 also provide a concrete basis for analyzing the permutations and selecting their step counts.
- A broad range of symmetric primitives is constructed using APEX, demonstrating the extensibility of the permutation family across diverse applications. These constructions include small-state hash and extendable-output functions, AEAD schemes, an ultralightweight block cipher, large-state block and tweakable block ciphers, and large-state hash functions based on the Sponge-F mode [37] and designed for China’s Next-Generation Commercial Cryptographic Algorithms program [38]. They cover different state sizes, security levels, and application settings while retaining a common underlying design.
- An extensive security analysis of the APEX permutations and their modes of use is provided. A method for selecting the round constants of AFS-64 is proposed by combining fixed-key quasidifferential analysis [39] with empirical fixed-key differential and linear experiments. Differential and linear long-trail bounds are derived for the full permutations, and the corresponding searches are adapted to mode-specific constraints, including rate-restricted endpoints, equal-capacity endpoint constraints, and related-tweak differences. The permutations are further evaluated against a broad range of other cryptanalytic techniques. Together, these results support the round-constant selection, the selected step counts, and the stated security claims.

- Reference and optimized software implementations are provided for 8-bit AVR, 32-bit ARMv7-M, and x86-64 platforms. Their performance is compared with that of representative lightweight and high-performance designs, including Esch, Schwaemm, SHA3-512, CRAX-S, SATURNIN-256/256, and TRAX-L. The results demonstrate the practical efficiency of the APEX-based constructions across different primitive classes and software environments.

Table 1 summarizes selected software performance comparisons. Complete cycle counts, code sizes, measurement details, and results for all instances are reported in Sections 5.4, 6.3, and 7.3.

Table 1 Selected software-performance comparisons for the APEX-based constructions. Each value gives the throughput of the APEX construction relative to the indicated reference; values above one denote higher APEX throughput. Where multiple platforms are listed, the values are given in the same order.

Primitive type	APEX construction	Reference	Workload	Platform	Relative throughput
Small-state hash	APEX ₃₈₄ ^(6,9) -HASH-256	ESCH256	64-byte message	AVR / ARM	1.37× / 1.15×
			1536-byte message	AVR / ARM	1.17× / 1.20×
	APEX ₅₁₂ ^(7,10) -HASH-384	ESCH384	64-byte message	AVR / ARM	1.14× / 1.16×
			1536-byte message	AVR / ARM	1.14× / 1.18×
AEAD	APEX ₂₅₆ ^(6,8) -AEAD-128/128	SCHWAEMM128-128	64-byte message	AVR / ARM	1.48× / 1.15×
			1536-byte message	AVR / ARM	1.14× / 1.12×
	APEX ₃₈₄ ^(6,9) -AEAD-256/128	SCHWAEMM256-128	64-byte message	AVR / ARM	1.19× / 1.17×
			1536-byte message	AVR / ARM	1.17× / 1.12×
	APEX ₃₈₄ ^(6,9) -AEAD-192/192	SCHWAEMM192-192	64-byte message	AVR / ARM	1.19× / 1.16×
			1536-byte message	AVR / ARM	1.19× / 1.15×
Large-state hash	APEX ₁₅₃₆ ^(7,10) -AEAD-256/256	SCHWAEMM256-256	64-byte message	AVR / ARM	1.17× / 1.19×
			1536-byte message	AVR / ARM	1.14× / 1.14×
Block cipher	APEX ₁₂ ¹² -HASH-F-512	SHA3-512	1-MiB message	x86-64 / ARM	2.09× / 2.88×
	APEX ₆₄ ⁸ -BC-128	CRAX-S	One 64-bit block	AVR / ARM	Enc.: 1.21× / 1.20× Dec.: 1.20× / 1.18×
	APEX ₂₅₆ ¹⁵ -BC-256	SATURNIN-256/256	One 256-bit block	ARM	Enc.: 1.77× Dec.: 1.65×
Tweakable block cipher	APEX ₂₅₆ ¹⁵ -TBC-256/128	TRAX-L	One 256-bit block	ARM	Enc.: 1.38× Dec.: 1.39×

2 Notation and Conventions

For nonnegative integers a and b , we identify $\mathbb{F}_2^{a+b}$ with $\mathbb{F}_2^a \times \mathbb{F}_2^b$. Elements are written as tuples when their components are considered separately and as concatenated bit strings otherwise. The empty bit string is denoted by ϵ . The byte order is assumed to be little-endian. We use $\boxplus$ for addition modulo 2^{32} and $\oplus$ for bitwise XOR of equal-length strings. The operators $\ll$ and $\gg$ denote logical left and right shifts, respectively, whereas $\lll$ and $\ggg$ denote left and right rotations. The accompanying integer specifies the number of bit positions. For $u \in \mathbb{F}_2^n$, $u[i]$ denotes its bit at index i , numbered from the least significant bit; thus $u[0]$ is the least significant bit and $u = \sum_{i=0}^{n-1} u[i]2^i$ when interpreted as a nonnegative integer. For $u, v \in \mathbb{F}_2^n$, the binary inner product is denoted by $\langle u, v \rangle = \bigoplus_{j=0}^{n-1} u[j]v[j] \in \mathbb{F}_2$.

3 The APEX Permutation Family

3.1 Specification

3.1.1 Family Overview

The APEX (*AFS-based Permutation for Efficiency and eXtensibility*) family consists of permutations $\text{APEX}_b^{n_s} : \mathbb{F}_2^b \rightarrow \mathbb{F}_2^b$, where b denotes the block size in bits and $n_s \in \mathbb{N}_{>0}$ denotes the number of steps. We denote by n_b the number of 64-bit branches; hence, $n_b = b/64$. In this work, we consider $b \in \{64, 128, 256, 384, \dots, 1536\}$, or equivalently, $n_b \in \{1, 2, 4, 6, \dots, 24\}$. Depending on the block size, the APEX permutations are defined in three cases:

1. For $b = 64$, $\text{APEX}_{64}^{n_s}$ is obtained by iterating AFS-64 for n_s steps. The construction is specified in Algorithm 3(a) and illustrated in Figure 1.
2. For $b = 128$, $\text{APEX}_{128}^{n_s}$ follows the classical SPN structure. Each step applies AFS-64 in parallel to the two branches under different, branch-dependent constants c_i , followed by a simple XOR-based two-branch Feistel transformation. The construction is specified in Algorithm 3(b) and illustrated in Figure 2.
3. For $b \in \{256, 384, \dots, 1536\}$, $\text{APEX}_b^{n_s}$ uses the same SPN structure at a scalable width. Each step applies AFS-64 in parallel to the n_b branches under different, branch-dependent constants c_i , followed by the scalable rotating Feistel transformation $\mathcal{L}_{n_b}$ defined in Section 3.1.3. The construction is specified in Algorithm 3(c) and illustrated in Figure 3.

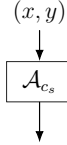


Fig. 1 One step of $\text{APEX}_{64}^{n_s}$ at step s , where the constant index is reduced modulo 24.

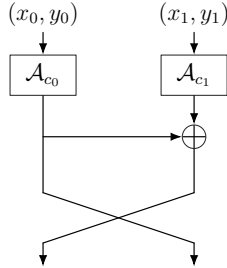


Fig. 2 One step of $\text{APEX}_{128}^{n_s}$. Step-separation injections are omitted for clarity.

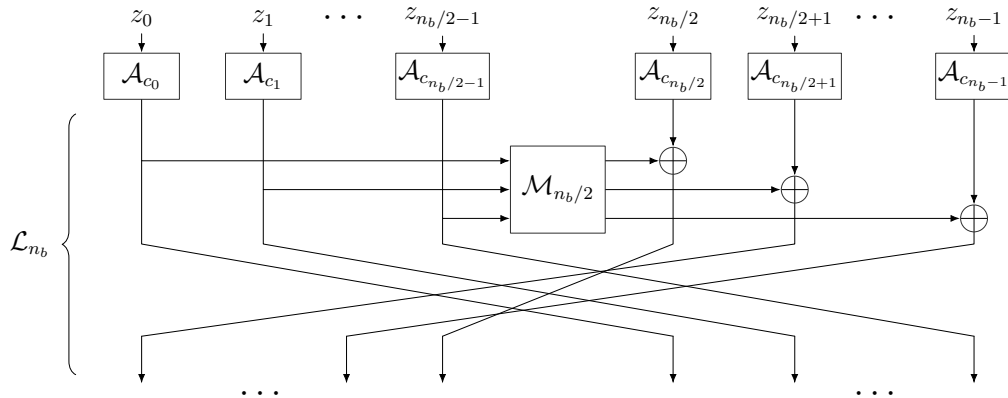


Fig. 3 The overall structure of one step of $\text{APEX}_{64n_b}^{n_s}$ for even $n_b \geq 4$. Each $z_i = (x_i, y_i)$ denotes a 64-bit branch. Step-separation injections are omitted for clarity.

3.1.2 The AFS-64 ARX-box

AFS (ARX-Feistel Structure) is a family of large-state ARX-based S-boxes introduced in [36]. An AFS- $2n$ instance operates on a pair of n -bit words and is specified by a configurational sequence, a rotation sequence, and round constants.

In this work, we adopt AFS-64, shortly denoted by $\mathcal{A}$, with configurational sequence 11000011 and rotation sequence (17, 24, 1, 1, 16, 31, 24, 0). Its instantiation with a 32-bit round constant c is denoted by $\mathcal{A}_c$. The instance consists of four internal rounds, each comprising two alternating Feistel-like word updates, for a total of eight updates. A configurational bit equal to 1 selects modular addition, whereas a bit equal to 0 selects XOR; the corresponding entry of the rotation sequence determines the rotation amount. The constant c is XORed into the updated word after every modular-addition update. Algorithm 1 gives the complete specification of $\mathcal{A}_c$, and Fig. 4 illustrates its update schedule.

Algorithm 1 The 64-bit ARX-box $\mathcal{A}_c$.

Input/Output: $(x, y) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$

```

 $x \leftarrow x \boxplus (y \ggg 17)$ 
 $x \leftarrow x \oplus c$ 
 $y \leftarrow y \boxplus (x \ggg 24)$ 
 $y \leftarrow y \oplus c$ 
 $x \leftarrow x \oplus (y \ggg 1)$ 
 $y \leftarrow y \oplus (x \ggg 1)$ 
 $x \leftarrow x \oplus (y \ggg 16)$ 
 $y \leftarrow y \oplus (x \ggg 31)$ 
 $x \leftarrow x \boxplus (y \ggg 24)$ 
 $x \leftarrow x \oplus c$ 
 $y \leftarrow y \boxplus (x \ggg 0)$ 
 $y \leftarrow y \oplus c$ 
return  $(x, y)$ 

```

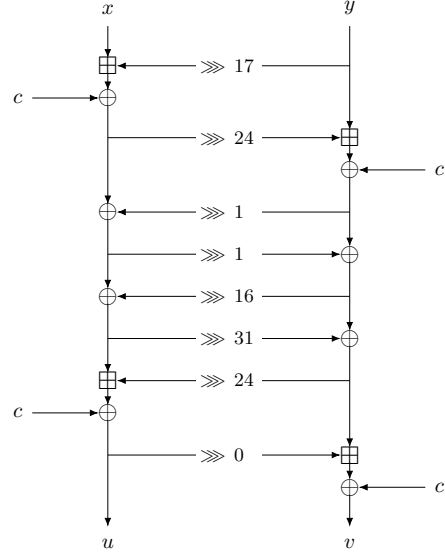


Fig. 4 The AFS-64 instance $\mathcal{A}_c$.

3.1.3 The Linear Layer $\mathcal{L}_{n_b}$

The linear layer $\mathcal{L}_{n_b}$ follows the diffusion-layer structure of SPARKLE [31], which in turn draws heavily from SPARKX-128 [22]. It is a Feistel round on n_b 64-bit branches with the linear Feistel mixing function $\mathcal{M}_{n_b/2}$. APEX retains the Feistel structure, one-branch rotation, and branch swap of the SPARKLE linear layer, while extending its concrete instantiation from the 4-, 6-, and 8-branch instances of SPARKLE to every even n_b satisfying $4 \leq n_b \leq 24$.

Following the notation of SPARKLE, we denote the linear Feistel mixing function by $\mathcal{M}_{n_b/2}$ and the complete linear layer by $\mathcal{L}_{n_b}$. Let the input state of $\mathcal{L}_{n_b}$ be written as $L \parallel R$, where each half contains $n_b/2$ branches and $L = ((x_0, y_0), (x_1, y_1), \dots, (x_{n_b/2-1}, y_{n_b/2-1})) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^{n_b/2}$, and define the 32-bit linear map

$$\ell(z) = (z \oplus (z \ll 16)) \ggg 16 \quad (1)$$

and the two global mixing words

$$t_x = \ell \left(\bigoplus_{i=0}^{n_b/2-1} x_i \right), \quad t_y = \ell \left(\bigoplus_{i=0}^{n_b/2-1} y_i \right). \quad (2)$$

The linear Feistel mixing function $\mathcal{M}_{n_b/2}$ is then given by

$$\mathcal{M}_{n_b/2}(L) = ((x_0 \oplus t_y, y_0 \oplus t_x), \dots, (x_{n_b/2-1} \oplus t_y, y_{n_b/2-1} \oplus t_x)). \quad (3)$$

Let ρ denote a cyclic left rotation by one branch. The complete linear layer can therefore be written compactly as

$$\mathcal{L}_{n_b}(L \parallel R) = \rho(R \oplus \mathcal{M}_{n_b/2}(L)) \parallel L, \quad (4)$$

where the XOR between branch vectors is componentwise. The resulting linear layer is illustrated as part of a complete step in Figure 3. Algorithm 2 gives the corresponding simultaneous assignments explicitly.

Algorithm 2 The linear layer $\mathcal{L}_{n_b}$.

Input/Output: $((x_0, y_0), (x_1, y_1), \dots, (x_{n_b-1}, y_{n_b-1})) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^{n_b}$, where $n_b \geq 4$ is even

▷ Feistel round

```

1:  $t_x \leftarrow \bigoplus_{i=0}^{n_b/2-1} x_i$ 
2:  $t_y \leftarrow \bigoplus_{i=0}^{n_b/2-1} y_i$ 
3:  $(t_x, t_y) \leftarrow ((t_x \oplus (t_x \ll 16)) \ggg 16, (t_y \oplus (t_y \ll 16)) \ggg 16)$ 
4: for all  $i \in [0, n_b/2 - 1]$  do
5:    $x_{i+n_b/2} \leftarrow x_{i+n_b/2} \oplus x_i \oplus t_y$ 
6:    $y_{i+n_b/2} \leftarrow y_{i+n_b/2} \oplus y_i \oplus t_x$ 
7: end for

```

▷ Branch permutation

```

8:  $(x_0, x_1, \dots, x_{n_b-1}) \leftarrow (x_{n_b/2+1}, x_{n_b/2+2}, \dots, x_{n_b-1}, x_{n_b/2}, x_0, x_1, \dots, x_{n_b/2-1})$ 
9:  $(y_0, y_1, \dots, y_{n_b-1}) \leftarrow (y_{n_b/2+1}, y_{n_b/2+2}, \dots, y_{n_b-1}, y_{n_b/2}, y_0, y_1, \dots, y_{n_b/2-1})$ 
10: return  $((x_0, y_0), (x_1, y_1), \dots, (x_{n_b-1}, y_{n_b-1}))$ 

```

Algorithm 3 The APEX permutation family.

Round constants: $c_0, c_1, \dots, c_{23} \in \mathbb{F}_2^{32}$

```

 $(c_0, c_1, c_2, c_3) \leftarrow (0x7B3F3442, 0x80BEC09D, 0xB3BE9BC4, 0x083F2CC1)$ 
 $(c_4, c_5, c_6, c_7) \leftarrow (0x4CBF5ED1, 0xF0BFC210, 0x0CBF0D90, 0x633E4BDE)$ 
 $(c_8, c_9, c_{10}, c_{11}) \leftarrow (0xE0BF8E96, 0x3CBF0DC8, 0xAF3E972D, 0x473F9712)$ 
 $(c_{12}, c_{13}, c_{14}, c_{15}) \leftarrow (0x133EC79D, 0x0F3E2E2A, 0xA0BF57B5, 0x743EC3E9)$ 
 $(c_{16}, c_{17}, c_{18}, c_{19}) \leftarrow (0xCBBF9791, 0x643FE36A, 0x533F3FAA, 0xD83EC2A0)$ 
 $(c_{20}, c_{21}, c_{22}, c_{23}) \leftarrow (0xAC3EE996, 0xEFBF8244, 0xC4BFE81B, 0x30BEA52B)$ 

```

(a) APEX$_{64}^{n_s}$. <i>Input/Output:</i> $(x_0, y_0) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$ <pre> 1: for $s \in [0, n_s - 1]$ do 2: $(x_0, y_0) \leftarrow \mathcal{A}_{c_s \bmod 24}(x_0, y_0)$ 3: end for 4: return (x_0, y_0) </pre>	(b) APEX$_{128}^{n_s}$. <i>Input/Output:</i> $((x_0, y_0), (x_1, y_1)) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^2$ <pre> 1: for $s \in [0, n_s - 1]$ do 2: $y_0 \leftarrow y_0 \oplus c_s \bmod 24$ 3: $y_1 \leftarrow y_1 \oplus (s \bmod 2^{32})$ 4: for all $i \in [0, 1]$ do 5: $(x_i, y_i) \leftarrow \mathcal{A}_{c_i}(x_i, y_i)$ 6: end for 7: $((x_0, y_0), (x_1, y_1)) \leftarrow ((x_1 \oplus x_0, y_1 \oplus y_0), (x_0, y_0))$ 8: end for 9: return $((x_0, y_0), (x_1, y_1))$ </pre>
---	--

(c) APEX $_b^{n_s}$.

Input/Output: $((x_0, y_0), (x_1, y_1), \dots, (x_{n_b-1}, y_{n_b-1})) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^{n_b}$, where $b = 64n_b$ and $4 \leq n_b \leq 24$ is even

```

1: for  $s \in [0, n_s - 1]$  do
2:    $y_0 \leftarrow y_0 \oplus c_s \bmod 24$ 
3:    $y_1 \leftarrow y_1 \oplus (s \bmod 2^{32})$ 
4:   for all  $i \in [0, n_b - 1]$  do
5:      $(x_i, y_i) \leftarrow \mathcal{A}_{c_i}(x_i, y_i)$ 
6:   end for
7:    $((x_0, y_0), (x_1, y_1), \dots, (x_{n_b-1}, y_{n_b-1})) \leftarrow \mathcal{L}_{n_b}((x_0, y_0), (x_1, y_1), \dots, (x_{n_b-1}, y_{n_b-1}))$ 
8: end for
9: return  $((x_0, y_0), (x_1, y_1), \dots, (x_{n_b-1}, y_{n_b-1}))$ 

```

3.1.4 Round Constants

The original AFS study [36] instantiated AFS-64 using the same eight round constants as Alzette [30]. Since an APEX permutation may contain up to 24 parallel AFS-64 instances, we select 24 distinct constants $c_0, \dots, c_{23} \in \mathbb{F}_2^{32}$. They serve as branch-dependent constants inside the AFS-64 layer, and the same table is reused for step-constant injection.

The constants are transparently derived from the hexadecimal fractional expansion of π . Their values are listed in Algorithm 3, and the detailed selection criteria are described in Section 3.2.

3.2 Design Rationale

3.2.1 The Long Trail Strategy

The Long Trail Strategy (LTS), introduced in the design of SPARX [22] and subsequently applied to SPARKLE [31], is a design approach for deriving differential and linear bounds for ARX-based primitives with large internal states. Direct computer-assisted analysis of the complete state rapidly becomes infeasible as its width increases. The LTS therefore organizes a primitive so that bounds established for a smaller nonlinear component can be lifted to the complete construction.

The long-trail argument. To apply the LTS, we require two components:

- a nonlinear operation $\mathcal{A}$ operating on ω bits such that the differential probabilities and absolute linear correlations of multiple consecutive applications of $\mathcal{A}$ are bounded; and
- a linear layer operating on ω -bit branches.

To derive a bound for the complete construction, we enumerate the truncated differential or linear trails allowed by the linear layer. Each such trail is processed in two stages: first, it is decomposed into *long trails*; second, the corresponding multi-iteration bounds of $\mathcal{A}$ are applied to the resulting decomposition. A long trail of length t is a continuous chain through t consecutive applications of $\mathcal{A}$ that receives no nonzero difference or mask contribution from outside.

Let Γ be an admissible truncated trail, and let $N_t(\Gamma)$ denote the number of long trails of length t in its decomposition. Let p_t be an upper bound on the differential probability of t consecutive applications of $\mathcal{A}$, and let λ_t be the corresponding upper bound on the absolute linear correlation. Let $\text{DP}(\tau_D)$ denote the differential probability of a differential trail τ_D , and let $\text{Cor}(\tau_L)$ denote the correlation of a linear trail τ_L . For every pair of such trails compatible with Γ , we then have

$$\text{DP}(\tau_D) \leq \prod_{t \geq 1} p_t^{N_t(\Gamma)}, \quad |\text{Cor}(\tau_L)| \leq \prod_{t \geq 1} \lambda_t^{N_t(\Gamma)}.$$

Example 1. Following the standard example introduced in the specification of SPARX [22] and later reproduced for SPARKLE [31], consider three consecutive steps of a two-branch construction, followed by one additional parallel application of $\mathcal{A}$ without a subsequent linear layer. The truncated differential trail

$$(\Delta_0^L, \Delta_0^R) \rightarrow (\Delta_1^L, \Delta_1^R) \rightarrow (0, \Delta_2^R) \rightarrow (\Delta_3^L, 0)$$

is depicted in Figure 5. It decomposes into three long trails: the green trail has length one and starts from Δ_0^R ; the red trail has length two and starts from Δ_0^L ; and the blue trail has length three and starts from Δ_1^L . Hence $N_1(\Gamma) = N_2(\Gamma) = N_3(\Gamma) = 1$.

LTS versus WTS. The LTS and the Wide Trail Strategy (WTS) both separate the analysis of a local nonlinear component from the propagation imposed by the complete linear layer, but they use this separation differently.

Bound derivation. The WTS typically assigns a uniform bound to each active small S-box and derives a global bound from a lower bound on the number of active S-boxes. If a S-boxes are active and each has differential

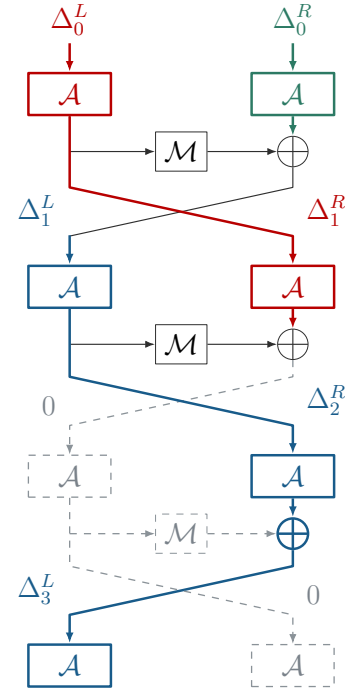


Fig. 5 An example of a long-trail decomposition in an SPN with a sparse Feistel-type linear layer. The green, red, and blue solid paths form long trails of lengths one, two, and three, respectively, while gray dashed paths denote zero differences.

probability at most p , the resulting trail probability is at most p^a . The LTS instead enumerates admissible truncated trails, decomposes each of them into long trails, and uses the multi-iteration bounds p_t and λ_t of the larger nonlinear component. Consequently, two truncated trails containing the same total number of active branches may yield different LTS bounds because their active branches are connected into chains of different lengths.

Example 1 illustrates this distinction; we discuss the differential case, and the linear case is analogous. Since its decomposition satisfies $N_1(\Gamma) = N_2(\Gamma) = N_3(\Gamma) = 1$, the LTS gives

$$\text{DP}(\tau_D) \leq p_1 p_2 p_3.$$

A WTS-style argument instead counts the six active applications of $\mathcal{A}$ and gives $\text{DP}(\tau_D) \leq p_1^6$. The LTS exploits a compatibility effect between consecutive applications of the ARX-box. A differential characteristic attaining the one-iteration bound p_1 may end in an output difference from which no characteristic attaining p_1 is available in the next application. Consequently, the multi-iteration bound may satisfy $p_t < p_1^t$, often by a substantial margin. The LTS bound $p_1 p_2 p_3$ in Example 1 can therefore be much smaller than the WTS-style bound p_1^6 . This phenomenon occurs for AFS-64: Table 2 gives $p_1 = 2^{-10}$, $p_2 = 2^{-37}$, and $p_3 \leq 2^{-66}$, and hence

$$p_1 p_2 p_3 \leq 2^{-10-37-66} = 2^{-113} < p_1^6 = 2^{-60}.$$

The two probability bounds therefore differ by a factor of 2^{53} . The tighter LTS estimate retains the compatibility constraints along each active chain that are omitted by an active-branch count alone.

Different design priorities. The WTS and the LTS place their principal design effort on different components. In the WTS, much of the effort is devoted to constructing a highly diffusive linear layer that guarantees a large minimum number of active small S-boxes. In the LTS, the principal nonlinear-design task is instead to construct an efficient large S-box with strong multi-iteration properties and to establish tight differential and linear bounds for several consecutive applications. Yan et al. [36] give a general framework for this task, instantiate it with the AFS family of large ARX-boxes, and introduce SMCS (SMT–MILP Convergence Search) to derive tighter bounds over more iterations of Alzette. Accordingly, the LTS does not require the linear layer to maximize immediate diffusion. It instead favors a sparse Feistel structure with more gradual per-step diffusion: the copied half preserves long trails, while the mixed half spreads activity across branches.

Two ingredients of an LTS argument. A sound LTS security argument depends on two central ingredients. First, the primitive structure must admit a valid long-trail decomposition, and tight multi-iteration differential and linear bounds for its ARX-box must be established. Second, among all admissible truncated trails, the analysis must find the optimal long-trail decomposition, namely the one yielding the largest admissible differential probability or absolute linear correlation (equivalently, the minimum weight). Efficiently determining this optimum is the main global search problem (for example, the word-level methods used in the SPARX and Alzette analyses [22, 30] were extended in the AFS study [36], which first introduced a bit-level model for long-trail decomposition. The model can be readily encoded using MILP, SAT/SMT, or CP. Its bit-level representation enables a finer-grained long-trail decomposition and removes the laborious prerequisite of manually characterizing the activity patterns admitted by the linear-layer functions). In APEX, $\mathcal{A}_c$ supplies the measured multi-iteration bounds, while the sparse Feistel layer $\mathcal{L}_{n_b}$ determines the admissible decompositions of the multi-branch permutations.

3.2.2 Choice and Properties of AFS-64

A detailed design rationale and security analysis of the AFS family are provided in [36]. Below, we briefly summarize its most important properties. AFS-64 is constructed from XOR-of-rotation and addition-of-rotation operations, namely $x \oplus (y \ggg s)$ and $x \boxplus (y \ggg r)$. These operations can be implemented efficiently using shifted-register instructions on 32-bit ARM processors and therefore provide high diffusion per instruction. The configurational and rotation sequences were selected to optimize differential and linear security subject to implementation-cost constraints. Although all constant rotations have essentially the same cost on 32-bit ARM processors, their costs differ on 8- and 16-bit microcontrollers. The selected rotation sequence therefore also favors efficient implementations on these smaller platforms.

Differential and linear properties. The bounds on the maximum expected differential characteristic probability (MEDCP) and the maximum expected linear characteristic correlation (MELCC) are taken from [36]. In that work, the automated SMCS technique was used to analyze up to three consecutive iterations of AFS-64. Table 2 compares the resulting bounds with those for the same number of Alzette iterations. Here, one iteration means one complete

application of the corresponding 64-bit ARX-box. For each iteration count shown, AFS-64 has both a lower MEDCP and a lower MELCC than Alzette, and therefore provides stronger resistance to single-trail differential and linear cryptanalysis.

Table 2 MEDCP and MELCC bounds for repeated applications of AFS-64 and Alzette [36].

ARX-box	Bound	1 iteration	2 iterations	3 iterations
AFS-64	MEDCP	2^{-10}	2^{-37}	$\leq 2^{-66}$
	MELCC	2^{-6}	2^{-20}	2^{-36}
Alzette	MEDCP	2^{-6}	2^{-34}	2^{-59}
	MELCC	2^{-2}	2^{-17}	2^{-30}

Diffusion. All output bits depend on all input bits after one iteration of AFS-64, although some dependencies remain weak. After two iterations, all output bits strongly depend on all input bits, and AFS-64 fulfills the strict avalanche criterion (SAC).

3.2.3 Choice of the Linear Layer

The two-branch case. For $\text{APEX}_{128}^{n_s}$, we initially considered the formal $n_b = 2$ boundary specialization of $\mathcal{L}_{n_b}$, in which the Feistel mixing function becomes

$$\mathcal{M}_1(x, y) = (x \oplus \ell(y), y \oplus \ell(x)).$$

We compared this construction with the simple XOR-based Feistel transformation

$$(x, y) \mapsto (y \oplus x, x).$$

The two variants were compared using differential and linear long-trail analysis, together with several other cryptanalytic techniques. Under the corresponding models, the additional middle map provided no material security improvement. In particular, the two variants yield identical differential and linear long-trail bounds. We therefore adopt the simple XOR-based Feistel layer, which avoids one additional 64-bit linear-map evaluation per step and gives a simpler specification and implementation.

The multi-branch case. The AFS family was designed to support the Long Trail Strategy. For $\text{APEX}_{64n_b}^{n_s}$ with even $n_b \geq 4$, we therefore use a classical SPN structure consisting of a parallel AFS-64 layer followed by a sparse linear layer. The SPN structure together with the sparse linear layer facilitates long-trail decomposition in both differential and linear analysis. Following the methodology used in the design of SPARX [22], we treated each AFS-64 instance as a black box characterized by its multi-iteration differential and linear bounds. For each candidate linear layer, we searched over the induced long-trail decompositions and derived the maximum attainable differential probability and absolute linear correlation. The resulting bounds were then used to compare the candidates. A Feistel structure provided the best balance between the two competing requirements. Copying one half of the state preserves uninterrupted local trails, while mixing the other half provides diffusion between branches. This is also the structure adopted by SPARKLE [31].

We instantiate the Feistel function with the linear mixing map $\mathcal{M}_{n_b/2}$ introduced for SPARKLE. It uses only word-wise XORs and applications of the inexpensive map ℓ . Moreover, $\mathcal{M}_w$ has differential branch number 3 for $w = 2$ and 4 for $w > 2$. A single active input branch activates every output branch, whereas a single active output branch requires every input branch to be active. Owing to the symmetry of its matrix, the corresponding branch-level diffusion property also holds for linear-mask propagation. These properties provide strong inter-branch diffusion without destroying the sparse Feistel structure required by the Long Trail Strategy.

Branch rotation. The final branch permutation cyclically rotates the mixed half $R \oplus \mathcal{M}_{n_b/2}(L)$ to the left by one branch and places it in the left half of the output, while the original left half L becomes the new right half. Let ρ_i denote a cyclic left rotation by i branches. Its order is

$$\text{ord}(\rho_i) = \frac{n_b/2}{\gcd(n_b/2, i)}.$$

Thus, when $\gcd(n_b/2, i) = 1$, the rotation has the maximum order $n_b/2$ and forms a single cycle over all branch positions in the half-state.

The truncated-differential model at the granularity of 64-bit branches and $\mathcal{M}_{n_b/2}$ are invariant under a relabeling of the branch positions. Consequently, two rotations ρ_i and ρ_j with $\gcd(n_b/2, i) = \gcd(n_b/2, j)$ have the same cycle decomposition and are equivalent in this model. For each supported $n_b \in \{4, 6, \dots, 24\}$, we therefore group the rotation amounts according to this greatest common divisor and test one rotation amount from each group using truncated-differential cryptanalysis. Rotations of order $n_b/2$ attain the shortest advantage-bearing truncated trails at every supported width. For $n_b \geq 8$, they strictly outperform every lower-order rotation, whereas all rotation classes tie for $n_b = 4$ and $n_b = 6$. We select $i = 1$ because $\gcd(n_b/2, 1) = 1$ for every supported n_b , making it the simplest uniform maximum-order choice. This choice also keeps software indexing simple: the one-branch rotation can be incorporated directly into the destination indices of the output assignments, avoiding a separate data-movement pass. Its single-cycle action continually changes the alignment between the copied and mixed branches and thereby suppresses short-period iterative truncated differentials. In contrast, the zero rotation admits a one-step iterative truncated differential. We therefore retain the one-branch rotation used by SPARKLE.

3.2.4 Selection of Round Constants

To make their generation transparent and reproducible, candidate constants are derived from the hexadecimal fractional expansion of π . After the integer part is removed, the hexadecimal digits are partitioned into consecutive blocks of eight digits, each interpreted as a 32-bit word. The resulting words are examined in their original order.

We first apply the quasidifferential framework of Beyne and Rijmen [39] in the fixed-key model. For an input-output differential Δ of $\mathcal{A}_c$, let $\mathcal{T}_\Delta$ denote its set of quasidifferential trails. For every $\tau \in \mathcal{T}_\Delta$, the constant-dependent sign is $\sigma_\tau(c) = \sigma_\tau(\mathbf{0})(-1)^{\langle M_\tau, c \rangle}$, where $\sigma_\tau(\mathbf{0}) \in \{+1, -1\}$ is the baseline sign of τ for the all-zero choice $c = \mathbf{0}$, and $M_\tau \in \mathbb{F}_2^{32}$ is the corresponding constant mask. Hence, $\langle M_\tau, c \rangle = 0$ preserves the baseline sign, whereas $\langle M_\tau, c \rangle = 1$ reverses it.

For the differential used to derive the hard constraints, the baseline differential weight (i.e., the negative base-2 logarithm of the probability) is $W_\Delta = 10$. We write the total weight of a quasidifferential contribution as $W_\tau = W_\Delta + \delta w_\tau$. The dominant contributions of total weights 12, 13, 15, 17, and 19, corresponding respectively to excess weights $\delta w_\tau = 2, 3, 5, 7$, and 9, yield a mutually consistent system. Requiring these contributions to have sign -1 gives

$$c[17] = c[18] = c[19] = c[20] = c[21] = 1, \quad c[22] = 0.$$

When the contributions of excess weight 10 (total weight 20) are also forced to be negative, the sign conditions become inconsistent. We therefore retain the compatible conditions above as hard constraints and evaluate the remaining conflicting contributions through a score.

More precisely, the quasidifferential estimate of the fixed-key differential probability is written as

$$P_\Delta(c) \approx 2^{-W_\Delta} F_\Delta(c), \quad F_\Delta(c) = 1 + \sum_{\tau \in \mathcal{T}_\Delta} \sigma_\tau(c) 2^{-\delta w_\tau}.$$

A smaller $F_\Delta(c)$ indicates stronger aggregate cancellation and a lower quasidifferential estimate of the fixed-key differential probability. For the set $\mathcal{D}$ of considered differentials, the primary criterion is to minimize $\max_{\Delta \in \mathcal{D}} F_\Delta(c)$, while $\sum_{\Delta \in \mathcal{D}} F_\Delta(c)$ is used as a secondary criterion.

We additionally require

$$12 \leq \text{wt}(c) \leq 20$$

where $\text{wt}(c)$ denotes the Hamming weight of c . This condition excludes unusually sparse or dense 32-bit constants.

Candidates satisfying the quasidifferential constraints and Hamming-weight condition above are further evaluated through empirical fixed-key differential and linear experiments, following the methodology used in the AFS and Alzette studies [30, 36]. The differential experiment uses three differential pairs associated with the MEDCP 2^{-10} and estimates their fixed-key probabilities. The linear experiment uses 960 linear approximations associated with the MELCC 2^{-6} and estimates their fixed-key linear-hull correlations. Each tested differential or linear approximation is evaluated using 2^{28} samples.

Across the 24 selected constants, the maximum tested differential probability remains within $2^{-10} \pm 10^{-5}$, indicating negligible differential clustering. The maximum absolute linear-hull correlation is approximately $2^{-5.25}$, indicating a slight clustering effect.

3.3 Security Analysis of the APEX Permutation Family

We evaluate the APEX permutation family under the full-state model, in which the adversary may control the complete b -bit state. Our analysis focuses primarily on differential and linear cryptanalysis, especially on bounds derived through the Long Trail Strategy. Following the LTS framework introduced for SPARX and subsequently developed in the analyses of Alzette and SPARKLE [22, 30, 31], together with the automated optimal long-trail search techniques introduced in the AFS study [36], we derive bounds on the maximum expected differential characteristic probability (MEDCP) and the maximum expected absolute linear characteristic correlation (MELCC) across the APEX permutation family.

3.3.1 Differential Long-Trail Cryptanalysis

Table 3 gives the stepwise MEDCP bounds for all supported APEX state sizes. These values are conservative upper bounds on the MEDCP and therefore generally underestimate the security strength of the permutations. Two sources contribute to this conservatism. First, even the optimal long-trail decomposition combines worst-case local bounds without requiring the corresponding locally optimal transitions to be simultaneously compatible at the bit level. This is analogous to a wide-trail bound derived from the number of active S-boxes. Second, direct MEDCP bounds are available only for up to three consecutive iterations of AFS-64; longer long trails are bounded by composing the available one-, two-, and three-iteration bounds.

Table 3 Differential long-trail weights $W_D(b, n_s)$ for all supported APEX state sizes. An entry w establishes $\text{MEDCP}(\text{APEX}_b^{n_s}) \leq 2^{-w}$. The bold green entry in each row is the first for which $w \geq b$. By monotonicity, each blank entry is no smaller than the last reported weight in the same row.

$b \backslash n_s$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
64	10	37	66																					
128	10	37	74	96	123	150	172	199	226	248	275	302	324											
256	10	37	86	116	168	205	236	266	314	344	386	416	464											
384	10	37	96	136	214	243	283	323	400	430	470	510	587											
512	10	37	106	156	251	280	330	380	475	504	554	604	699											
640	10	37	116	176	266	317	377	437	492	566	638	694												
768	10	37	126	192	276	354	424	494	563	603	680	720	804											
896	10	37	136	192	283	376	435	511	582	622	697	756	830	921										
1024	10	37	146	192	283	376	435	511	582	622	697	756	830	921	982	1052								
1152	10	37	152	192	283	376	435	511	582	622	697	756	830	921	982	1052	1127	1186						
1280	10	37	152	192	283	376	435	511	582	622	697	756	830	921	982	1052	1127	1186	1244	1301				
1408	10	37	152	192	283	376	435	511	582	622	697	756	830	921	982	1052	1127	1186	1244	1301	1394	1468		
1536	10	37	152	192	283	376	435	511	582	622	697	756	830	921	982	1052	1127	1186	1244	1301	1394	1468	1529	1616

3.3.2 Linear Long-Trail Cryptanalysis

The linear long-trail analysis proceeds analogously to the differential case. Table 4 gives the stepwise MELCC bounds and inherits the same conservative nature discussed above: the analysis combines worst-case local bounds, while direct MELCC bounds for AFS-64 are available only for up to three consecutive iterations.

Table 4 Linear long-trail weights $W_L(b, n_s)$ for all supported APEX state sizes. An entry w establishes $\text{MELCC}(\text{APEX}_b^{n_s}) \leq 2^{-w}$. The bold green entry in each row is the first for which $w \geq b/2$. By monotonicity, each blank entry is no smaller than the last reported weight in the same row.

$b \backslash n_s$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
64	6	20	36																					
128	6	20	40	54	68	82	96	110	124	138	152	166	180											
256	6	20	48	66	92	112	130	148	172	192	212	230	252											
384	6	20	54	78	116	132	156	180	218	234	258	282	320											
512	6	20	60	90	136	152	182	212	258	274	304	334	380											
640	6	20	66	102	148	172	208	244	272	312	350													
768	6	20	72	108	154	192	234	274	308	332	374	398												
896	6	20	78	108	156	206	240	282	318	342	382	416	456											
1024	6	20	84	108	156	206	240	282	318	342	382	416	456	504	540									
1152	6	20	84	108	156	206	240	282	318	342	382	416	456	504	540	576								
1280	6	20	84	108	156	206	240	282	318	342	382	416	456	504	540	576	616	650						
1408	6	20	84	108	156	206	240	282	318	342	382	416	456	504	540	576	616	650	682	714				
1536	6	20	84	108	156	206	240	282	318	342	382	416	456	504	540	576	616	650	682	714	764	804	840	884

3.3.3 Other Cryptanalytic Results

Beyond the long-trail bounds, we further evaluate the APEX permutation family using complementary cryptanalytic and diffusion analyses.

3.3.3.1 Integral Cryptanalysis

We perform bit-based division-property analysis [40] of $\text{APEX}_b^{n_s}$. The propagation constraints are encoded using monomial prediction [41], with modular addition modeled using perfect monomial prediction [42]. We consider inputs with $b - 1$ active bits and examine the resulting balanced output bits.

The longest integral distinguishers found are summarized in Table 5. The corresponding input activity patterns and balanced output bits are given in Appendix A.

Table 5 Longest integral distinguishers found for the $\text{APEX}_b^{n_s}$ permutation family. For $b = 64$, 1.25 steps denotes five internal AFS-64 rounds. For $b \geq 128$, a half-step boundary includes the parallel $\mathcal{A}$ -layer of the next step, but not its linear layer.

State size b	Forward	Inverse
64	1.25 steps	1.25 steps
128	3.5 steps	3 steps
$b \geq 256$	4.5 steps	4 steps

For $b = 64$, $\text{APEX}_{64}^{n_s}$ is precisely an iteration of AFS-64. One APEX_{64} step comprises the four internal rounds of AFS-64. The longest integral distinguishers found in both directions cover five such rounds, and therefore reach 1.25 APEX_{64} steps, consistently with the analysis in [36].

For $b = 128$, the longest forward integral distinguisher found reaches the 3.5-step boundary: three complete steps followed by the parallel $\mathcal{A}$ -layer of the fourth step, stopping before its linear layer. The longest inverse integral distinguisher found covers three complete inverse steps.

For $b \geq 256$, the structure admits integral distinguishers analogous to those established for SPARKLE [31]. In the forward direction, the distinguisher covers four complete steps followed by the parallel $\mathcal{A}$ -layer of the fifth step, stopping before its linear layer. We refer to this boundary as 4.5 steps. In the inverse direction, the distinguisher covers four complete inverse steps. Using the same structural argument as Propositions 4.2.1 and 4.2.2 of the SPARKLE specification [43], we obtain the following result for the APEX permutation family.

Proposition 1 (Structural integral distinguishers) *Let $b \geq 256$. In the forward direction, consider an input in which all $b/2$ bits of the right half are active, while the left half contains one fully active 64-bit branch and one additional active bit in another branch. After four complete APEX steps and the $\overline{\mathcal{A}}$ -layer of the fifth step, every bit in the right half is balanced. In the inverse direction, consider the mirrored input: the left half is fully active, while the right half contains one fully active 64-bit branch and one additional active bit in another branch. After four complete inverse steps, every bit in the left half is balanced.*

Proof We first explain the structural division-property notation used in the argument. Write

$$\mathbf{64} = (64, \dots, 64), \quad \mathbf{e} = (0, \dots, 0, 1, 64), \quad \mathbf{u} = (1, 1, 0, \dots, 0),$$

where each vector has $n_b/2$ entries, and let $\mathbf{0}$ be the all-zero vector. We use the standard structural division-property notation $\mathcal{D}_{\mathbf{k}_L, \mathbf{k}_R}^{\mathbf{64}, \mathbf{64}}$. The superscript specifies the dimensions of the $n_b/2$ component branches in each half-state, whereas the subscript gives their division-property exponents. A branch exponent $k_i \in \{0, \dots, 64\}$ records the monomial degree retained in the i th branch. At the input, it is also the number of active cube variables in that branch. Thus $\mathbf{64}$ represents a fully active half-state, $\mathbf{e}$ represents one fully active branch together with one active bit in another branch, and $\mathbf{u}$ retains one unit in each of two different branches. When the two half-states are treated as single $b/2$ -bit components, we use $\mathcal{D}_{(k_L), (k_R)}^{(b/2), (b/2)}$. In particular, $\mathcal{D}_{(0), (1)}^{(b/2), (b/2)}$ denotes a singleton exponent in the right half, while $\mathcal{D}_{(1), (0)}^{(b/2), (b/2)}$ denotes one in the left half.

An arrow between two division properties is a covering transition: the expression on the right contains every division trail admitted by the preceding layers, although it may also contain additional trails introduced by the abstraction. Therefore, if a singleton output exponent is absent even from this cover, the corresponding output bit is balanced. The fixed branch rotation in $\mathcal{L}_{n_b}$ merely permutes coordinates within a half-state, so it can be suppressed by relabeling the branches.

Let $\bar{\mathcal{A}}$ denote the parallel layer of n_b AFS-64 permutations and write $\mathcal{L} = \mathcal{L}_{n_b}$. The constant injections do not alter the division-property exponents. Each $\mathcal{A}_{c_i}$ is bijective and therefore preserves the structural properties with branch exponents 0, 1, and 64. Up to a permutation of the branch positions, every forward division trail from the stated input is covered by

$$\mathcal{D}_{\mathbf{e}, \mathbf{64}}^{\mathbf{64}, \mathbf{64}} \xrightarrow[\text{step 1}]{\mathcal{L} \circ \bar{\mathcal{A}}} \mathcal{D}_{\mathbf{64}, \mathbf{e}}^{\mathbf{64}, \mathbf{64}} \xrightarrow[\text{step 2}]{\mathcal{L} \circ \bar{\mathcal{A}}} \mathcal{D}_{\mathbf{0}, \mathbf{u}}^{\mathbf{64}, \mathbf{64}} \xrightarrow[\text{step 3}]{\mathcal{L} \circ \bar{\mathcal{A}}} \mathcal{D}_{\mathbf{u}, \mathbf{0}}^{\mathbf{64}, \mathbf{64}}. \quad (5)$$

The first transition exchanges the roles of the two halves: the Feistel copy places the sparse property $\mathbf{e}$ in one half, while the fully active property $\mathbf{64}$ covers the contribution entering the other half. The second transition is the only counting step in (5). The half initially carrying $\mathbf{64}$ has $b/2$ active units. Saturating the other half, which already carries 65 units, can transfer at most $b/2 - 65$ of them through the Feistel mixing function. At least 65 units consequently remain, and hence at least two branches remain active because one branch can contain at most 64 units. Replacing these residual units by the two unit exponents in $\mathbf{u}$ weakens the property and gives a cover of all admissible trails. The third transition then copies these two surviving units to the opposite half, while the zero half contributes nothing.

The $\bar{\mathcal{A}}$ -layer of the fourth step leaves these two unit exponents active. The fourth linear layer can transfer neither, one, or both of them through its Feistel function. After the final parallel $\bar{\mathcal{A}}$ -layer, all resulting properties are covered by

$$\mathcal{D}_{\mathbf{u}, \mathbf{0}}^{\mathbf{64}, \mathbf{64}} \xrightarrow[\text{to 4.5 steps}]{\bar{\mathcal{A}} \circ \mathcal{L} \circ \bar{\mathcal{A}}} \mathcal{D}_{(1), (0)}^{(b/2), (b/2)} \cup \mathcal{D}_{(0), (2)}^{(b/2), (b/2)}. \quad (6)$$

The three possibilities—transferring neither, one, or both units—are all covered by the two properties on the right of (6); the intermediate one-one case is already covered by the first property. Crucially, $\mathcal{D}_{(0), (1)}^{(b/2), (b/2)}$, which would represent a singleton output exponent in the right half, is absent. Every bit in the right output half is therefore balanced. Equation (5) accounts for the first three steps, while (6) contains the $\bar{\mathcal{A}}$ - and $\mathcal{L}$ -layers of step 4 followed by the $\bar{\mathcal{A}}$ -layer of step 5. This is precisely the claimed 4.5-step boundary.

For the inverse direction, start from $\mathcal{D}_{\mathbf{64}, \mathbf{e}}^{\mathbf{64}, \mathbf{64}}$. The same covering argument applied to $\mathcal{L}_{n_b}^{-1}$ and the bijections $\mathcal{A}_{c_i}^{-1}$ gives

$$\mathcal{D}_{\mathbf{64}, \mathbf{e}}^{\mathbf{64}, \mathbf{64}} \xrightarrow[\text{inverse step 1}]{\bar{\mathcal{A}}^{-1} \circ \mathcal{L}^{-1}} \mathcal{D}_{\mathbf{e}, \mathbf{64}}^{\mathbf{64}, \mathbf{64}} \xrightarrow[\text{inverse step 2}]{\bar{\mathcal{A}}^{-1} \circ \mathcal{L}^{-1}} \mathcal{D}_{\mathbf{u}, \mathbf{e}}^{\mathbf{64}, \mathbf{64}} \xrightarrow[\text{inverse step 3}]{\bar{\mathcal{A}}^{-1} \circ \mathcal{L}^{-1}} \mathcal{D}_{\mathbf{0}, \mathbf{u}}^{\mathbf{64}, \mathbf{64}} \xrightarrow[\text{inverse step 4}]{\bar{\mathcal{A}}^{-1} \circ \mathcal{L}^{-1}} \mathcal{D}_{(2), (0)}^{(b/2), (b/2)} \cup \mathcal{D}_{(0), (1)}^{(b/2), (b/2)}. \quad (7)$$

Thus, $\mathcal{L}^{-1}$ is applied first in each arrow and $\bar{\mathcal{A}}^{-1}$ second. The target singleton property $\mathcal{D}_{(1), (0)}^{(b/2), (b/2)}$ is absent from the final cover, so every bit in the left output half is balanced. No additional partial layer is used; the inverse distinguisher therefore covers four complete steps. $\square$

We independently verified the structural distinguishers in Proposition 1 using an automated MILP model. We further searched for longer distinguishers under the setting with $b - 1$ active input bits. In the forward searches, the inactive input bit was placed at global bit positions 63 and $63 + b/2$, inherited from the strongest forward integral distinguisher of AFS-64. In the inverse searches, bit positions 30 and $30 + b/2$ were tested, following one of the strongest inverse integral distinguishers of AFS-64. No longer distinguisher was found in any of these searches.

3.3.3.2 Boomerang Cryptanalysis

In a classical boomerang distinguisher [44], the permutation P is split into two parts, $P = P_1 \circ P_0$. If differentials through P_0 and P_1 hold with probabilities p and q , respectively, the classical two-differential estimate gives boomerang probability $p^2 q^2$.

Table 6 Classical boomerang weights $W_{\text{boom}} = -\log_2 P_{\text{boom}}$ inferred from the differential long-trail bounds in Table 3. The bold green entry in each row is the first weight satisfying $W_{\text{boom}}(b, n_s) \geq b$.

$b \backslash n_s$	3	4	5	6	7	8	9	10	11	12	13	14
128	94	148										
256	94	148	246	306								
384	94	148	266	346	464							
512	94	148	286	386	524							
640	94	148	306	426	584	704						
768	94	148	326	458	626	768						
896	94	148	346	458	640	768	944					
1024	94	148	366	458	640	768	944	1096				
1152	94	148	378	458	640	768	944	1096	1238			
1280	94	148	378	458	640	768	944	1096	1238	1318		
1408	94	148	378	458	640	768	944	1096	1238	1318	1468	
1536	94	148	378	458	640	768	944	1096	1238	1318	1468	1586

3.3.3.3 Truncated-Differential Cryptanalysis

We use the branch-level truncated-differential method proposed in SPARKLE [43] to analyze the $\text{APEX}_b^{n_s}$ permutations. Each 64-bit branch is represented as either active or inactive, and all admissible truncated trails in this representation are searched exhaustively. If p denotes the probability assigned to a trail by this branch-level transition model, its normalized weight is defined as $W_{\text{TD}} = -\log_2(p)/64$. A truncated trail is called effective if W_{TD} is strictly smaller than the number of inactive branches in its output pattern. Equivalently, its model probability exceeds the approximate random-permutation probability of obtaining those inactive output branches. The longest effective trails found are summarized in Table 7.

Table 7 Longest effective branch-level truncated differentials found for the $\text{APEX}_b^{n_s}$ permutation family. All entries count complete APEX steps.

State size b	128	256	384	512	640	768	896	1024	1152	1280	1408	1536
Longest n_s	1	2	2	3	4	5	6	7	8	9	10	11

3.3.3.4 Impossible-Differential and Zero-Correlation Linear Cryptanalysis

For $b \geq 256$, we use the branch-truncated approach described in [45, 46], treating each AFS-64 instance as a black box. For zero-correlation linear cryptanalysis, the same approach is applied to the inverse transpose of the linear layer. We additionally analyze the case $b = 128$ directly.

For $b = 128$, the longest impossible-differential and zero-correlation linear distinguishers found cover four steps. For every supported state size $b \geq 256$, both types of distinguishers also cover four steps. These results are summarized in Table 8. The four-step relations for $b = 128$ are proved below. Details of the impossible-differential and zero-correlation linear distinguishers are given in Appendix B.

Table 8 Longest impossible-differential and zero-correlation linear distinguishers found for the $\text{APEX}_b^{n_s}$ permutation family. All entries count complete APEX steps.

State size b	Impossible differential	Zero-correlation linear
128	4 steps	4 steps
$b \in \{256, 384, \dots, 1536\}$	4 steps	4 steps

Proposition 2 (Four-step distinguishers for $\text{APEX}_{128}^{n_s}$) *For arbitrary nonzero differences $\delta, \Delta \in \mathbb{F}_2^{64}$, APEX_{128}^4 admits the impossible differential*

$$(0, \delta) \xrightarrow[\text{DP}=0]{\text{APEX}_{128}^4} (\Delta, \Delta)$$

over four steps. Moreover, for arbitrary nonzero masks $\mu, \nu \in \mathbb{F}_2^{64}$, we have the zero-correlation relation

$$(\mu, 0) \xrightarrow[\text{Cor}=0]{\text{APEX}_{128}^4} (\nu, \nu).$$

Proof Write the two output differences of the parallel $\mathcal{A}$ -layer in step i as (α_i, β_i) . The following Feistel layer maps them to $(\beta_i \oplus \alpha_i, \alpha_i)$. Since every $\mathcal{A}_{c_j}$ is a permutation, a zero input difference produces a zero output difference, whereas a nonzero input difference produces a nonzero output difference.

Starting from $(0, \delta)$, the state differences after the first two steps have the forced forms

$$(0, \delta) \longrightarrow (\beta_1, 0) \longrightarrow (\alpha_2, \alpha_2), \quad \beta_1 \neq 0, \quad \alpha_2 \neq 0.$$

The right output difference of the third step, and hence the right input difference of the fourth step, is nonzero. Consequently, $\beta_4 \neq 0$. If the fourth-step output difference were (Δ, Δ) , then

$$\alpha_4 = \Delta, \quad \beta_4 \oplus \alpha_4 = \Delta$$

would imply $\beta_4 = 0$, a contradiction. This proves the impossible differential.

For the linear case, the inverse transpose of the Feistel layer propagates a mask pair as

$$(u, v) \mapsto (v, u \oplus v).$$

A nonzero-correlation approximation through a bijection cannot pair a zero mask with a nonzero mask. Starting from $(\mu, 0)$, the first two steps therefore force mask pairs of the forms

$$(\mu, 0) \longrightarrow (0, u_1) \longrightarrow (u_2, u_2), \quad u_1, u_2 \neq 0.$$

Thus, the left input mask of the fourth $\mathcal{A}$ -layer is nonzero. If the final mask were (ν, ν) , the inverse-transpose Feistel equation would force the corresponding left output mask of that $\mathcal{A}$ -layer to be zero. Hence, no nonzero-correlation linear trail is compatible with the prescribed endpoint masks. Therefore, the complete four-step linear hull has correlation zero. $\square$

3.3.3.5 Yoyo Games

Yoyo distinguishers were introduced in [47] to expose insufficient diffusion in the block cipher Skipjack. We also investigated Yoyo distinguishers experimentally for reduced-step $\text{APEX}_b^{n_s}$ permutations. For $b = 128$, we found probability-one truncated Yoyo distinguishers covering 3 steps, but none covering more than 3 steps. For every supported state size $b \geq 256$, we found probability-one truncated Yoyo distinguishers covering 4 steps, but none covering more than 4 steps. All longest distinguishers found have the same structure. As customary in the description of a Yoyo game, we use “plaintext” and “ciphertext” for the input and output of the permutation, although APEX itself is unkeyed:

1. Choose two plaintexts whose difference is confined to one 64-bit branch in the right half.
2. Apply the permutation to both plaintexts to obtain two ciphertexts.
3. Create two new ciphertexts by swapping one 64-bit branch in the left half between the two ciphertexts.
4. Apply the inverse permutation to the two modified ciphertexts.

The experiment yields a truncated Yoyo distinguisher when the two resulting plaintexts again differ in only one 64-bit branch, a property that is not expected to hold with probability one for a random permutation.

3.3.3.6 Diffusion Test (SAC)

The strict avalanche criterion (SAC), introduced in [48], measures diffusion in cryptographic functions. It requires that, for every input bit i and output bit j , complementing input bit i changes output bit j with probability $1/2$. We take a set $\mathcal{S}$ of randomly sampled b -bit states and let P denote the tested forward or inverse permutation. The entries of the $b \times b$ dependency matrix are $\mathbb{D}_{i,j} = |\{x \in \mathcal{S} : P(x)_j \neq P(x \oplus 2^i)_j\}|$. For a secure cryptographic permutation, every $\mathbb{D}_{i,j}$ is expected to follow the binomial distribution $B(|\mathcal{S}|, 1/2)$. Under SAC, the mean $\mu_{\mathbb{D}}$ and empirical variance $\sigma_{\mathbb{D}}^2$ of the matrix entries are expected to be close to $|\mathcal{S}|/2$ and $|\mathcal{S}|/4$, respectively. We say that sampled full diffusion is reached when $\mathbb{D}$ has no zero entry.

We evaluated $\text{APEX}_b^{n_s}$ in both the forward and inverse directions using 10^7 random samples. Figure 6 depicts the distribution of the occurrences of specific values in $\mathbb{D}_{i,j}$ for APEX_{512}^3 and 10^7 random samples, compared with what is expected from the binomial distribution $B(10^7, 1/2)$. Full diffusion is reached after one step for APEX_{64} , two steps for APEX_{128} , and three steps for all APEX_b with $b \geq 256$. Our results indicate that two steps are sufficient to fulfill the SAC for APEX_{64} and APEX_{128} , while three steps are sufficient for all APEX_b with $b \geq 256$. The complete results are reported in Table 9.

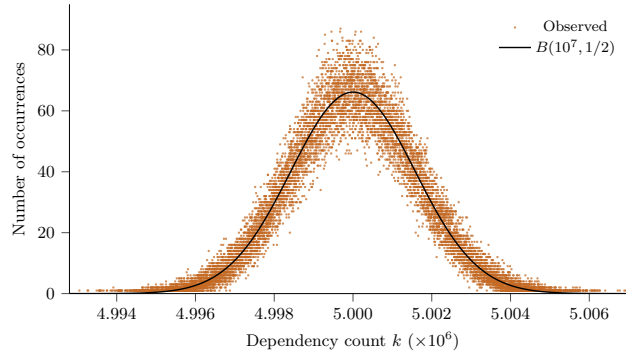


Fig. 6 Number of times (vertical axis) for which a specific entry (horizontal axis) occurs in the forward dependency matrix after three steps of APEX_{512}^3 for 10^7 samples (orange), and the theoretical number of occurrences corresponding to a $B(10^7, 1/2)$ distribution (black).

Table 9 Properties of the dependency matrix $\mathbb{D}$ for $\text{APEX}_b^{n_s}$.*Forward direction*

(b, n_s)	$\min_{i,j} \mathbb{D}_{i,j}$	$\max_{i,j} \mathbb{D}_{i,j}$	$\mu_{\mathbb{D}} = \frac{1}{b^2} \sum_{i,j=0}^{b-1} \mathbb{D}_{i,j}$	$\sigma_{\mathbb{D}}^2 = \frac{1}{b^2} \sum_{i,j=0}^{b-1} (\mathbb{D}_{i,j} - \mu_{\mathbb{D}})^2$
(64, 1)	283,151	9,794,288	4,311,521	3,798,491,582,333
(64, 2)	4,993,828	5,005,997	5,000,017	2,652,420
(64, 3)	4,994,273	5,004,894	4,999,982	2,485,818
(128, 2)	4,993,615	5,007,055	4,999,995	2,551,046
(128, 3)	4,994,031	5,006,006	4,999,966	2,499,771
(256, 3)	4,993,186	5,006,240	4,999,992	2,470,382
(384, 3)	4,993,032	5,006,981	4,999,992	2,495,901
(512, 3)	4,993,052	5,006,930	5,000,000	2,505,647
(640, 3)	4,992,584	5,007,164	5,000,006	2,498,481
(768, 3)	4,991,774	5,007,246	5,000,000	2,498,286
(896, 3)	4,992,297	5,007,639	5,000,001	2,499,503
(1024, 3)	4,992,380	5,007,428	4,999,998	2,499,913
(1152, 3)	4,991,941	5,007,349	5,000,000	2,498,920
(1280, 3)	4,991,301	5,008,874	5,000,001	2,502,904
(1408, 3)	4,991,740	5,007,836	4,999,999	2,500,542
(1536, 3)	4,992,106	5,008,007	4,999,999	2,501,095

Inverse direction

(b, n_s)	$\min_{i,j} \mathbb{D}_{i,j}$	$\max_{i,j} \mathbb{D}_{i,j}$	$\mu_{\mathbb{D}} = \frac{1}{b^2} \sum_{i,j=0}^{b-1} \mathbb{D}_{i,j}$	$\sigma_{\mathbb{D}}^2 = \frac{1}{b^2} \sum_{i,j=0}^{b-1} (\mathbb{D}_{i,j} - \mu_{\mathbb{D}})^2$
(64, 1)	156,576	9,885,193	4,200,834	4,422,990,579,299
(64, 2)	4,994,063	5,006,084	4,999,992	2,481,634
(64, 3)	4,994,053	5,005,387	5,000,042	2,567,491
(128, 2)	4,993,592	5,008,457	5,000,030	2,568,108
(128, 3)	4,994,379	5,005,802	5,000,001	2,483,591
(256, 3)	4,992,398	5,006,618	4,999,996	2,526,636
(384, 3)	4,992,570	5,006,991	4,999,999	2,490,707
(512, 3)	4,992,773	5,007,201	4,999,995	2,509,359
(640, 3)	4,992,808	5,007,421	5,000,000	2,498,409
(768, 3)	4,992,607	5,007,306	5,000,003	2,505,390
(896, 3)	4,992,601	5,007,359	5,000,003	2,504,110
(1024, 3)	4,991,213	5,008,031	4,999,998	2,496,817
(1152, 3)	4,992,191	5,008,142	5,000,000	2,500,700
(1280, 3)	4,992,399	5,007,883	4,999,999	2,498,949
(1408, 3)	4,991,889	5,007,878	5,000,000	2,502,514
(1536, 3)	4,991,867	5,007,609	4,999,999	2,499,040

3.3.3.7 Symmetry-Based Cryptanalysis

The use of distinct and dense branch constants, together with the addition of different constants before each step, breaks symmetries within the APEX permutations. For APEX_{64} , step-dependent round constants play the same role by breaking symmetry between successive iterations.

Slide attacks [49]. For $\text{APEX}_b^{n_s}$ with $b = 64$, the use of a different round constant in each step assures that each step is different. For $b \geq 128$, the XOR of the step counter into the second branch before each step also assures that each step is different. Therefore, classical slide attacks cannot be applied to the APEX permutations.

Rotational cryptanalysis. Due to the extensive use of dense round constants, we expect APEX to be resistant to rotational cryptanalysis. Moreover, for $b \geq 256$, the function ℓ used in the linear Feistel mix $\mathcal{M}_{n_b/2}$ also breaks the rotational symmetry of 32-bit words.

3.4 Implementation Considerations

The APEX permutation family is primarily designed for efficient software implementation. Its AFS-64 branches consist entirely of 32-bit modular additions, XORs, and fixed rotations, which map naturally to general-purpose processors, particularly 32-bit architectures. The regular branch structure also permits scalable hardware implementations, although a detailed hardware evaluation is outside the scope of this work. Since the concrete implementation strategy depends on the state width, step schedule, and surrounding mode, implementation results are presented together with the corresponding applications in the following sections. We focus primarily on software implementations throughout

this paper. This software-oriented design choice follows the established distinction in lightweight cryptography between software-efficient and hardware-efficient primitives, as exemplified by SPECK and SIMON, respectively [11].

4 Naming Convention for APEX-Based Constructions

We adopt a uniform naming convention for cryptographic primitives built from the APEX permutation family. The following sections present several concrete applications of APEX. These applications are representative rather than exhaustive: the flexibility and extensibility of the permutation family allow further modes and parameter sets to be defined for other requirements.

We use the following generic form for the names of APEX-based constructions:

$$\text{APEX}_{width}^{steps}\text{-mode}[-\text{variant}]\text{-parameters}.$$

Here, *width* denotes the size in bits of the underlying APEX permutation, and *steps* specifies the number of APEX steps used by the construction. A construction need not use the same step count for every permutation call. If a construction employs multiple step counts, the *steps* field is written as the corresponding ordered tuple. *mode* identifies the construction, while the optional *variant* field distinguishes a named variant of that mode. The final *parameters* field lists, in a mode-specific fixed order, the relevant mode parameters, such as output, nonce, key, and tweak lengths.

Table 10 Representative applications of the APEX permutation family considered in this paper. All lengths are in bits.

Primitive	Representative instances	<i>parameters</i> field
HASH function	$\text{APEX}_{384}^{(6,9)\text{-HASH-256}}$	Output digest length
	$\text{APEX}_{512}^{(7,10)\text{-HASH-384}}$	
	$\text{APEX}_{1536}^{12\text{-HASH-F-512}}$	
	$\text{APEX}_{1536}^{16\text{-HASH-F-768}}$	
	$\text{APEX}_{1536}^{20\text{-HASH-F-1024}}$	
Authenticated Encryption with Associated Data	$\text{APEX}_{256}^{(6,8)\text{-AEAD-128/128}}$	Nonce length / Key length (the tag length equals the key length)
	$\text{APEX}_{384}^{(6,9)\text{-AEAD-256/128}}$	
	$\text{APEX}_{384}^{(6,9)\text{-AEAD-192/192}}$	
	$\text{APEX}_{512}^{(7,10)\text{-AEAD-256/256}}$	
eXtendable-Output Function	$\text{APEX}_{384}^{(6,9)\text{-HASH-XO-256}}$	Underlying HASH digest length (the XOF output length is variable)
	$\text{APEX}_{512}^{(7,10)\text{-HASH-XO-384}}$	
Block Cipher	$\text{APEX}_{64}^8\text{-BC-128}$	Key length
	$\text{APEX}_{256}^{15\text{-BC-256}}$	
Tweakable Block Cipher	$\text{APEX}_{256}^{15\text{-TBC-256/128}}$	Key length / Tweak length

5 Small-State Hashing and AEAD

In this section, we present several lightweight cryptographic primitives with small internal states built from the APEX permutation family, including hash functions, eXtendable-output functions (XOFs), and authenticated-encryption-with-associated-data (AEAD) schemes.

5.1 Specification

5.1.1 The Hash Functions $\text{APEX}_{384}^{(6,9)\text{-HASH-256}}$ and $\text{APEX}_{512}^{(7,10)\text{-HASH-384}}$

We propose two fixed-output hash functions, $\text{APEX}_{384}^{(6,9)\text{-HASH-256}}$ and $\text{APEX}_{512}^{(7,10)\text{-HASH-384}}$. They process messages $M \in \mathbb{F}_2^*$ and output digests D of 256 and 384 bits, respectively. Both functions use a sponge-based hashing mode

instantiated with APEX permutations. The rate is $r = 128$ bits in both instances, while the capacity c equals the digest length. The slim permutation is used after each nonfinal absorption block and between consecutive output blocks. The big permutation is used once after the final absorption block and before the squeezing phase. The instance parameters are summarized in Table 11. The two hash functions are formally specified in Algorithms 5 and 6 and illustrated in Figure 7 and Figure 8, respectively.

The target security level for collision, second-preimage, and preimage resistance is $c/2$ bits. The message length is limited to $r \cdot 2^{c/2}$ bits. The resulting security levels and data limits are also included in Table 11.

Table 11 Parameters and target security levels of the two hash instances. Sizes and security levels are in bits unless stated otherwise.

Instance	Parameters						Collision	Security level		Data limit (bytes)
	b	r	c	d	n_{slim}	n_{big}		2nd preimage	Preimage	
APEX ₃₈₄ ^(6,9) -HASH-256	384	128	256	256	6	9	128	128	128	2^{132}
APEX ₅₁₂ ^(7,10) -HASH-384	512	128	384	384	7	10	192	192	192	2^{196}

The state is initialized to zero. The message is divided into 128-bit blocks. A final block shorter than 128 bits is padded using the simple 10* rule, denoted by pad_r , and specified in Algorithm 4. In particular, the empty message produces one padded block. If the message length is a positive multiple of 128, its final complete block is not padded. Following the minimum-padding technique of Hirose [50], a message-domain constant Const_M is XORed into the inner part to prevent trivial collisions. Its value depends on whether the final block is padded.

Message blocks are injected indirectly. In the 256-bit instance, each 128-bit message block is extended by 64 zero bits and transformed by $\mathcal{M}_3$. In the 384-bit instance, each block is extended by 128 zero bits and transformed by $\mathcal{M}_4$. The resulting 192-bit and 256-bit values, respectively, are XORed into the left half of the state. This is equivalent to a regular sponge description in which the message is injected directly and the permutation is conjugated by the corresponding linear map. After the final block is processed with the big permutation, the leftmost 128 state bits form the first output block. For each additional output block, the slim permutation is applied once and the new leftmost 128 state bits are appended. Their concatenation is finally reduced to its leftmost t bits by trunc_t^L to produce the digest.

Algorithm 4 The padding function pad_r .

Input/Output: $M \in \mathbb{F}_2^*$ with $|M| < r$

$i \leftarrow (-1 - |M|) \bmod r$

$M \leftarrow M \parallel 1 \parallel 0^i$

return M

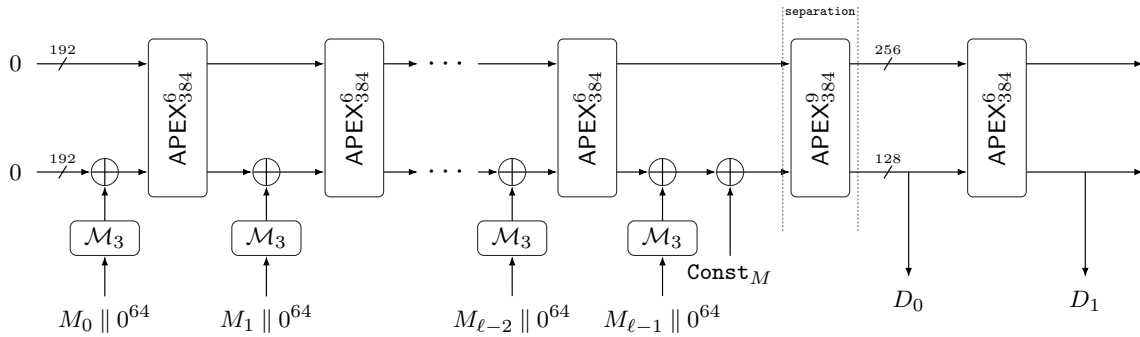


Fig. 7 The hash function APEX₃₈₄^(6,9)-HASH-256 with rate $r = 128$ and capacity $c = 256$. Each message block is extended by 64 zero bits and mapped by $\mathcal{M}_3$ before injection. For a padded final block, $\text{Const}_M = 0^{191} \parallel 1 \in \mathbb{F}_2^{192}$. Otherwise, $\text{Const}_M = 0^{190} \parallel 10 \in \mathbb{F}_2^{192}$. The digest is $D = D_0 \parallel D_1 \in \mathbb{F}_2^{256}$.

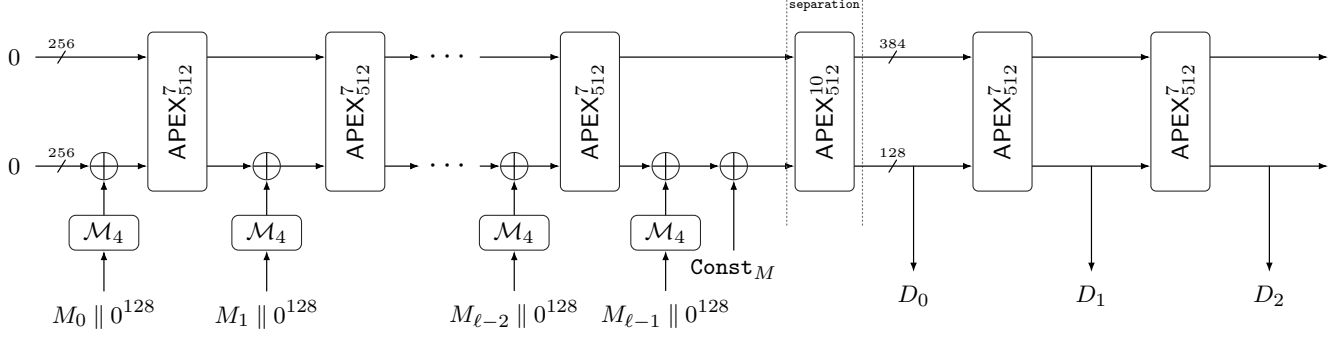


Fig. 8 The hash function $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$ with rate $r = 128$ and capacity $c = 384$. Each message block is extended by 128 zero bits and mapped by $\mathcal{M}_4$ before injection. For a padded final block, $\text{Const}_M = 0^{255} \parallel 1 \in \mathbb{F}_2^{256}$. Otherwise, $\text{Const}_M = 0^{254} \parallel 10 \in \mathbb{F}_2^{256}$. The digest is $D = D_0 \parallel D_1 \parallel D_2 \in \mathbb{F}_2^{384}$.

Algorithm 5 $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$.

Input/Output: $M \in \mathbb{F}_2^* / D \in \mathbb{F}_2^{256}$

▷ Message parsing and padding

if $M \neq \epsilon$ **then**
 $M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M$
 where $|M_i| = 128$ ($0 \leq i < \ell - 1$) and $1 \leq |M_{\ell-1}| \leq 128$

else
 $\ell \leftarrow 1$
 $M_0 \leftarrow \epsilon$

end if
if $|M_{\ell-1}| < 128$ **then**
 $M_{\ell-1} \leftarrow \text{pad}_{128}(M_{\ell-1})$
 $\text{Const}_M \leftarrow 0^{191} \parallel 1 \in \mathbb{F}_2^{192}$

else
 $\text{Const}_M \leftarrow 0^{190} \parallel 10 \in \mathbb{F}_2^{192}$

end if
 $S \leftarrow 0 \in \mathbb{F}_2^{384}$
for all $j \in [0, \ell - 2]$ **do**
 $M'_j \leftarrow \mathcal{M}_3(M_j \parallel 0^{64})$
 $S \leftarrow \text{APEX}_{384}^6(S \oplus (M'_j \parallel 0^{192}))$
end for
 $M'_{\ell-1} \leftarrow \mathcal{M}_3(M_{\ell-1} \parallel 0^{64})$
 $S \leftarrow \text{APEX}_{384}^9(S \oplus ((M'_{\ell-1} \oplus \text{Const}_M) \parallel 0^{192}))$

▷ Absorption

$D_0 \leftarrow \text{trunc}_{128}^L(S)$
 $S \leftarrow \text{APEX}_{384}^6(S)$
 $D_1 \leftarrow \text{trunc}_{128}^L(S)$

return $D_0 \parallel D_1$

Algorithm 6 $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$.

Input/Output: $M \in \mathbb{F}_2^* / D \in \mathbb{F}_2^{384}$

▷ Message parsing and padding

if $M \neq \epsilon$ **then**
 $M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M$
 where $|M_i| = 128$ ($0 \leq i < \ell - 1$) and $1 \leq |M_{\ell-1}| \leq 128$

else
 $\ell \leftarrow 1$
 $M_0 \leftarrow \epsilon$

end if
if $|M_{\ell-1}| < 128$ **then**
 $M_{\ell-1} \leftarrow \text{pad}_{128}(M_{\ell-1})$
 $\text{Const}_M \leftarrow 0^{255} \parallel 1 \in \mathbb{F}_2^{256}$

else
 $\text{Const}_M \leftarrow 0^{254} \parallel 10 \in \mathbb{F}_2^{256}$

end if
 $S \leftarrow 0 \in \mathbb{F}_2^{512}$
for all $j \in [0, \ell - 2]$ **do**
 $M'_j \leftarrow \mathcal{M}_4(M_j \parallel 0^{128})$
 $S \leftarrow \text{APEX}_{512}^7(S \oplus (M'_j \parallel 0^{256}))$
end for
 $M'_{\ell-1} \leftarrow \mathcal{M}_4(M_{\ell-1} \parallel 0^{128})$
 $S \leftarrow \text{APEX}_{512}^{10}(S \oplus ((M'_{\ell-1} \oplus \text{Const}_M) \parallel 0^{256}))$

▷ Absorption

$D_0 \leftarrow \text{trunc}_{128}^L(S)$
 $S \leftarrow \text{APEX}_{512}^7(S)$
 $D_1 \leftarrow \text{trunc}_{128}^L(S)$
 $S \leftarrow \text{APEX}_{512}^7(S)$
 $D_2 \leftarrow \text{trunc}_{128}^L(S)$
return $D_0 \parallel D_1 \parallel D_2$

▷ Squeezing

5.1.2 The Extendable-Output Functions $\text{APEX}_{384}^{(6,9)}\text{-HASH-XO-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-XO-384}$

The hash functions $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$ can be readily adapted to produce outputs of arbitrary length. We define the extendable-output functions (XOFs) $\text{APEX}_{384}^{(6,9)}\text{-HASH-XO-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-XO-384}$, which closely follow their fixed-output counterparts. They use distinct values of Const_M for domain separation and take an additional parameter $t > 0$ specifying the output length. The squeezing phase is extended until t output bits have been generated. The two XOFs are formally specified in Algorithms 7 and 8, respectively. The suffixes 256 and 384 identify the underlying HASH profiles. The parameters and target security levels of the two XOFs are summarized in Table 12.

Table 12 Parameters and target security levels of the two XOF instances. The requested output length is t bits. Other sizes and security levels are in bits unless stated otherwise.

Instance	b	r	Parameters				Collision	Security level 2nd preimage	Preimage	Data limit (bytes)
$\text{APEX}_{384}^{(6,9)}\text{-HASH-XO-256}$	384	128	256	t	6	9	$\min\{128, t/2\}$	$\min\{128, t\}$	$\min\{128, t\}$	2^{132}
$\text{APEX}_{512}^{(7,10)}\text{-HASH-XO-384}$	512	128	384	t	7	10	$\min\{192, t/2\}$	$\min\{192, t\}$	$\min\{192, t\}$	2^{196}

Algorithm 7 $\text{APEX}_{384}^{(6,9)}\text{-HASH-XO-256}$.

Input/Output: $M \in \mathbb{F}_2^*$, $t \in \mathbb{N}_{>0}$ / $D \in \mathbb{F}_2^t$

```

    ▷ Message parsing and padding
    if  $M \neq \epsilon$  then
         $M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M$ 
        where  $|M_i| = 128$  ( $0 \leq i < \ell - 1$ ) and  $1 \leq |M_{\ell-1}| \leq 128$ 
    else
         $\ell \leftarrow 1$ 
         $M_0 \leftarrow \epsilon$ 
    end if
    if  $|M_{\ell-1}| < 128$  then
         $M_{\ell-1} \leftarrow \text{pad}_{128}(M_{\ell-1})$ 
         $\text{Const}_M \leftarrow (0, \dots, 0, 1, 0, 1) \in \mathbb{F}_2^{192}$ 
    else
         $\text{Const}_M \leftarrow (0, \dots, 0, 1, 1, 0) \in \mathbb{F}_2^{192}$ 
    end if

    ▷ Absorption
     $S \leftarrow 0 \in \mathbb{F}_2^{384}$ 
    for all  $j \in [0, \ell - 2]$  do
         $M'_j \leftarrow \mathcal{M}_3(M_j \parallel 0^{64})$ 
         $S \leftarrow \text{APEX}_{384}^6(S \oplus (M'_j \parallel 0^{192}))$ 
    end for
     $M'_{\ell-1} \leftarrow \mathcal{M}_3(M_{\ell-1} \parallel 0^{64})$ 
     $S \leftarrow \text{APEX}_{384}^9(S \oplus ((M'_{\ell-1} \oplus \text{Const}_M) \parallel 0^{192}))$ 

    ▷ Squeezing
     $D_0 \leftarrow \text{trunc}_{128}^L(S)$ 
    for all  $j \in [1, \lceil t/128 \rceil - 1]$  do
         $S \leftarrow \text{APEX}_{384}^6(S)$ 
         $D_j \leftarrow \text{trunc}_{128}^L(S)$ 
    end for
    return  $\text{trunc}_t^L(D_0 \parallel D_1 \parallel \dots \parallel D_{\lceil t/128 \rceil - 1})$ 

```

Algorithm 8 $\text{APEX}_{512}^{(7,10)}\text{-HASH-XO-384}$.

Input/Output: $M \in \mathbb{F}_2^*$, $t \in \mathbb{N}_{>0}$ / $D \in \mathbb{F}_2^t$

```

    ▷ Message parsing and padding
    if  $M \neq \epsilon$  then
         $M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M$ 
        where  $|M_i| = 128$  ( $0 \leq i < \ell - 1$ ) and  $1 \leq |M_{\ell-1}| \leq 128$ 
    else
         $\ell \leftarrow 1$ 
         $M_0 \leftarrow \epsilon$ 
    end if
    if  $|M_{\ell-1}| < 128$  then
         $M_{\ell-1} \leftarrow \text{pad}_{128}(M_{\ell-1})$ 
         $\text{Const}_M \leftarrow (0, \dots, 0, 1, 0, 1) \in \mathbb{F}_2^{256}$ 
    else
         $\text{Const}_M \leftarrow (0, \dots, 0, 1, 1, 0) \in \mathbb{F}_2^{256}$ 
    end if

    ▷ Absorption
     $S \leftarrow 0 \in \mathbb{F}_2^{512}$ 
    for all  $j \in [0, \ell - 2]$  do
         $M'_j \leftarrow \mathcal{M}_4(M_j \parallel 0^{128})$ 
         $S \leftarrow \text{APEX}_{512}^7(S \oplus (M'_j \parallel 0^{256}))$ 
    end for
     $M'_{\ell-1} \leftarrow \mathcal{M}_4(M_{\ell-1} \parallel 0^{128})$ 
     $S \leftarrow \text{APEX}_{512}^{10}(S \oplus ((M'_{\ell-1} \oplus \text{Const}_M) \parallel 0^{256}))$ 

    ▷ Squeezing
     $D_0 \leftarrow \text{trunc}_{128}^L(S)$ 
    for all  $j \in [1, \lceil t/128 \rceil - 1]$  do
         $S \leftarrow \text{APEX}_{512}^7(S)$ 
         $D_j \leftarrow \text{trunc}_{128}^L(S)$ 
    end for
    return  $\text{trunc}_t^L(D_0 \parallel D_1 \parallel \dots \parallel D_{\lceil t/128 \rceil - 1})$ 

```

5.1.3 The Authenticated Ciphers

We propose four instances for authenticated encryption with associated data: $\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$, $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$, $\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$, and $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$. Given a key K and a nonce N , each instance processes associated data $A \in \mathbb{F}_2^*$ and a message $M \in \mathbb{F}_2^*$, and returns a ciphertext C with $|C| = |M|$ together with an authentication tag T . Given (K, N, A, C, T) , decryption returns M when the tag is valid and the error symbol $\perp$ otherwise.

All four instances use a variant of the **Beetle** [51] mode described in [43], which in turn builds on **SpongeWrap** [52]. The instances differ in the width b and step counts ($n_{\text{slim}}, n_{\text{big}}$) of the underlying $\text{APEX}_b^{(n_{\text{slim}}, n_{\text{big}})}$ permutation, the rate r , the capacity c , and the tag length $|T|$. The big permutation is used for initialization, for the transition from associated data to the message, and for finalization. The slim permutation updates the intermediate state. Table 13 summarizes the parameters, conservative security claims, and recommended per-key data limits. As a generic capacity-birthday reference, $2^{c/2}$ rate blocks correspond to $r \cdot 2^{c/2}$ processed bits. For uniformity and simplicity, we set the recommended data limit to 2^{68} bytes for all instances except $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$, for which the limit is 2^{133} bytes.

In the **Beetle** mode, ciphertext generation and outer-state updating are coupled through the two-output feedback map ϕ . One output is emitted as a ciphertext block, while the other becomes the updated outer state. The map separates these two values by applying **shuffle** to the current outer state:

$$\text{shuffle}(X) = X_1 \parallel (X_0 \oplus X_1),$$

Table 13 Parameters, security claims, and recommended per-key data limits of the four AEAD instances. Each security claim applies jointly to confidentiality and integrity and is the integer-bit floor of the corresponding ideal-permutation bound. All lengths are in bits unless stated otherwise.

Instance	b	r	c	$ K $	$ N $	$ T $	Security claim	Recommended data limit (bytes)
$\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$	256	128	128	128	128	128	121	2^{68}
$\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$	384	256	128	128	256	128	120	2^{68}
$\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$	384	192	192	192	192	192	184	2^{68}
$\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$	512	256	256	256	256	256	248	2^{133}

where $X = X_0 \parallel X_1$ and $X_0, X_1 \in \mathbb{F}_2^{r/2}$. The encryption feedback function $\phi : (\mathbb{F}_2^r \times \mathbb{F}_2^r) \rightarrow (\mathbb{F}_2^r \times \mathbb{F}_2^r)$ is defined by

$$\phi(X, D) = (\phi_1(X, D), \phi_2(X, D)) = (\text{shuffle}(X) \oplus D, X \oplus D),$$

where the first component updates the rate part of the state and the second component is the ciphertext block. The decryption feedback function $\phi' : (\mathbb{F}_2^r \times \mathbb{F}_2^r) \rightarrow (\mathbb{F}_2^r \times \mathbb{F}_2^r)$ is defined by

$$\phi'(X, D) = (\phi'_1(X, D), \phi'_2(X, D)) = (\text{shuffle}(X) \oplus X \oplus D, X \oplus D),$$

where D is a ciphertext block and the second component recovers the corresponding plaintext block.

Write the APEX state as $S = S_L \parallel S_R$, where $|S_L| = r$ and $|S_R| = c$. After the feedback operation and the injection of the appropriate domain-separation constant, the rate-whitening transformation

$$(S_L, S_R) \mapsto (S_L \oplus \mathcal{W}_{c,r}(S_R), S_R)$$

is applied immediately before the next APEX permutation. It is omitted only for the initialization call. Thus, every noninitial update follows the order feedback, domain separation, rate whitening, and permutation. For the three instances with $r = c$, $\mathcal{W}_{c,r}$ is the identity. For $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$, write $S_R = (x, y) \in (\mathbb{F}_2^{64})^2$ and define $\mathcal{W}_{128,256}(x, y) = (x, y, x, y)$. Domain-separation constants distinguish complete and incomplete final blocks of associated data and messages. After the final big permutation, the key is XORed into the capacity and the tag is read from the capacity.

The encryption and decryption procedures for $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$ are formally specified in Algorithms 9 and 10, and depicted in Figures 9 and 10, respectively. The corresponding procedures for $\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$ are given in Algorithms 11 and 12. The procedures for $\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$ are given in Algorithms 13 and 14, while those for $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$ are given in Algorithms 15 and 16. For all three instances with $r = c$, $\mathcal{W}_{c,r}$ is the identity.

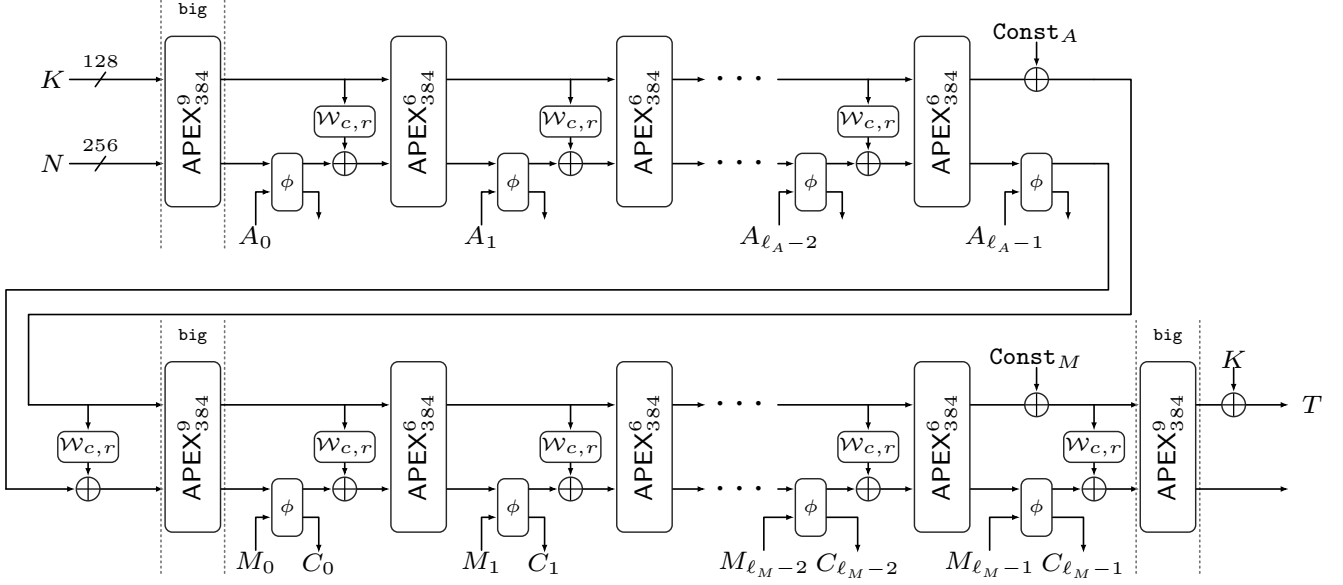


Fig. 9 The encryption procedure of $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$ with rate $r = 256$ and capacity $c = 128$. The rate-whitening map is $\mathcal{W}_{128,256}(x, y) = (x, y, x, y)$ for $x, y \in \mathbb{F}_2^{64}$.

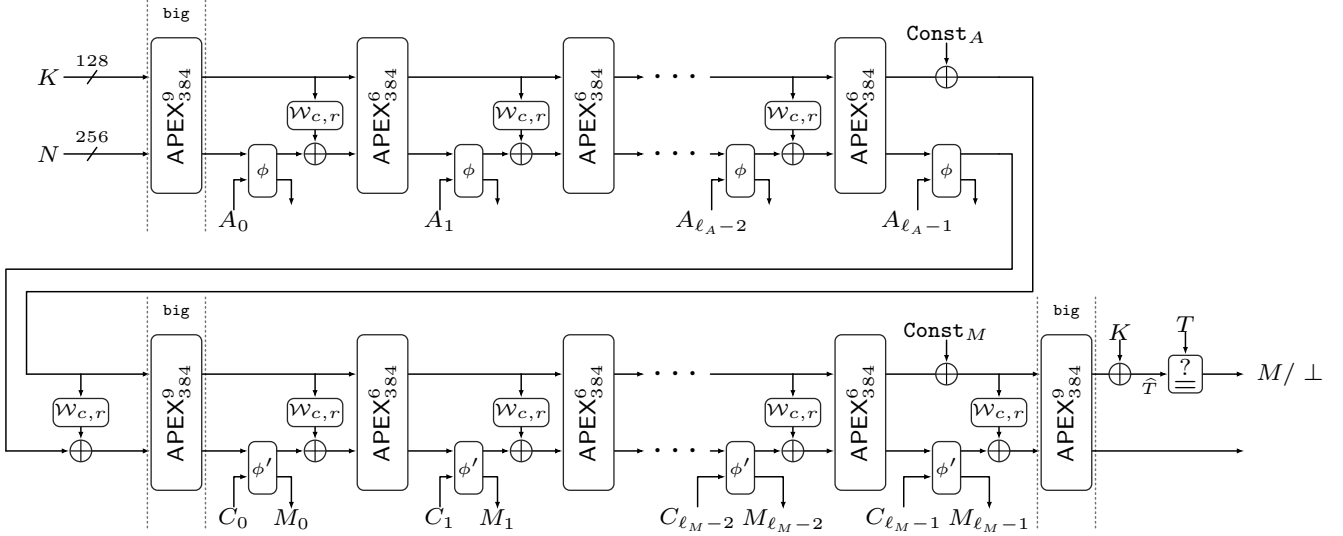


Fig. 10 The decryption procedure of $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$ with rate $r = 256$ and capacity $c = 128$. The rate-whitening map is $\mathcal{W}_{128,256}(x, y) = (x, y, x, y)$ for $x, y \in \mathbb{F}_2^{64}$. The recomputed tag $\hat{T}$ is compared with the received tag T before returning M or $\perp$.

Algorithm 9 Encryption for $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$.

Input: (K, N, A, M) , where $K \in \mathbb{F}_2^{128}$ is a key, $N \in \mathbb{F}_2^{256}$ is a nonce, $A \in \mathbb{F}_2^*$ is the associated data, and $M \in \mathbb{F}_2^*$ is the plaintext

Output: (C, T) , where $C \in \mathbb{F}_2^{|M|}$ is the ciphertext and $T \in \mathbb{F}_2^{128}$ is the authentication tag

if $A \neq \epsilon$ **then**

$A_0 \parallel A_1 \parallel \dots \parallel A_{\ell_A-1} \leftarrow A$

where $|A_i| = 256$ ($0 \leq i < \ell_A - 1$) and $1 \leq |A_{\ell_A-1}| \leq 256$

if $|A_{\ell_A-1}| < 256$ **then**

$A_{\ell_A-1} \leftarrow \text{pad}_{256}(A_{\ell_A-1})$

$\text{Const}_A \leftarrow 0^{125} \parallel 100 \in \mathbb{F}_2^{128}$

else

$\text{Const}_A \leftarrow 0^{125} \parallel 101 \in \mathbb{F}_2^{128}$

end if

end if

▷ Associated-data parsing and padding

if $M \neq \epsilon$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell_M-1} \leftarrow M$

where $|M_i| = 256$ ($0 \leq i < \ell_M - 1$) and $1 \leq |M_{\ell_M-1}| \leq 256$

$t \leftarrow |M_{\ell_M-1}|$

if $t < 256$ **then**

$M_{\ell_M-1} \leftarrow \text{pad}_{256}(M_{\ell_M-1})$

$\text{Const}_M \leftarrow 0^{125} \parallel 110 \in \mathbb{F}_2^{128}$

else

$\text{Const}_M \leftarrow 0^{125} \parallel 111 \in \mathbb{F}_2^{128}$

end if

end if

▷ Message parsing and padding

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9(N \parallel K)$, where $|S_L| = 256$ and $|S_R| = 128$

▷ State initialization

▷ Processing associated data

if $A \neq \epsilon$ **then**

for all $j \in [0, \ell_A - 2]$ **do**

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^6((\phi_1(S_L, A_j) \oplus \mathcal{W}_{128,256}(S_R)) \parallel S_R)$

end for

▷ Finalization if the message is empty

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi_1(S_L, A_{\ell_A-1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \text{Const}_A)) \parallel (S_R \oplus \text{Const}_A))$

end if

▷ Encrypting the message

if $M \neq \epsilon$ **then**

for all $j \in [0, \ell_M - 2]$ **do**

$C_j \leftarrow \phi_2(S_L, M_j)$

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^6((\phi_1(S_L, M_j) \oplus \mathcal{W}_{128,256}(S_R)) \parallel S_R)$

end for

$C_{\ell_M-1} \leftarrow \text{trunc}_t(\phi_2(S_L, M_{\ell_M-1}))$

▷ Finalization

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi_1(S_L, M_{\ell_M-1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \text{Const}_M)) \parallel (S_R \oplus \text{Const}_M))$

end if

return $(C_0 \parallel C_1 \parallel \dots \parallel C_{\ell_M-1}, S_R \oplus K)$

Algorithm 10 Decryption for $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$.

Input: (K, N, A, C, T) , where $K \in \mathbb{F}_2^{128}$ is a key, $N \in \mathbb{F}_2^{256}$ is a nonce, $A \in \mathbb{F}_2^*$ is the associated data, $C \in \mathbb{F}_2^*$ is the ciphertext, and $T \in \mathbb{F}_2^{128}$ is the authentication tag

Output: the plaintext $M \in \mathbb{F}_2^{|C|}$ if T is valid, and $\perp$ otherwise

▷ Associated-data parsing and padding

if $A \neq \epsilon$ **then**
 $A_0 \parallel A_1 \parallel \dots \parallel A_{\ell_A-1} \leftarrow A$
 where $|A_i| = 256$ ($0 \leq i < \ell_A - 1$) and $1 \leq |A_{\ell_A-1}| \leq 256$
 if $|A_{\ell_A-1}| < 256$ **then**
 $A_{\ell_A-1} \leftarrow \text{pad}_{256}(A_{\ell_A-1})$
 $\text{Const}_A \leftarrow 0^{125} \parallel 100 \in \mathbb{F}_2^{128}$
 else
 $\text{Const}_A \leftarrow 0^{125} \parallel 101 \in \mathbb{F}_2^{128}$
 end if
end if

▷ Ciphertext parsing and padding

if $C \neq \epsilon$ **then**
 $C_0 \parallel C_1 \parallel \dots \parallel C_{\ell_M-1} \leftarrow C$
 where $|C_i| = 256$ ($0 \leq i < \ell_M - 1$) and $1 \leq |C_{\ell_M-1}| \leq 256$
 $t \leftarrow |C_{\ell_M-1}|$
 if $t < 256$ **then**
 $C_{\ell_M-1} \leftarrow \text{pad}_{256}(C_{\ell_M-1})$
 $\text{Const}_M \leftarrow 0^{125} \parallel 110 \in \mathbb{F}_2^{128}$
 else
 $\text{Const}_M \leftarrow 0^{125} \parallel 111 \in \mathbb{F}_2^{128}$
 end if
end if

▷ State initialization

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9(N \parallel K)$, where $|S_L| = 256$ and $|S_R| = 128$

▷ Processing associated data

if $A \neq \epsilon$ **then**
 for all $j \in [0, \ell_A - 2]$ **do**
 $S_L \parallel S_R \leftarrow \text{APEX}_{384}^6((\phi_1(S_L, A_j) \oplus \mathcal{W}_{128,256}(S_R)) \parallel S_R)$
 end for

▷ Finalization if the ciphertext is empty

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi_1(S_L, A_{\ell_A-1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \text{Const}_A)) \parallel (S_R \oplus \text{Const}_A))$

end if

▷ Decrypting and authenticating

if $C \neq \epsilon$ **then**
 for all $j \in [0, \ell_M - 2]$ **do**
 $M_j \leftarrow \phi_2'(S_L, C_j)$
 $S_L \parallel S_R \leftarrow \text{APEX}_{384}^6((\phi_1'(S_L, C_j) \oplus \mathcal{W}_{128,256}(S_R)) \parallel S_R)$
 end for
 $M_{\ell_M-1} \leftarrow \text{trunc}_t(\phi_2'(S_L, C_{\ell_M-1}))$

▷ Finalization and tag verification

if $t < 256$ **then**
 $S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi_1(S_L, \text{pad}_{256}(M_{\ell_M-1})) \oplus \mathcal{W}_{128,256}(S_R \oplus \text{Const}_M)) \parallel (S_R \oplus \text{Const}_M))$
else
 $S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi_1'(S_L, C_{\ell_M-1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \text{Const}_M)) \parallel (S_R \oplus \text{Const}_M))$
end if
end if
if $S_R \oplus K = T$ **then**
 return $M_0 \parallel M_1 \parallel \dots \parallel M_{\ell_M-1}$
else
 return $\perp$
end if

Algorithm 11 Encryption for $\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$.

Input: (K, N, A, M) , where $K \in \mathbb{F}_2^{128}$ is a key, $N \in \mathbb{F}_2^{128}$ is a nonce, $A \in \mathbb{F}_2^*$ is the associated data, and $M \in \mathbb{F}_2^*$ is the plaintext

Output: (C, T) , where $C \in \mathbb{F}_2^{|M|}$ is the ciphertext and $T \in \mathbb{F}_2^{128}$ is the authentication tag

if $A \neq \epsilon$ **then**

$A_0 \parallel A_1 \parallel \dots \parallel A_{\ell_A-1} \leftarrow A$

where $|A_i| = 128$ ($0 \leq i < \ell_A - 1$) and $1 \leq |A_{\ell_A-1}| \leq 128$

if $|A_{\ell_A-1}| < 128$ **then**

$A_{\ell_A-1} \leftarrow \text{pad}_{128}(A_{\ell_A-1})$

$\text{Const}_A \leftarrow 0^{125} \parallel 100 \in \mathbb{F}_2^{128}$

else

$\text{Const}_A \leftarrow 0^{125} \parallel 101 \in \mathbb{F}_2^{128}$

end if

end if

▷ Associated-data parsing and padding

if $M \neq \epsilon$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell_M-1} \leftarrow M$

where $|M_i| = 128$ ($0 \leq i < \ell_M - 1$) and $1 \leq |M_{\ell_M-1}| \leq 128$

$t \leftarrow |M_{\ell_M-1}|$

if $t < 128$ **then**

$M_{\ell_M-1} \leftarrow \text{pad}_{128}(M_{\ell_M-1})$

$\text{Const}_M \leftarrow 0^{125} \parallel 110 \in \mathbb{F}_2^{128}$

else

$\text{Const}_M \leftarrow 0^{125} \parallel 111 \in \mathbb{F}_2^{128}$

end if

end if

▷ Message parsing and padding

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^8(N \parallel K)$, where $|S_L| = |S_R| = 128$

▷ State initialization

if $A \neq \epsilon$ **then**

for all $j \in [0, \ell_A - 2]$ **do**

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^6((\phi_1(S_L, A_j) \oplus S_R) \parallel S_R)$

end for

▷ Processing associated data

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^8((\phi_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \text{Const}_A) \parallel (S_R \oplus \text{Const}_A))$

end if

▷ Finalization if the message is empty

if $M \neq \epsilon$ **then**

for all $j \in [0, \ell_M - 2]$ **do**

$C_j \leftarrow \phi_2(S_L, M_j)$

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^6((\phi_1(S_L, M_j) \oplus S_R) \parallel S_R)$

end for

$C_{\ell_M-1} \leftarrow \text{trunc}_t(\phi_2(S_L, M_{\ell_M-1}))$

▷ Encrypting the message

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^8((\phi_1(S_L, M_{\ell_M-1}) \oplus S_R \oplus \text{Const}_M) \parallel (S_R \oplus \text{Const}_M))$

▷ Finalization

end if

return $(C_0 \parallel C_1 \parallel \dots \parallel C_{\ell_M-1}, S_R \oplus K)$

Algorithm 12 Decryption for $\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$.

Input: (K, N, A, C, T) , where $K \in \mathbb{F}_2^{128}$ is a key, $N \in \mathbb{F}_2^{128}$ is a nonce, $A \in \mathbb{F}_2^*$ is the associated data, $C \in \mathbb{F}_2^*$ is the ciphertext, and $T \in \mathbb{F}_2^{128}$ is the authentication tag

Output: the plaintext $M \in \mathbb{F}_2^{|C|}$ if T is valid, and $\perp$ otherwise

if $A \neq \epsilon$ **then**

$A_0 \parallel A_1 \parallel \dots \parallel A_{\ell_A-1} \leftarrow A$

where $|A_i| = 128$ ($0 \leq i < \ell_A - 1$) and $1 \leq |A_{\ell_A-1}| \leq 128$

if $|A_{\ell_A-1}| < 128$ **then**

$A_{\ell_A-1} \leftarrow \text{pad}_{128}(A_{\ell_A-1})$

$\text{Const}_A \leftarrow 0^{125} \parallel 100 \in \mathbb{F}_2^{128}$

else

$\text{Const}_A \leftarrow 0^{125} \parallel 101 \in \mathbb{F}_2^{128}$

end if

end if

▷ Associated-data parsing and padding

if $C \neq \epsilon$ **then**

$C_0 \parallel C_1 \parallel \dots \parallel C_{\ell_M-1} \leftarrow C$

where $|C_i| = 128$ ($0 \leq i < \ell_M - 1$) and $1 \leq |C_{\ell_M-1}| \leq 128$

$t \leftarrow |C_{\ell_M-1}|$

if $t < 128$ **then**

$C_{\ell_M-1} \leftarrow \text{pad}_{128}(C_{\ell_M-1})$

$\text{Const}_M \leftarrow 0^{125} \parallel 110 \in \mathbb{F}_2^{128}$

else

$\text{Const}_M \leftarrow 0^{125} \parallel 111 \in \mathbb{F}_2^{128}$

end if

end if

▷ Ciphertext parsing and padding

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^8(N \parallel K)$, where $|S_L| = |S_R| = 128$

▷ State initialization

if $A \neq \epsilon$ **then**

for all $j \in [0, \ell_A - 2]$ **do**

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^6((\phi_1(S_L, A_j) \oplus S_R) \parallel S_R)$

end for

▷ Processing associated data

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^8((\phi_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \text{Const}_A) \parallel (S_R \oplus \text{Const}_A))$

end if

▷ Finalization if the ciphertext is empty

if $C \neq \epsilon$ **then**

for all $j \in [0, \ell_M - 2]$ **do**

$M_j \leftarrow \phi'_2(S_L, C_j)$

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^6((\phi'_1(S_L, C_j) \oplus S_R) \parallel S_R)$

end for

$M_{\ell_M-1} \leftarrow \text{trunc}_t(\phi'_2(S_L, C_{\ell_M-1}))$

▷ Finalization and tag verification

if $t < 128$ **then**

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^8((\phi_1(S_L, \text{pad}_{128}(M_{\ell_M-1})) \oplus S_R \oplus \text{Const}_M) \parallel (S_R \oplus \text{Const}_M))$

else

$S_L \parallel S_R \leftarrow \text{APEX}_{256}^8((\phi'_1(S_L, C_{\ell_M-1}) \oplus S_R \oplus \text{Const}_M) \parallel (S_R \oplus \text{Const}_M))$

end if

end if

if $S_R \oplus K = T$ **then**

return $M_0 \parallel M_1 \parallel \dots \parallel M_{\ell_M-1}$

else

return $\perp$

end if

Algorithm 13 Encryption for $\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$.

Input: (K, N, A, M) , where $K \in \mathbb{F}_2^{192}$ is a key, $N \in \mathbb{F}_2^{192}$ is a nonce, $A \in \mathbb{F}_2^*$ is the associated data, and $M \in \mathbb{F}_2^*$ is the plaintext

Output: (C, T) , where $C \in \mathbb{F}_2^{|M|}$ is the ciphertext and $T \in \mathbb{F}_2^{192}$ is the authentication tag

if $A \neq \epsilon$ **then**

$A_0 \parallel A_1 \parallel \dots \parallel A_{\ell_A-1} \leftarrow A$

where $|A_i| = 192$ ($0 \leq i < \ell_A - 1$) and $1 \leq |A_{\ell_A-1}| \leq 192$

if $|A_{\ell_A-1}| < 192$ **then**

$A_{\ell_A-1} \leftarrow \text{pad}_{192}(A_{\ell_A-1})$

$\text{Const}_A \leftarrow 0^{188} \parallel 1000 \in \mathbb{F}_2^{192}$

else

$\text{Const}_A \leftarrow 0^{188} \parallel 1001 \in \mathbb{F}_2^{192}$

end if

end if

▷ Associated-data parsing and padding

if $M \neq \epsilon$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell_M-1} \leftarrow M$

where $|M_i| = 192$ ($0 \leq i < \ell_M - 1$) and $1 \leq |M_{\ell_M-1}| \leq 192$

$t \leftarrow |M_{\ell_M-1}|$

if $t < 192$ **then**

$M_{\ell_M-1} \leftarrow \text{pad}_{192}(M_{\ell_M-1})$

$\text{Const}_M \leftarrow 0^{188} \parallel 1010 \in \mathbb{F}_2^{192}$

else

$\text{Const}_M \leftarrow 0^{188} \parallel 1011 \in \mathbb{F}_2^{192}$

end if

end if

▷ Message parsing and padding

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9(N \parallel K)$, where $|S_L| = |S_R| = 192$

▷ State initialization

if $A \neq \epsilon$ **then**

for all $j \in [0, \ell_A - 2]$ **do**

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^6((\phi_1(S_L, A_j) \oplus S_R) \parallel S_R)$

end for

▷ Processing associated data

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \text{Const}_A) \parallel (S_R \oplus \text{Const}_A))$

end if

▷ Finalization if the message is empty

if $M \neq \epsilon$ **then**

for all $j \in [0, \ell_M - 2]$ **do**

$C_j \leftarrow \phi_2(S_L, M_j)$

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^6((\phi_1(S_L, M_j) \oplus S_R) \parallel S_R)$

end for

$C_{\ell_M-1} \leftarrow \text{trunc}_t(\phi_2(S_L, M_{\ell_M-1}))$

▷ Encrypting the message

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi_1(S_L, M_{\ell_M-1}) \oplus S_R \oplus \text{Const}_M) \parallel (S_R \oplus \text{Const}_M))$

▷ Finalization

end if

return $(C_0 \parallel C_1 \parallel \dots \parallel C_{\ell_M-1}, S_R \oplus K)$

Algorithm 14 Decryption for $\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$.

Input: (K, N, A, C, T) , where $K \in \mathbb{F}_2^{192}$ is a key, $N \in \mathbb{F}_2^{192}$ is a nonce, $A \in \mathbb{F}_2^*$ is the associated data, $C \in \mathbb{F}_2^*$ is the ciphertext, and $T \in \mathbb{F}_2^{192}$ is the authentication tag

Output: the plaintext $M \in \mathbb{F}_2^{|C|}$ if T is valid, and $\perp$ otherwise

if $A \neq \epsilon$ **then**

$A_0 \parallel A_1 \parallel \dots \parallel A_{\ell_A-1} \leftarrow A$

where $|A_i| = 192$ ($0 \leq i < \ell_A - 1$) and $1 \leq |A_{\ell_A-1}| \leq 192$

if $|A_{\ell_A-1}| < 192$ **then**

$A_{\ell_A-1} \leftarrow \text{pad}_{192}(A_{\ell_A-1})$

$\text{Const}_A \leftarrow 0^{188} \parallel 1000 \in \mathbb{F}_2^{192}$

else

$\text{Const}_A \leftarrow 0^{188} \parallel 1001 \in \mathbb{F}_2^{192}$

end if

end if

▷ Associated-data parsing and padding

if $C \neq \epsilon$ **then**

$C_0 \parallel C_1 \parallel \dots \parallel C_{\ell_M-1} \leftarrow C$

where $|C_i| = 192$ ($0 \leq i < \ell_M - 1$) and $1 \leq |C_{\ell_M-1}| \leq 192$

$t \leftarrow |C_{\ell_M-1}|$

if $t < 192$ **then**

$C_{\ell_M-1} \leftarrow \text{pad}_{192}(C_{\ell_M-1})$

$\text{Const}_M \leftarrow 0^{188} \parallel 1010 \in \mathbb{F}_2^{192}$

else

$\text{Const}_M \leftarrow 0^{188} \parallel 1011 \in \mathbb{F}_2^{192}$

end if

end if

▷ Ciphertext parsing and padding

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9(N \parallel K)$, where $|S_L| = |S_R| = 192$

▷ State initialization

▷ Processing associated data

if $A \neq \epsilon$ **then**

for all $j \in [0, \ell_A - 2]$ **do**

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^6((\phi_1(S_L, A_j) \oplus S_R) \parallel S_R)$

end for

▷ Finalization if the ciphertext is empty

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \text{Const}_A) \parallel (S_R \oplus \text{Const}_A))$

end if

▷ Decrypting and authenticating

if $C \neq \epsilon$ **then**

for all $j \in [0, \ell_M - 2]$ **do**

$M_j \leftarrow \phi'_2(S_L, C_j)$

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^6((\phi'_1(S_L, C_j) \oplus S_R) \parallel S_R)$

end for

$M_{\ell_M-1} \leftarrow \text{trunc}_t(\phi'_2(S_L, C_{\ell_M-1}))$

▷ Finalization and tag verification

if $t < 192$ **then**

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi_1(S_L, \text{pad}_{192}(M_{\ell_M-1})) \oplus S_R \oplus \text{Const}_M) \parallel (S_R \oplus \text{Const}_M))$

else

$S_L \parallel S_R \leftarrow \text{APEX}_{384}^9((\phi'_1(S_L, C_{\ell_M-1}) \oplus S_R \oplus \text{Const}_M) \parallel (S_R \oplus \text{Const}_M))$

end if

end if

if $S_R \oplus K = T$ **then**

return $M_0 \parallel M_1 \parallel \dots \parallel M_{\ell_M-1}$

else

return $\perp$

end if

Algorithm 15 Encryption for $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$.

Input: (K, N, A, M) , where $K \in \mathbb{F}_2^{256}$ is a key, $N \in \mathbb{F}_2^{256}$ is a nonce, $A \in \mathbb{F}_2^*$ is the associated data, and $M \in \mathbb{F}_2^*$ is the plaintext

Output: (C, T) , where $C \in \mathbb{F}_2^{|M|}$ is the ciphertext and $T \in \mathbb{F}_2^{256}$ is the authentication tag

if $A \neq \epsilon$ **then**

$A_0 \parallel A_1 \parallel \dots \parallel A_{\ell_A-1} \leftarrow A$

where $|A_i| = 256$ ($0 \leq i < \ell_A - 1$) and $1 \leq |A_{\ell_A-1}| \leq 256$

if $|A_{\ell_A-1}| < 256$ **then**

$A_{\ell_A-1} \leftarrow \text{pad}_{256}(A_{\ell_A-1})$

$\text{Const}_A \leftarrow 0^{251} \parallel 10000 \in \mathbb{F}_2^{256}$

else

$\text{Const}_A \leftarrow 0^{251} \parallel 10001 \in \mathbb{F}_2^{256}$

end if

end if

▷ Associated-data parsing and padding

if $M \neq \epsilon$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell_M-1} \leftarrow M$

where $|M_i| = 256$ ($0 \leq i < \ell_M - 1$) and $1 \leq |M_{\ell_M-1}| \leq 256$

$t \leftarrow |M_{\ell_M-1}|$

if $t < 256$ **then**

$M_{\ell_M-1} \leftarrow \text{pad}_{256}(M_{\ell_M-1})$

$\text{Const}_M \leftarrow 0^{251} \parallel 10010 \in \mathbb{F}_2^{256}$

else

$\text{Const}_M \leftarrow 0^{251} \parallel 10011 \in \mathbb{F}_2^{256}$

end if

end if

▷ Message parsing and padding

$S_L \parallel S_R \leftarrow \text{APEX}_{512}^{10}(N \parallel K)$, where $|S_L| = |S_R| = 256$

▷ State initialization

if $A \neq \epsilon$ **then**

for all $j \in [0, \ell_A - 2]$ **do**

$S_L \parallel S_R \leftarrow \text{APEX}_{512}^7((\phi_1(S_L, A_j) \oplus S_R) \parallel S_R)$

end for

▷ Processing associated data

$S_L \parallel S_R \leftarrow \text{APEX}_{512}^{10}((\phi_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \text{Const}_A) \parallel (S_R \oplus \text{Const}_A))$

end if

▷ Finalization if the message is empty

if $M \neq \epsilon$ **then**

for all $j \in [0, \ell_M - 2]$ **do**

$C_j \leftarrow \phi_2(S_L, M_j)$

$S_L \parallel S_R \leftarrow \text{APEX}_{512}^7((\phi_1(S_L, M_j) \oplus S_R) \parallel S_R)$

end for

$C_{\ell_M-1} \leftarrow \text{trunc}_t(\phi_2(S_L, M_{\ell_M-1}))$

▷ Encrypting the message

$S_L \parallel S_R \leftarrow \text{APEX}_{512}^{10}((\phi_1(S_L, M_{\ell_M-1}) \oplus S_R \oplus \text{Const}_M) \parallel (S_R \oplus \text{Const}_M))$

▷ Finalization

end if

return $(C_0 \parallel C_1 \parallel \dots \parallel C_{\ell_M-1}, S_R \oplus K)$

Algorithm 16 Decryption for $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$.

Input: (K, N, A, C, T) , where $K \in \mathbb{F}_2^{256}$ is a key, $N \in \mathbb{F}_2^{256}$ is a nonce, $A \in \mathbb{F}_2^*$ is the associated data, $C \in \mathbb{F}_2^*$ is the ciphertext, and $T \in \mathbb{F}_2^{256}$ is the authentication tag

Output: the plaintext $M \in \mathbb{F}_2^{|C|}$ if T is valid, and $\perp$ otherwise

```

if  $A \neq \epsilon$  then
     $A_0 \parallel A_1 \parallel \dots \parallel A_{\ell_A-1} \leftarrow A$ 
    where  $|A_i| = 256$  ( $0 \leq i < \ell_A - 1$ ) and  $1 \leq |A_{\ell_A-1}| \leq 256$ 
    if  $|A_{\ell_A-1}| < 256$  then
         $A_{\ell_A-1} \leftarrow \text{pad}_{256}(A_{\ell_A-1})$ 
         $\text{Const}_A \leftarrow 0^{251} \parallel 10000 \in \mathbb{F}_2^{256}$ 
    else
         $\text{Const}_A \leftarrow 0^{251} \parallel 10001 \in \mathbb{F}_2^{256}$ 
    end if
end if
    ▷ Associated-data parsing and padding

if  $C \neq \epsilon$  then
     $C_0 \parallel C_1 \parallel \dots \parallel C_{\ell_M-1} \leftarrow C$ 
    where  $|C_i| = 256$  ( $0 \leq i < \ell_M - 1$ ) and  $1 \leq |C_{\ell_M-1}| \leq 256$ 
     $t \leftarrow |C_{\ell_M-1}|$ 
    if  $t < 256$  then
         $C_{\ell_M-1} \leftarrow \text{pad}_{256}(C_{\ell_M-1})$ 
         $\text{Const}_M \leftarrow 0^{251} \parallel 10010 \in \mathbb{F}_2^{256}$ 
    else
         $\text{Const}_M \leftarrow 0^{251} \parallel 10011 \in \mathbb{F}_2^{256}$ 
    end if
end if
    ▷ Ciphertext parsing and padding

     $S_L \parallel S_R \leftarrow \text{APEX}_{512}^{10}(N \parallel K)$ , where  $|S_L| = |S_R| = 256$ 
    ▷ State initialization
    ▷ Processing associated data

    if  $A \neq \epsilon$  then
        for all  $j \in [0, \ell_A - 2]$  do
             $S_L \parallel S_R \leftarrow \text{APEX}_{512}^7((\phi_1(S_L, A_j) \oplus S_R) \parallel S_R)$ 
        end for
        ▷ Finalization if the ciphertext is empty

         $S_L \parallel S_R \leftarrow \text{APEX}_{512}^{10}((\phi_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \text{Const}_A) \parallel (S_R \oplus \text{Const}_A))$ 
    end if
    ▷ Decrypting and authenticating

    if  $C \neq \epsilon$  then
        for all  $j \in [0, \ell_M - 2]$  do
             $M_j \leftarrow \phi'_2(S_L, C_j)$ 
             $S_L \parallel S_R \leftarrow \text{APEX}_{512}^7((\phi'_1(S_L, C_j) \oplus S_R) \parallel S_R)$ 
        end for
         $M_{\ell_M-1} \leftarrow \text{trunc}_t(\phi'_2(S_L, C_{\ell_M-1}))$ 
        ▷ Finalization and tag verification

        if  $t < 256$  then
             $S_L \parallel S_R \leftarrow \text{APEX}_{512}^{10}((\phi_1(S_L, \text{pad}_{256}(M_{\ell_M-1})) \oplus S_R \oplus \text{Const}_M) \parallel (S_R \oplus \text{Const}_M))$ 
        else
             $S_L \parallel S_R \leftarrow \text{APEX}_{512}^{10}((\phi'_1(S_L, C_{\ell_M-1}) \oplus S_R \oplus \text{Const}_M) \parallel (S_R \oplus \text{Const}_M))$ 
        end if
    end if
    if  $S_R \oplus K = T$  then
        return  $M_0 \parallel M_1 \parallel \dots \parallel M_{\ell_M-1}$ 
    else
        return  $\perp$ 
    end if

```

5.2 Design Rationale

5.2.1 Choice of Modes

5.2.1.1 Established-mode strategy.

The purpose of these constructions is to demonstrate the use of the APEX permutation family in lightweight hashing and authenticated encryption. We therefore employ established modes with well-understood security properties

instead of introducing new modes of operation. This choice separates the evaluation of the underlying permutation from that of the surrounding mode and permits direct comparison with existing permutation-based designs.

5.2.1.2 The Mode Used for Hashing and XOFs.

The hash functions $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$ adopt a variant of the sponge-based hashing mode used by Esch [43]. Their corresponding XOFs, $\text{APEX}_{384}^{(6,9)}\text{-HASH-XO-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-XO-384}$, reuse the complete absorption procedure. They differ only in their domain-separation constants and in continuing the squeezing phase until the requested output length is reached.

In this variant, message blocks are injected indirectly through $\mathcal{M}_{n_b/2}$. This preprocessing breaks the alignment between the 128-bit rate and the parallel AFS-64 layer, preventing a differential trail from bypassing the first or last AFS-64 layer merely through branch-aligned message injection. Moreover, $\mathcal{M}_3$ and $\mathcal{M}_4$ have differential branch number four, while only two 64-bit message branches are supplied to either map. Every nonzero injected message difference therefore activates at least two output branches. Following the argument of Lemma 3.1.1 in [43], at least two double AFS-64 segments are active at both the introduction and cancellation boundaries. Since the two-iteration MEDCP of AFS-64 is at most 2^{-37} , any differential trail introduced and subsequently cancelled through indirect message injections, with at least one complete APEX permutation invocation between them, has probability at most $(2^{-37})^4 = 2^{-148}$. Since $\mathcal{M}_{n_b/2}$ is invertible, the construction is equivalent to a regular sponge in which the underlying permutation is conjugated by the corresponding linear map.

5.2.1.3 The Mode Used for Authenticated Encryption.

For the four AEAD instances $\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$, $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$, $\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$, and $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$, we employ the variant of the Beetle mode [51] used by SCHWAEMM [43]. The Beetle mode was designed specifically to minimize hardware area by permitting a smaller permutation state for a target security level. For an ideal b -bit permutation whose state is partitioned into an r -bit rate and a c -bit capacity, its combined feedback raises the provable AE-security bound from the $c/2$ -bit capacity birthday bound of a conventional duplex sponge to $\min\{c - \log_2 r, b/2, r\}$ bits, without imposing an a priori birthday-bound restriction on the number of decryption queries. This tighter bound permits a comparatively small capacity and internal state while retaining a high rate and a security level close to c bits. Its two-output feedback map ϕ derives the ciphertext block and the next outer state from distinct combinations of the current outer state and message block. This variant of the Beetle mode further introduces the rate-whitening map $\mathcal{W}_{c,r}$. Rate whitening is not part of the original Beetle feedback, but it can be absorbed into the definition of the underlying permutation and is therefore compatible with the Beetle framework. In sponge-based authenticated encryption, the rate is visible to the adversary, whereas the capacity remains hidden. If the rate is aligned with the parallel ARX-box layer, the adversary may directly evaluate part of the permutation. The copied half of the sparse Feistel linear layer may allow this partial evaluation to extend across portions of two consecutive steps. We therefore XOR $\mathcal{W}_{c,r}(S_R)$ into the rate before every noninitial permutation call. Consequently, the rate entering the permutation depends on the hidden capacity, so such a partial evaluation first requires guessing capacity information. Together with the domain-separation constants, these operations separate associated-data processing, message processing, complete and incomplete final blocks, and finalization. The four instances therefore share one mode and differ only in the underlying APEX width and step counts, rate, capacity, and key, nonce, and tag lengths.

5.2.1.4 Position of message injection.

Across the HASH, XOF, and AEAD constructions, absorbed data are assigned to state branches beginning with the left half. For HASH and XOF, the image of a message block under $\mathcal{M}_3$ or $\mathcal{M}_4$ is injected into the complete left half. For AEAD, the rate first contains all branches of the left half. When $r > b/2$, it is completed with as many branches from the right half as required. This placement has three advantages. First, the injected branches enter the Feistel mixing function in the same step, promoting rapid diffusion through the state. Second, after the AFS-64 layer, the left half is copied to the right half by the linear layer. An injected contribution therefore passes through two consecutive AFS-64 layers along the copied path, which reduces the attacker’s control over its propagation. Third, immediately before each subsequent absorption, the left half already contains the output of the preceding Feistel mixing and consequently depends on contributions from both halves of the state. An attacker seeking a collision or a cancellation across absorptions must therefore control both the injected difference and these mixed contributions. This makes backward propagation and the maintenance of a prescribed pattern immediately before injection more difficult.

5.2.2 Choice of Parameters

5.2.2.1 Hash and XOF parameters.

Under the generic sponge security bounds, a construction with capacity c and a d -bit digest provides a collision-security target of $\min\{d/2, c/2\}$ and preimage and second-preimage targets of $\min\{d, c/2\}$ [53]. We require each of these targets to be at least 128 bits. Consequently, both d and c must be at least 256 bits. To retain a practical absorption rate while keeping the internal state compact, we use a common rate of $r = 128$ bits for both profiles. The minimum width of the underlying APEX permutation is therefore $b = r + c = 384$ bits. This yields $(b, r, c, d) = (384, 128, 256, 256)$ for $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$. For the higher 192-bit security level, we retain $r = 128$ and increase both the capacity and the digest length to 384 bits. The resulting state width is $b = 512$ bits, giving $(b, r, c, d) = (512, 128, 384, 384)$ for $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$. The two parameter sets thus provide 128-bit and 192-bit targets, respectively, for collision, preimage, and second-preimage resistance. The XOFs $\text{APEX}_{384}^{(6,9)}\text{-HASH-XO-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-XO-384}$ are obtained directly from $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$, respectively, and therefore inherit the state size, rate, capacity, and input limit of the corresponding HASH profile. For an output of t bits, the two XOFs provide collision-security targets of $\min\{128, t/2\}$ and $\min\{192, t/2\}$, respectively, and preimage and second-preimage targets of $\min\{128, t\}$ and $\min\{192, t\}$, respectively.

The message-domain constant Const_M is encoded according to the compact convention of [43]. Each value has the width $b/2$ of the transformed message-injection half and is zero except for its final three bits. The first of these bits identifies the function domain, with 0 for HASH and 1 for HASH-XO. The final two bits are 01 for a padded final block and 10 for an unpadded complete final block. The four resulting values of Const_M therefore have suffixes 001, 010, 101, and 110. They are pairwise distinct and nonzero, separating the HASH and HASH-XO domains as well as the two final-block encodings before the big permutation. These values provide a transparent domain encoding rather than secret or pseudorandom material.

The generic sponge bound reaches the capacity birthday regime after about $2^{c/2}$ calls to the underlying permutation [53]. We therefore limit the total absorbed input to at most $2^{c/2}$ rate blocks, or equivalently $r \cdot 2^{c/2}$ bits. Since $r = 128$, these limits are exactly 2^{132} and 2^{196} bytes for the two parameter pairs. The corresponding values are reported in Tables 11 and 12.

5.2.2.2 AEAD parameters.

The four AEAD profiles follow a common state layout. We set $b = r + c$, $|N| = r$, and $|K| = |T| = c$. Consequently, the public nonce fills the rate part and the secret key fills the capacity part during initialization, so that $N \parallel K$ occupies the complete state without padding or truncation. The tag length is aligned with both the key length and the capacity. These rules give all four profiles the same initialization and finalization structure.

Within this common layout, the profiles address four different design goals. The $\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$ profile is the compact choice and uses the smallest state compatible with 128-bit keys and tags. The $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$ profile is the throughput-oriented 128-bit choice. It doubles the rate from 128 to 256 bits while retaining 128-bit keys and tags. The $\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$ profile uses equal 192-bit rate and capacity to provide an intermediate, balanced parameter set. The $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$ profile likewise uses equal 256-bit rate and capacity and serves as the high-security member of the family, with 256-bit keys and tags.

The AEAD domain-separation constants Const_A and Const_M follow the encoding scheme of [43]. Let $q = c/64$ denote the number of 64-bit branches in the c -bit capacity. Interpreted as c -bit words, $\text{Const}_A = u_A \oplus (1 \ll q) \in \mathbb{F}_2^c$, where $u_A = 0$ for a padded associated-data block and $u_A = 1$ for a complete associated-data block. Similarly, $\text{Const}_M = u_M \oplus (1 \ll q) \in \mathbb{F}_2^c$, where $u_M = 2$ for a padded message or ciphertext block and $u_M = 3$ for a complete message or ciphertext block. Equivalently, a marker bit is set at position q , while the two least significant bits encode the data type and final-block status. This gives the suffix sets $\{100, 101, 110, 111\}$ for $c = 128$, $\{1000, 1001, 1010, 1011\}$ for $c = 192$, and $\{10000, 10001, 10010, 10011\}$ for $c = 256$. The marker ensures that all four constants are nonzero, and the two-bit field separates their processing domains. Encryption and decryption use the same encoding.

The security analysis of Beetle [51] shows that, under the ideal-permutation assumption used for the mode analysis, a construction with a b -bit permutation, an r -bit rate, and a c -bit capacity has the generic joint confidentiality-and-integrity bound $\sigma_{\text{AE}}(b, r, c) = \min\{c - \log_2 r, b/2, r\}$. We use this mode-level bound as the security target for selecting the AEAD parameters. For all four profiles, $c - \log_2 r$ is the active term in this minimum. Increasing the rate therefore improves the amount of data processed per permutation call at the cost of a logarithmic reduction in the mode bound, whereas increasing the capacity raises both the security target and the state size. Substitution

gives 121, 120, $192 - \log_2 192 \approx 184.42$, and 248 bits for the four profiles in the order listed above. Table 13 reports their integer-bit floors, namely 121, 120, 184, and 248 bits.

We select the per-key data limits separately using the generic capacity-birthday scale of $2^{c/2}$ rate blocks as a conservative reference. The corresponding amounts are $r \cdot 2^{c/2}$ processed bits, or 2^{68} , 2^{69} , $3 \cdot 2^{99}$, and 2^{133} bytes for the four profiles, respectively. To give the first three profiles one simple usage rule, we set their common limit to the smallest of these values, 2^{68} bytes. This is the reference value for the compact profile and a conservative reduction for the throughput-oriented and 192-bit profiles. The 256-bit profile retains its 2^{133} -byte reference value.

5.2.3 Choice of Step Numbers

Following the slim-and-big permutation strategy described in [43], our small-state hash, XOF, and AEAD constructions use two versions of the underlying APEX permutation with different step counts, referred to as the slim and big versions. The slim version is used in the performance-critical processing of nonfinal message blocks and between consecutive output blocks, which dominate the throughput for long inputs. The big version is reserved for security-sensitive positions: initialization, after the final absorbed block, at phase transitions, and during finalization. At these positions, it initializes the state, separates processing domains, binds the complete absorbed input, or precedes the extraction of a digest or authentication tag. We therefore use a larger step count at these positions to provide additional security margin. For each state width, all HASH, XOF, and AEAD profiles use the same pair $(n_{\text{slim}}, n_{\text{big}})$, chosen to satisfy the strongest security requirement among the profiles at that width. This keeps bulk processing efficient while providing additional security margin at initialization, phase transitions, and finalization.

5.2.3.1 Slim versions.

The slim versions of the APEX permutations are designed to resist distinguishers in which the adversary can control only the outer, or rate, part of the state when the permutations are employed in the sponge-based modes considered here. Different security bounds must be considered depending on whether a slim permutation is used in an AEAD profile or in a HASH or XOF profile.

For $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$, and therefore also for their XOF counterparts, we consider rate-to-rate differential trails (both the input and output differences are restricted to the rate). Although the external message rate is 128 bits, the indirect injection through $\mathcal{M}_3$ or $\mathcal{M}_4$ maps each message block into one half of the state. The rate at the permutation boundary is therefore taken to be $b/2$. The corresponding bounds are given in Table 15.

For the AEAD profiles, the domain-separation constant XORed for the final block lies in the inner, or capacity, part of the state. An adversary may therefore be able to introduce an inner-part difference through the final-block domain separation. To cover this possibility, we consider rate-to-anything differential trails (the input difference is restricted to the rate, while the output difference may occupy the complete state). The corresponding bounds are reported in Table 15. For the same reason, the big permutation is used to separate associated-data processing from message processing, since the last, possibly padded, associated-data block may introduce a difference into the inner part through its domain-separation constant.

For linear attacks on the HASH and XOF profiles, we likewise use the rate-to-rate bounds in Table 16. For the AEAD profiles, the rate-whitening layer can introduce linear masks into the inner part. We must therefore use the unrestricted bounds for the complete permutation given in Table 4. The differential security target for a HASH or XOF profile is $c/2 + r$, where c denotes the capacity and $r = 128$ is the external message rate, while its linear security target is $c/2$ bits. For an AEAD profile, the required linear target is determined by the recommended per-key data limit in Table 13.

$\text{APEX}_{256}^{(6,8)}$ is used only in $\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$. For this profile, we conservatively require a 128-bit differential security level for the underlying permutation. Table 15 shows that four steps are sufficient to bound the probability of every rate-to-anything differential trail by 2^{-128} . For linear attacks, the recommended 2^{68} -byte data limit corresponds to 2^{64} rate blocks. Recall that a linear approximation with absolute correlation C has an estimated data complexity of at least $1/C^2$. It is therefore sufficient to ensure $C \leq 2^{-32}$. The unrestricted linear bound in Table 4 reaches this threshold after three steps. The longest distinguishers found in the other analyses cover no more than four complete steps (see Section 3.3.3).

$\text{APEX}_{384}^{(6,9)}$ is used in four profiles. For $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$, we conservatively require a 128-bit differential security level for the underlying permutation. The rate is $r = 256$, and the recommended per-key data limit is 2^{68} bytes. Four steps are sufficient for rate-to-anything differential trails according to Table 15, while three steps satisfy the linear threshold determined by the data limit according to Table 4. For $\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$, we

conservatively require a 192-bit differential security level for the underlying permutation. The rate is $r = 192$, and the same data limit is used. Five steps are sufficient for rate-to-anything differential trails, and three steps are sufficient for linear trails.

For $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$, the required differential target is $c/2 + r = 256$ bits, while the required linear security target is 128 bits. Owing to the indirect message injection, the effective rate at the permutation boundary is $b/2 = 192$ bits. Tables 15 and 16 show that five steps are sufficient for differential trails and four steps are sufficient for linear trails. The XOF $\text{APEX}_{384}^{(6,9)}\text{-HASH-XO-256}$ inherits the same requirements. The longest distinguishers found in the other analyses cover no more than four complete steps (see Section 3.3.3).

Finally, APEX_{512} is used in three profiles. For $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$, we conservatively require a 256-bit differential security level for the underlying permutation. The rate is $r = 256$, and the 2^{133} -byte data limit corresponds to 2^{128} rate blocks. Six steps are sufficient for rate-to-anything differential trails according to Table 15. For linear attacks, the data limit requires $C \leq 2^{-64}$, which is reached after four steps in Table 4.

For $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$, the differential target is $c/2 + r = 320$ bits, while the required linear security target is 192 bits. The indirect injection gives an effective rate of $b/2 = 256$ bits. Tables 15 and 16 show that six steps are sufficient for differential trails and four steps are sufficient for linear trails. The XOF $\text{APEX}_{512}^{(7,10)}\text{-HASH-XO-384}$ inherits the same requirements. The longest distinguishers found in the other analyses cover no more than four complete steps (see Section 3.3.3).

5.2.3.2 Big versions.

For the big versions of the APEX permutations, our design target is that no distinguisher should have both time and data complexities below $2^{b/2}$, where b is the state width. Under the differential and linear LTS bounds, satisfying this target requires at least 5, 5, and 6 steps for APEX_{256} , APEX_{384} , and APEX_{512} , respectively (see Tables 3 and 4). All other forms of cryptanalysis that we evaluated cover fewer steps.

For the final choice of the number of steps, we add three steps as a security margin for APEX_{256} and four steps for APEX_{384} and APEX_{512} , thus choosing $n_{\text{big}} = 8, 9, 10$, respectively. All three widths reach full diffusion after three steps, as reported in Section 3.3.3.6. Thus, the added margin contains the three steps required for full diffusion in every case, together with one additional step for the 384- and 512-bit widths. These wider permutations support higher-security profiles, and the extra step has limited effect on long-message performance because the big versions are invoked only at mode boundaries rather than for every bulk block. A conservative choice is also appropriate because these calls include initialization with a secret key in the AEAD schemes.

Overall, this gives security margins of approximately 60%, 80%, and 67% for APEX_{256} , APEX_{384} , and APEX_{512} , respectively.

5.3 Security Analysis

The security analysis is organized in three parts. We first state the security claims for the HASH, XOF, and AEAD instances. We then consider attacks on the underlying APEX permutations and attacks on the modes of operation.

5.3.1 Security Claims

5.3.1.1 For the HASH and XOF instances.

For $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$, we claim security levels of $c/2$ bits against collision, preimage, and second-preimage attacks, where c is both the capacity and the digest length. Thus, the claimed security levels are 128 and 192 bits, respectively. These claims also cover length-extension attacks. We limit the total absorbed input to $2^{c/2}$ rate blocks, or equivalently $r \cdot 2^{c/2}$ bits, since collisions are expected beyond this amount of data. Accordingly, a cryptanalytic result violates these claims only if its time complexity is at most $2^{c/2}$ evaluations of the underlying permutation or its inverse.

For the corresponding XOFs $\text{APEX}_{384}^{(6,9)}\text{-HASH-XO-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-XO-384}$ with an output of t bits, the claimed security level is $\min\{c/2, t/2\}$ bits against collision attacks and $\min\{c/2, t\}$ bits against preimage and second-preimage attacks. The maximum permitted output length t is $r \cdot 2^{c/2}$ bits, equal to the corresponding input-data limit expressed in bits.

5.3.1.2 For the AEAD instances.

The security bound of the Beetle mode [51] is $\min\{r, (r+c)/2, c - \log_2 r\}$ bits for both confidentiality under adaptive chosen-plaintext attacks and integrity under adaptive forgery attempts, where r and c denote the rate and capacity. Our claims are made in the nonce-respecting setting: a nonce must not be reused for two encryption queries under the same key.

Following this bound and the parameters in Table 13, we claim 121 bits of security for $\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$, 120 bits for $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$, and 184 bits for $\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$. For each of these instances, at most 2^{68} bytes of data may be processed under one key. For $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$, we claim 248 bits of security and permit at most 2^{133} bytes of data under one key. Thus, an attack violating a σ -bit claim above must use at most 2^σ evaluations of the underlying permutation or its inverse and remain within the corresponding per-key data limit.

5.3.1.3 Nonce-misuse setting.

The preceding claims do not apply when a nonce is reused. Duplexed-sponge authenticated ciphers can be vulnerable to chosen-plaintext decryption and semi-universal forgery attacks in this setting [54]. For example, encrypting a zero rate block under a fixed (K, N, A) reveals the current outer state through the ciphertext, and this information can be used to decrypt another ciphertext obtained under the same tuple. The big initialization permutation and the slim permutations used during absorption are expected to make full-state recovery difficult in practice. Accordingly, the construction is expected to retain some practical resistance to key-recovery attacks even under nonce reuse. In addition, unique associated data may provide effective input uniqueness even if the nonce is repeated. These observations do not form part of our formal security claim. We therefore require a unique nonce for every encryption under a given key.

5.3.1.4 Known-key attacks.

The near-capacity security bound of Beetle applies in the secret-key setting. If the key is known, a classical meet-in-the-middle attack on the sponge state reduces collision and preimage security to approximately half the capacity when the tag is extracted in one step. This birthday-complexity limitation is common to sponge-based modes. We do not identify an AEAD use case requiring known-key security and make no known-key security claim for the four AEAD instances.

5.3.2 Attacks on the Underlying Permutations

The full-state cryptanalysis of the underlying APEX permutation family is presented in Section 3.3. Table 14 summarizes part of the results for the three state widths used by the constructions in this section.

Table 14 This table lists upper bounds on the number of steps for which we found an attack, or for which the differential and linear bounds are too low to guarantee $b/2$ -bit security, where b denotes the state width, with respect to several attack vectors.

Attack	Ref.	APEX ₂₅₆	APEX ₃₈₄	APEX ₅₁₂
Differential cryptanalysis	[55]	4	4	5
Linear cryptanalysis	[56]	3	4	4
Boomerang attacks	[44]	3	4	4
Truncated differentials	[57]	2	2	3
Yoyo games	[47]	4	4	4
Impossible differentials	[58, 59]	4	4	4
Zero-correlation linear cryptanalysis	[60]	4	4	4
Integral and division property	[40, 61, 62]	4	4	4
	# steps n_{slim}	6	6	7
	# steps n_{big}	8	9	10

5.3.3 Attacks on the Modes

The HASH, XOF, and AEAD interfaces restrict the differences and linear masks that an adversary can introduce and observe. We therefore refine the full-state analysis according to these mode-specific restrictions.

5.3.3.1 Message injection.

For HASH and XOF, each 128-bit message block is mapped by $\mathcal{M}_3$ or $\mathcal{M}_4$ and injected into the left half of the state. For AEAD, the adversary controls the rate part of the state. When $r > b/2$, this part contains the complete left half and the required number of branches from the right half. Moreover, the final-block domain-separation constant is injected into the capacity and may introduce an inner-part difference at a phase boundary.

5.3.3.2 Differential attacks.

To analyze the differential security of the sponge-based constructions, we consider vanishing absorbed trails, in which a nonzero difference is introduced during absorption and subsequently returns to zero. Let $P_{\text{van}}(a)$ denote the maximum probability of such a trail spanning a absorption intervals, and let U_{n_s} be an upper bound on the probability of any differential through one n_s -step permutation call. If no absorbed difference cancels an existing state difference, then $P_{\text{van}}(a) \leq U_{n_s}^a$.

Let κ denote the target security level. Once $U_{n_s}^2 < 2^{-\kappa}$, every trail passing through at least two permutation calls without cancellation already has probability below $2^{-\kappa}$. Hence, a vanishing absorbed trail with probability greater than $2^{-\kappa}$ must use the next absorption to cancel the state difference produced after the preceding absorption. The first absorbed difference is restricted to the rate. Immediately before the second absorption, the state difference must again be restricted to the rate so that it can be cancelled by the next absorbed block. Consequently, it is sufficient to search for rate-to-rate differential trails.

For $\text{APEX}_{384}^{(6,9)\text{-HASH-256}}$ and $\text{APEX}_{512}^{(7,10)\text{-HASH-384}}$, and likewise for their XOF counterparts, the external message rate is 128 bits. However, indirect injection through $\mathcal{M}_3$ and $\mathcal{M}_4$ maps a message difference to the leftmost three and four 64-bit branches, respectively. The effective rate at the permutation boundary is therefore $192 = b/2$ bits for the 384-bit profile and $256 = b/2$ bits for the 512-bit profile. We accordingly search for rate-to-rate trails over the complete left half of the state.

This search is conservative. A valid endpoint difference cannot be an arbitrary difference in the left half, but must belong to the image of the corresponding indirect-injection map, namely $\mathcal{M}_3(\delta \parallel 0^{64})$ or $\mathcal{M}_4(\delta \parallel 0^{128})$ for some $\delta \in \mathbb{F}_2^{128}$. Enlarging these image spaces to the complete left half may admit trails that cannot be realized by any 128-bit message difference. The maximum differential probability obtained from this relaxed search may therefore be higher than the probability attainable through the actual mode interface. Equivalently, the reported differential weights may underestimate the security of the construction. The resulting bounds are thus conservative and may overestimate the strength of an attack, but not the security provided by the mode.

For AEAD, we instead consider rate-to-anything trails. The input difference is restricted to the rate, while the output difference may occupy the complete state. This accounts for the capacity difference that may be introduced by the final-block domain-separation constant. The long-trail search methods developed in [22, 36, 43] accommodate these restricted trails with only a minor modification. We restrict the admissible input and output branch patterns in the truncated-trail model, while leaving the long-trail decomposition and the local AFS-64 bounds unchanged. This yields the mode-specific differential long-trail bounds reported in Table 15.

5.3.3.3 Linear attacks.

For HASH and XOF, we analogously consider rate-to-rate linear trails, with both endpoint masks restricted to the effective rate. The search proceeds analogously to the differential case. We restrict the admissible endpoint mask patterns in the truncated-trail model and evaluate the resulting long-trail decompositions using the multi-iteration MELCC bounds of AFS-64. The resulting rate-to-rate linear long-trail bounds for HASH and XOF are reported in Table 16.

For AEAD, the rate-whitening map $\mathcal{W}_{c,r}$ XORs a linear image of the capacity into the rate. Under backward mask propagation, a nonzero rate mask may therefore induce a nonzero capacity mask. We consequently use the unrestricted linear bounds of the complete permutation reported in Table 4. An approximation with absolute correlation C has an estimated data complexity of at least C^{-2} . The required linear bounds are therefore determined together with the per-key data limits of the corresponding instances.

Table 15 Mode-specific differential long-trail weights. Each entry gives $-\log_2 p$, where p is an upper bound on the probability of an n_s -step differential trail of APEX_b satisfying the input- and output-difference restrictions indicated in the Type column. The type $b \rightarrow b$ denotes unrestricted full-state trails, $r \rightarrow r$ restricts both the input and output differences to the rate, and $r \rightarrow b$ restricts only the input difference to the rate. The symbol $\emptyset$ means that no such trail exists in the model.

b	r	c (security)	Type	3	4	5	6	7	8
256	192	64	$b \rightarrow b$	86	116	168	205	236	266
			$r \rightarrow r$	94	131	168	205	242	279
			$r \rightarrow b$	86	131	168	205	236	279
	128	128	$r \rightarrow r$	111	148	222	224	261	298
			$r \rightarrow b$	111	140	170	207	255	285
	64	192	$r \rightarrow r$	$\emptyset$	148	222	226	296	298
384	256	128	$b \rightarrow b$	96	136	214	243	283	323
			$r \rightarrow r$	131	177	214	243	316	360
			$r \rightarrow b$	96	168	214	243	283	360
	192	192	$r \rightarrow r$	148	185	296	298	335	372
			$r \rightarrow b$	148	177	214	254	331	364
	128	256	$r \rightarrow r$	148	185	296	300	370	372
			$r \rightarrow b$	148	185	214	254	331	371
	64	320	$r \rightarrow r$	$\emptyset$	185	296	302	370	372
			$r \rightarrow b$	148	185	214	254	331	371
	512	256	$b \rightarrow b$	106	156	251	280	330	380
			$r \rightarrow r$	185	222	296	370	407	446
			$r \rightarrow b$	158	205	251	301	390	438
		384	$r \rightarrow r$	$\emptyset$	222	296	370	407	446
			$r \rightarrow b$	158	205	251	301	390	446
		448	$r \rightarrow r$	$\emptyset$	222	370	378	444	446
			$r \rightarrow b$	185	222	251	301	407	446

Table 16 Mode-specific linear long-trail weights. Each entry gives $-\log_2 C$, where C is an upper bound on the expected absolute correlation of an n_s -step linear trail of APEX_b satisfying the input- and output-mask restrictions indicated in the Type column. The type $b \rightarrow b$ denotes unrestricted full-state trails, whereas $r \rightarrow r$ restricts both the input and output masks to the indicated rate. The symbol $\emptyset$ means that no such trail exists in the model.

b	r	c (security)	Type	3	4	5	6	7
256	192	64	$b \rightarrow b$	48	66	92	112	130
			$r \rightarrow r$	48	66	92	112	130
			$r \rightarrow r$	48	82	108	126	130
			$r \rightarrow r$	$\emptyset$	82	122	126	162
384	128	256	$b \rightarrow b$	54	78	116	132	156
			$r \rightarrow r$	54	78	116	140	156
			$r \rightarrow r$	54	102	134	154	156
			$r \rightarrow r$	84	102	148	168	188
			$r \rightarrow r$	$\emptyset$	102	162	168	202
512	128	384	$b \rightarrow b$	60	90	136	152	182
			$r \rightarrow r$	60	122	156	180	182
			$r \rightarrow r$	$\emptyset$	122	164	188	224

5.4 Implementation

We provide reference and optimized C implementations of the hash functions $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ and $\text{APEX}_{512}^{(7,10)}\text{-HASH-384}$ and of the AEAD schemes $\text{APEX}_{256}^{(6,8)}\text{-AEAD-128/128}$, $\text{APEX}_{384}^{(6,9)}\text{-AEAD-256/128}$, $\text{APEX}_{384}^{(6,9)}\text{-AEAD-192/192}$, and $\text{APEX}_{512}^{(7,10)}\text{-AEAD-256/256}$. We also provide assembler implementations of $\text{APEX}_{256}^{n_s}$, $\text{APEX}_{384}^{n_s}$, and $\text{APEX}_{512}^{n_s}$ for 8-bit AVR ATmega and 32-bit ARM Cortex-M microcontrollers.

Implementation environment. All execution times for the ARM implementations were determined using the cycle-accurate instruction-set simulator of Keil μ Vision with a generic Cortex-M3 model as the target device, and the C implementation was compiled with ARMCC 5.06 at optimization level -O2. All execution times for the AVR

implementations were determined using the cycle-accurate instruction-set simulator of Microchip Studio 7 with the ATmega128 microcontroller as the target device, and the C implementation was compiled with AVR-GCC 5.4.0 at optimization level -O2.

Implementation results.

Table 17 summarizes the execution times and throughputs of the optimized C and assembly implementations of the APEX permutations on AVR and ARM, with the corresponding SPARKLE results included for comparison. The assembly implementations are optimized for a balance between code size and execution speed and retain their step loops instead of fully unrolling them.

On AVR, the hand-written APEX assembly implementations are approximately 2.3–2.4 times faster than their optimized C counterparts. On ARM, the corresponding factor is approximately 2.2–2.4. For the selected slim and big versions at each state width, the APEX assembly implementations require 12–20% fewer cycles than the corresponding SPARKLE implementations on AVR and 12–18% fewer cycles on ARM.

Since the step count remains parameterized, a single assembly routine for each state width supports both the slim and big versions. The code size and stack consumption of these implementations are reported in Table 18. Their total code sizes range from 348 to 628 bytes, and their stack consumption does not exceed 56 bytes. These resource requirements remain close to those of the corresponding SPARKLE implementations.

We also developed more aggressively speed-optimized ARM implementations in which the step loops are fully unrolled. As shown in Table 19, full unrolling reduces the execution time by approximately 11–20%, at the cost of a substantial increase in code size. For example, the six-step APEX₃₈₄ implementation is reduced from 808 to 670 cycles, while its code size increases from 492 to 2404 bytes.

Table 17 Performance of the APEX and SPARKLE permutations on an 8-bit AVR ATmega128 and a 32-bit ARM Cortex-M3 microcontroller. The results are given in cycles/byte, with the number in parentheses representing the total cycle count for one execution of the permutation. The SPARKLE results are those reported in [43].

Permutation	Steps	AVR		ARM	
		C	asm	C	asm
APEX ₂₅₆	6 (slim)	362 (11590)	154 (4936)	40 (1266)	17 (532)
	8 (big)	481 (15388)	205 (6554)	52 (1674)	22 (694)
SPARKLE ₂₅₆	7 (slim)	697 (22305)	179 (5728)	46 (1487)	19 (605)
	10 (big)	992 (31761)	254 (8146)	66 (2111)	26 (842)
APEX ₃₈₄	6 (slim)	353 (16966)	149 (7168)	39 (1854)	17 (808)
	9 (big)	528 (25351)	223 (10711)	58 (2760)	25 (1183)
SPARKLE ₃₈₄	7 (slim)	680 (32679)	173 (8318)	45 (2173)	19 (930)
	11 (big)	1066 (51215)	271 (13022)	71 (3397)	30 (1430)
APEX ₅₁₂	7 (slim)	407 (26033)	171 (10953)	44 (2842)	20 (1307)
	10 (big)	580 (37106)	244 (15612)	63 (4042)	29 (1847)
SPARKLE ₅₁₂	8 (slim)	768 (49169)	194 (12454)	51 (3263)	23 (1489)
	12 (big)	1150 (73633)	291 (18638)	76 (4879)	35 (2209)

Table 18 Code size and stack consumption of the loop-based APEX and SPARKLE permutation implementations on a 32-bit ARM Cortex-M3 microcontroller. The code size is given as the number of bytes occupied by the permutation in the `text` segment plus the bytes required for the round constants. The SPARKLE results are those reported in [43].

Permutation	Code Size (byte)	Stack Usage (byte)
APEX ₂₅₆	316 + 32 = 348	40
SPARKLE ₂₅₆	316 + 32 = 348	40
APEX ₃₈₄	456 + 36 = 492	56
SPARKLE ₃₈₄	452 + 32 = 484	48
APEX ₅₁₂	588 + 40 = 628	56
SPARKLE ₅₁₂	596 + 32 = 628	52

Table 19 Code size and performance of the fully unrolled APEX permutation implementations on a 32-bit ARM Cortex-M3 microcontroller.

Permutation	Steps	Code Size (byte)	Time (cycles)	Stack Usage (byte)
APEX ₂₅₆	6 (slim)	1512	431	40
	8 (big)	1996	556	40
APEX ₃₈₄	6 (slim)	2404	670	44
	9 (big)	3660	993	44
APEX ₅₁₂	7 (slim)	3976	1155	48
	10 (big)	5680	1637	48

Table 20 compares the fully unrolled ARMv7-M implementation of APEX₃₈₄ with those of the permutations used in several permutation-based candidates from the NIST lightweight cryptography standardization process. To account for the different rates of the corresponding AEAD instances, Time/Rate denotes the permutation execution time divided by the rate in bytes, expressed in cycles/byte, and serves as a normalized measure of throughput. A lower Time/Rate value indicates higher throughput. Under this normalization, APEX₃₈₄ achieves the lowest Time/Rate value of 20.94 cycles/byte, and hence the highest normalized throughput among the implementations listed in the table. It should be noted that the GIMLI AEAD instance targets a 256-bit security level, which should be taken into account when interpreting its comparatively lower normalized throughput.

Table 20 Comparison of fully unrolled ARMv7-M assembler implementations of ASCON, GIMLI, APEX₃₈₄, SPARKLE384, and XOODOO on a Cortex-M3 microcontroller. The ASCON, GIMLI, SPARKLE384, and XOODOO results are those reported in [43]. Here Time/Rate is the permutation time divided by the rate, in bytes, of the corresponding AEAD instance. The instances used for this normalization, together with their rates and claimed security levels, are ASCON-128a (16 bytes, 128 bits), GIMLI AEAD (16 bytes, 256 bits), APEX₃₈₄^(6,9)-AEAD-256/128 (32 bytes, 120 bits), SCHWAEMM256-128 (32 bytes, 120 bits), and XOODYAK (24 bytes, 128 bits), respectively.

Permutation	Code Size (byte)	Time (cycles)	Time/Rate (cycles/byte)
ASCON (8 rounds)	1810	499	31.19
GIMLI (24 rounds)	3950	1041	65.06
APEX ₃₈₄ (6 steps)	2404	670	20.94
SPARKLE384 (7 steps)	2820	781	24.40
XOODOO (12 rounds)	2376	657	27.38

Tables 21 and 22 report the benchmark results for our small-state hash functions and AEAD schemes on AVR and ARM. Replacing the C implementation of the APEX permutation with hand-optimized assembly reduces the cycle counts by approximately 55–57% on AVR and 50–55% on ARM. For short messages of 64 bytes, the optimized C implementations of APEX require 44–56% fewer cycles/byte than the corresponding SCHWAEMM and ESCH implementations on AVR, and 13–16% fewer on ARM. With hand-optimized assembly for the permutation, the corresponding reductions are 13–33% on AVR and 13–16% on ARM. For long messages of 1536 bytes, the optimized C implementations of APEX require 43–46% fewer cycles/byte on AVR and 13–15% fewer on ARM. With hand-optimized assembly for the permutation, the corresponding reductions are 12–16% on AVR and 11–17% on ARM.

Table 21 Benchmarking results for the four APEX AEAD and two APEX HASH instances, together with the corresponding SCHWAEMM and ESCH instances, on an 8-bit AVR ATmega128 microcontroller when processing messages of 64 and 1536 bytes. For AEAD, the benchmarked operation is encryption with no associated data. The SCHWAEMM and ESCH results are those reported in [43]. Results are given in cycles/byte, with the total cycle count in parentheses.

Instance	64 bytes of data		1536 bytes of data	
	Pure C	C + asm	Pure C	C + asm
APEX ₂₅₆ ^(6,8) -AEAD-128/128	1073 (68666)	480 (30749)	766 (1176070)	339 (521293)
SCHWAEMM128-128	2444 (156416)	712 (45583)	1421 (2182899)	387 (594898)
APEX ₃₈₄ ^(6,9) -AEAD-256/128	1113 (71215)	499 (31936)	580 (890705)	259 (398372)
SCHWAEMM256-128	2105 (134748)	596 (38166)	1071 (1644606)	302 (464347)
APEX ₃₈₄ ^(6,9) -AEAD-192/192	1379 (88260)	611 (39132)	756 (1161438)	333 (511521)
SCHWAEMM192-192	2594 (165994)	727 (46526)	1399 (2148858)	395 (606716)
APEX ₅₁₂ ^(7,10) -AEAD-256/256	1626 (104048)	715 (45755)	872 (1338918)	380 (584231)
SCHWAEMM256-256	3014 (192918)	839 (53704)	1574 (2417064)	434 (666554)
APEX ₃₈₄ ^(6,9) -HASH-256	1498 (95847)	652 (41736)	1102 (1692875)	477 (732656)
ESCH256	2714 (173678)	893 (57187)	1978 (3038834)	559 (860071)
APEX ₅₁₂ ^(7,10) -HASH-384	2664 (170465)	1144 (73193)	1697 (2606073)	727 (1116013)
ESCH384	4732 (302837)	1308 (83725)	2992 (4595649)	830 (161717)

Table 22 Benchmarking results for the four APEX AEAD and two APEX HASH instances, together with the corresponding SCHWAEMM and ESCH instances, on a 32-bit ARM Cortex-M3 microcontroller when processing messages of 64 and 1536 bytes. For AEAD, the benchmarked operation is encryption with no associated data. The SCHWAEMM and ESCH results are those reported in [43]. Results are given in cycles/byte, with the total cycle count in parentheses.

Instance	64 bytes of data		1536 bytes of data	
	Pure C	C + asm	Pure C	C + asm
APEX ₂₅₆ ^(6,8) -AEAD-128/128	125 (7992)	60 (3835)	87 (134308)	41 (62715)
SCHWAEMM128-128	148 (9491)	69 (4384)	101 (155495)	46 (70440)
APEX ₃₈₄ ^(6,9) -AEAD-256/128	129 (8224)	63 (4027)	67 (102386)	33 (50119)
SCHWAEMM256-128	154 (9851)	74 (4715)	77 (118917)	37 (57109)
APEX ₃₈₄ ^(6,9) -AEAD-192/192	158 (10140)	77 (4898)	86 (132269)	41 (63282)
SCHWAEMM192-192	189 (12066)	89 (5698)	100 (153597)	47 (72077)
APEX ₅₁₂ ^(7,10) -AEAD-256/256	185 (11859)	93 (5937)	98 (151285)	49 (74799)
SCHWAEMM256-256	219 (14029)	111 (7072)	113 (173051)	56 (86284)
APEX ₃₈₄ ^(6,9) -HASH-256	167 (10719)	78 (4963)	122 (186899)	55 (85003)
ESCH256	198 (12654)	90 (5774)	144 (221678)	66 (101454)
APEX ₅₁₂ ^(7,10) -HASH-384	296 (18946)	142 (9082)	187 (287494)	89 (136502)
ESCH384	341 (21847)	165 (10561)	216 (332623)	105 (161717)

Table 23 compares $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ with several hash functions producing 256-bit digests on an 8-bit AVR ATmega128 microcontroller. All throughput values are obtained by hashing a 500-byte message. The comparison includes SHA-256, KECCAK, three other SHA-3 finalists (BLAKE-256, GROESTL-256, and JH-256), and two GIMLI-HASH implementations. The results show that $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ achieves the best throughput among the listed implementations, requiring 504 cycles/byte. This is 5.3% fewer cycles/byte than SHA-256 and 10.3% fewer than BLAKE-256. In both $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ and ESCH256, only the underlying permutation is hand-optimized in assembler. All other listed hash implementations are written entirely in assembler.

Table 23 Comparison of $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ with other hash functions producing a 256-bit digest on an 8-bit AVR ATmega128 microcontroller. Throughput is measured by hashing a 500-byte message. The implementations of $\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$ and ESCH256 use hand-optimized assembler only for the permutation, whereas the implementations of all other hash functions are written entirely in assembler.

Hash function	Ref.	Throughput (cycles/byte)	Code size (byte)
$\text{APEX}_{384}^{(6,9)}\text{-HASH-256}$	This work	504	1662
ESCH256	[43]	578	1868
BLAKE-256	[63]	562	1166
GIMLI-HASH small	[64]	1610	778*
GIMLI-HASH fast	[64]	725	19218*
GROESTL-256	[63]	686	1400
JH-256	[63]	5062	1020
KECCAK [†]	[63]	1432	868
SHA-256	[63]	532	1090

*The code size corresponds to the permutation alone. [†]The reported KECCAK instance uses $r = 1088$ and $c = 512$.

6 Large-State Hash Functions for the Next-Generation Commercial Cryptographic Algorithms Program

On 5 February 2025, China’s Institute of Commercial Cryptography Standards (ICCS) launched the Next-generation Commercial Cryptographic Algorithms Program (NGCC), a global call for proposals covering public-key algorithms, cryptographic hash algorithms, and block ciphers [38]. As part of this program, the Submission Requirements for Cryptographic Hash Algorithms require a proposal to support both 512- and 1024-bit digests, permit an optional 768-bit digest, and support a maximum message length of at least $2^{64} - 1$ bits [65]. For a d -bit digest, the corresponding evaluation criteria set the classical security targets at $d/2$ bits against collision attacks and d bits against preimage and second-preimage attacks, and require the complexity of quantum attacks to be no lower than that of generic quantum attacks [66].

To address these requirements, we propose three large-state hash functions based on APEX_{1536} using the Sponge with Feed-Forward (Sponge-F) construction [37], a feed-forward variant of the sponge mode: $\text{APEX}_{1536}^{12}\text{-HASH-F-512}$, $\text{APEX}_{1536}^{16}\text{-HASH-F-768}$, and $\text{APEX}_{1536}^{20}\text{-HASH-F-1024}$. The 512- and 1024-bit members provide the two required digest lengths, while the 768-bit member provides the optional intermediate profile.

6.1 Specification

The hash functions $\text{APEX}_{1536}^{12}\text{-HASH-F-512}$, $\text{APEX}_{1536}^{16}\text{-HASH-F-768}$, and $\text{APEX}_{1536}^{20}\text{-HASH-F-1024}$ accept arbitrary-length bit strings $M \in \mathbb{F}_2^*$ satisfying $0 \leq |M| < 2^{64}$, and therefore support messages up to $2^{64} - 1$ bits. All three hash functions use the 1536-bit permutation $\text{APEX}_{1536}^{n_s}$, and hence have state width $b = 1536$. The step count n_s is selected separately for each instance according to its security target. Let d denote the digest length. We set the capacity to $c = d + 64$ and obtain the rate as $r = b - c$. The state is partitioned into an r -bit rate and a c -bit capacity, with the digest extracted from the rightmost d bits of the capacity. Table 24 summarizes the parameters and security claims of the three large-state hash functions.

Table 24 Parameters and security claims of the three HASH-F instances. Sizes and security claims are in bits unless stated otherwise; each security claim is given as classical (quantum).

Instance	Parameters				Security claim classical (quantum)			Data limit (bits)
	b	r	c	d	Collision	Preimage	2nd preimage	
$\text{APEX}_{1536}^{12}\text{-HASH-F-512}$	1536	960	576	512	256 (128)	512 (256)	512 (256)	$2^{64} - 1$
$\text{APEX}_{1536}^{16}\text{-HASH-F-768}$	1536	704	832	768	384 (192)	768 (384)	768 (384)	$2^{64} - 1$
$\text{APEX}_{1536}^{20}\text{-HASH-F-1024}$	1536	448	1088	1024	512 (256)	1024 (512)	1024 (512)	$2^{64} - 1$

The three hash functions share the following specification. The state S is initialized to $0^r \parallel IV_d$, where $IV_d \in \mathbb{F}_2^c$ is a fixed initial value for the corresponding digest length d , with the concrete values given in Table 25. The message is padded using the simple 10^* rule, denoted by pad_r , and specified in Algorithm 4, and is then divided into r -bit blocks. Padding is always applied. In particular, the empty message produces one padded block, while a message whose length is a multiple of r receives an additional block $1 \parallel 0^{r-1}$.

Message blocks are XORed directly into the rate part of the state. When the final padded block is injected, the fixed nonzero final-block constant $\theta = 0^{c-1} \parallel 1$ is simultaneously XORed into the capacity. The corresponding APEX permutation is then applied. After every permutation call, the c -bit capacity part of its input is XORed into the capacity part of its output. This XOR constitutes the feed-forward operation of **Sponge-F**. After the permutation and feed-forward operation for the final padded message block have been completed, the digest is $D = \text{trunc}_d^R(S)$, where trunc_d^R returns the rightmost d bits of S .

The hash functions $\text{APEX}_{1536}^{12}\text{-HASH-F-512}$, $\text{APEX}_{1536}^{16}\text{-HASH-F-768}$, and $\text{APEX}_{1536}^{20}\text{-HASH-F-1024}$ are formally specified in Algorithms 17, 18, and 19, and illustrated in Figures 11, 12, and 13, respectively.

Table 25 Fixed initial values IV_d of the three HASH-F instances $\text{APEX}_{1536}^{n_s}\text{-HASH-F-}d$ in hexadecimal.

Initial value	Hexadecimal
IV_{512}	06CEA2A79B31F9C7E7E6C9800913C441EE4BB701620315A4E4C050054322810825FB65F11ADB51FF A7EDE851DF905C9B4AEDD3C6B24F328966AF258F7DD6F580E210EAC58D64BEBA
IV_{768}	A9731B04F86C0BCD98B1F567C76E868696015C0313A6FE39FFDC1C45A426EC4E7683785BCBE39F9B CFA39AE9B0990F2C0EB52EACD58B9EBA96B4CA794F3C92C56DED1B5B96BD96B596D738536BF4748C FB89EA58EED20E4CF809D0767A15CDC864E35B5B969EA8B2
IV_{1024}	7DD933F45A8DA35DE7C92E02B490799F39686F9AC800D7CEE9A9933EAFD4AE0F8C98A77CFD4071A3 2E4AD6C9FA4867DA87DAAC5F4E17E29C16FC1DED86C82A71E7442ADDDFF93EE3467600669824BB389 CF5010CACE5C34AB242E815277EEAF7993AA1B488D4941509EC081FCC93E43D984655786F65C908B 9CED3ECA8D6352C65F933A1B4D17FEFE

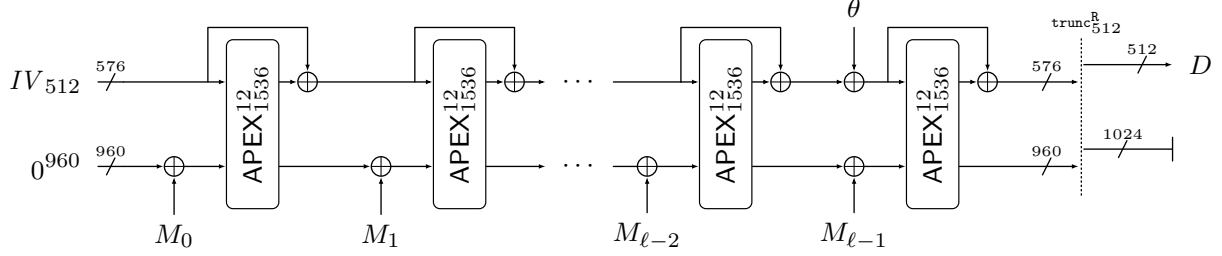


Fig. 11 The hash function $\text{APEX}_{1536}^{12}\text{-HASH-F-512}$ with rate $r = 960$ and capacity $c = 576$. Each message block is XORed into the rate, and the capacity part of each permutation input is fed forward into its output. For the final padded block, $\theta = 0^{575} \parallel 1 \in \mathbb{F}_2^{576}$ is injected into the capacity. The rightmost 512 bits of the final capacity form the digest $D \in \mathbb{F}_2^{512}$.

Algorithm 17 $\text{APEX}_{1536}^{12}\text{-HASH-F-512}$.

Input/Output: $M \in \mathbb{F}_2^*$ with $0 \leq |M| < 2^{64}$ / $D \in \mathbb{F}_2^{512}$

▷ Message parsing and padding

$\ell \leftarrow \left\lfloor \frac{|M|}{960} \right\rfloor + 1$

if $(M \neq \epsilon)$ and $(|M| \bmod 960 \neq 0)$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M$

where $|M_i| = 960$ ($0 \leq i < \ell - 1$) and $1 \leq |M_{\ell-1}| < 960$

$M_{\ell-1} \leftarrow \text{pad}_{960}(M_{\ell-1})$

else if $(M \neq \epsilon)$ and $(|M| \bmod 960 = 0)$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M \parallel \text{pad}_{960}(\epsilon)$

where $|M_i| = 960$ ($0 \leq i < \ell$)

else

$M_0 \leftarrow \text{pad}_{960}(\epsilon)$

end if

▷ State initialization

$S \leftarrow 0^{960} \parallel IV_{512} \in \mathbb{F}_2^{1536}$

▷ Absorption and feed-forward

for all $j \in [0, \ell - 2]$ **do**

$M'_j \leftarrow S \oplus (M_j \parallel 0^{576})$

$S \leftarrow \text{APEX}_{1536}^{12}(M'_j) \oplus (0^{960} \parallel \text{trunc}_{576}^R(M'_j))$

end for

$\theta \leftarrow 0^{575} \parallel 1 \in \mathbb{F}_2^{576}$

$M'_{\ell-1} \leftarrow S \oplus (M_{\ell-1} \parallel \theta)$

$S \leftarrow \text{APEX}_{1536}^{12}(M'_{\ell-1}) \oplus (0^{960} \parallel \text{trunc}_{576}^R(M'_{\ell-1}))$

▷ Digest extraction

$D \leftarrow \text{trunc}_{512}^R(S)$

return D

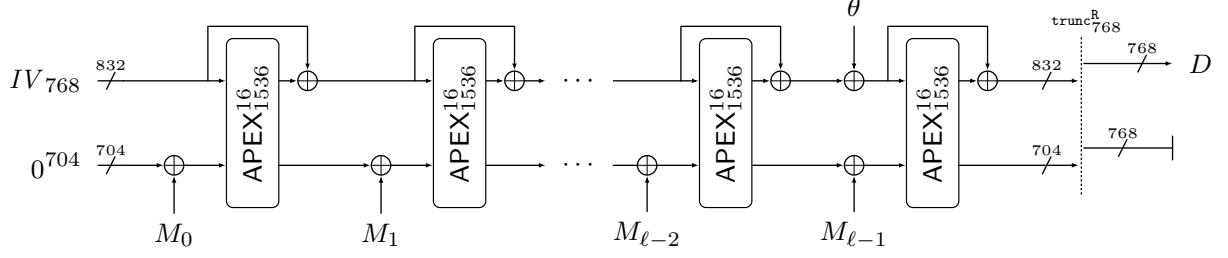


Fig. 12 The hash function $\text{APEX}_{1536}^{16}\text{-HASH-F-768}$ with rate $r = 704$ and capacity $c = 832$. Each message block is XORed into the rate, and the capacity part of each permutation input is fed forward into its output. For the final padded block, $\theta = 0^{831} \parallel 1 \in \mathbb{F}_2^{832}$ is injected into the capacity. The rightmost 768 bits of the final capacity form the digest $D \in \mathbb{F}_2^{768}$.

Algorithm 18 $\text{APEX}_{1536}^{16}\text{-HASH-F-768}$.

Input/Output: $M \in \mathbb{F}_2^*$ with $0 \leq |M| < 2^{64}$ / $D \in \mathbb{F}_2^{768}$

$\ell \leftarrow \left\lfloor \frac{|M|}{704} \right\rfloor + 1$

if $(M \neq \epsilon)$ and $(|M| \bmod 704 \neq 0)$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M$

where $|M_i| = 704$ ($0 \leq i < \ell - 1$) and $1 \leq |M_{\ell-1}| < 704$

$M_{\ell-1} \leftarrow \text{pad}_{704}(M_{\ell-1})$

else if $(M \neq \epsilon)$ and $(|M| \bmod 704 = 0)$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M \parallel \text{pad}_{704}(\epsilon)$

where $|M_i| = 704$ ($0 \leq i < \ell$)

else

$M_0 \leftarrow \text{pad}_{704}(\epsilon)$

end if

▷ Message parsing and padding

$S \leftarrow 0^{704} \parallel IV_{768} \in \mathbb{F}_2^{1536}$

▷ State initialization

for all $j \in [0, \ell - 2]$ **do**

$M'_j \leftarrow S \oplus (M_j \parallel 0^{832})$

$S \leftarrow \text{APEX}_{1536}^{16}(M'_j) \oplus (0^{704} \parallel \text{trunc}_{832}^R(M'_j))$

end for

▷ Absorption and feed-forward

$\theta \leftarrow 0^{831} \parallel 1 \in \mathbb{F}_2^{832}$

$M'_{\ell-1} \leftarrow S \oplus (M_{\ell-1} \parallel \theta)$

$S \leftarrow \text{APEX}_{1536}^{16}(M'_{\ell-1}) \oplus (0^{704} \parallel \text{trunc}_{832}^R(M'_{\ell-1}))$

▷ Digest extraction

$D \leftarrow \text{trunc}_{768}^R(S)$

return D

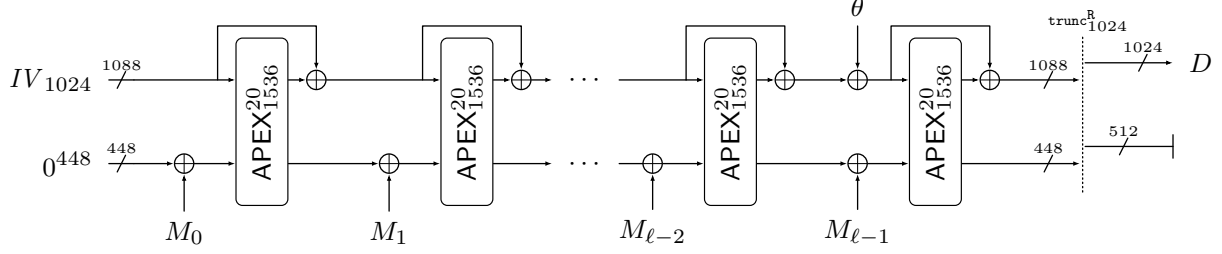


Fig. 13 The hash function $\text{APEX}_{1536}^{20}\text{-HASH-F-1024}$ with rate $r = 448$ and capacity $c = 1088$. Each message block is XORed into the rate, and the capacity part of each permutation input is fed forward into its output. For the final padded block, $\theta = 0^{1087} \parallel 1 \in \mathbb{F}_2^{1088}$ is injected into the capacity. The rightmost 1024 bits of the final capacity form the digest $D \in \mathbb{F}_2^{1024}$.

Algorithm 19 $\text{APEX}_{1536}^{20}\text{-HASH-F-1024}$.

Input/Output: $M \in \mathbb{F}_2^*$ with $0 \leq |M| < 2^{64}$ / $D \in \mathbb{F}_2^{1024}$

$\ell \leftarrow \left\lfloor \frac{|M|}{448} \right\rfloor + 1$

▷ Message parsing and padding

if $(M \neq \epsilon)$ and $(|M| \bmod 448 \neq 0)$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M$

where $|M_i| = 448$ ($0 \leq i < \ell - 1$) and $1 \leq |M_{\ell-1}| < 448$

$M_{\ell-1} \leftarrow \text{pad}_{448}(M_{\ell-1})$

else if $(M \neq \epsilon)$ and $(|M| \bmod 448 = 0)$ **then**

$M_0 \parallel M_1 \parallel \dots \parallel M_{\ell-1} \leftarrow M \parallel \text{pad}_{448}(\epsilon)$

where $|M_i| = 448$ ($0 \leq i < \ell$)

else

$M_0 \leftarrow \text{pad}_{448}(\epsilon)$

end if

▷ State initialization

$S \leftarrow 0^{448} \parallel IV_{1024} \in \mathbb{F}_2^{1536}$

▷ Absorption and feed-forward

for all $j \in [0, \ell - 2]$ **do**

$M'_j \leftarrow S \oplus (M_j \parallel 0^{1088})$

$S \leftarrow \text{APEX}_{1536}^{20}(M'_j) \oplus (0^{448} \parallel \text{trunc}_{1088}^R(M'_j))$

end for

$\theta \leftarrow 0^{1087} \parallel 1 \in \mathbb{F}_2^{1088}$

$M'_{\ell-1} \leftarrow S \oplus (M_{\ell-1} \parallel \theta)$

$S \leftarrow \text{APEX}_{1536}^{20}(M'_{\ell-1}) \oplus (0^{448} \parallel \text{trunc}_{1088}^R(M'_{\ell-1}))$

▷ Digest extraction

$D \leftarrow \text{trunc}_{1024}^R(S)$

return D

6.2 Design Rationale and Security Analysis

6.2.1 Choice of the Sponge-F Mode

The NGCC evaluation criteria require the prescribed classical security levels while also taking resistance to generic quantum attacks into account. For a conventional sponge with a d -bit digest, a c -bit capacity, and squeezing rate r' , the generic collision, preimage, and second-preimage security bounds are $\min\{d/2, c/2\}$, $\min\{d, \max\{d - r', c/2\}\}$, and $\min\{d, c/2\}$ bits, respectively [37, 53]. In particular, meeting the required d -bit second-preimage security already forces $c \geq 2d$. With the state width fixed to $b = 1536$, this leaves a rate of at most 512 bits for $d = 512$, no positive rate for $d = 768$, and is impossible for $d = 1024$. A conventional sponge therefore cannot efficiently realize all three target profiles using APEX_{1536} .

We instead employ Sponge with Feed-Forward (**Sponge-F**), a variant of the sponge construction introduced in [37]. After each permutation call, **Sponge-F** XORs the capacity part of the permutation input into the corresponding part of its output. This feed-forward operation mitigates the invertibility-related limitation of the conventional sponge

and allows all three target profiles to retain a positive rate while meeting the required classical and generic quantum security levels.

6.2.2 Parameter Selection

The NGCC requirements prescribe digest lengths $d \in \{512, 768, 1024\}$ together with the corresponding classical and quantum security levels. To keep the three instances structurally uniform and retain a comparatively high absorption rate under these requirements, they all use the 1536-bit permutation $\text{APEX}_{1536}^{n_s}$, so that $b = 1536$, with an instance-dependent step count n_s .

For **Sponge-F** with digest length d , capacity c , and ℓ padded message blocks, the classical collision, preimage, and second-preimage bounds are $d/2$, d , and $\min\{d, c - \log_2 \ell\}$ bits, respectively. The corresponding generic quantum bounds are $d/4$, $d/2$, and $\min\{d/2, (c - \log_2 \ell)/2\}$ bits [37]. Since padding is always applied, a message M is processed in $\ell = \lfloor |M|/r \rfloor + 1$ blocks. The input restriction $|M| < 2^{64}$ gives $\log_2 \ell \leq 64$. We therefore choose $c = d + 64$, which guarantees $c - \log_2 \ell \geq d$, and set $r = b - c = 1536 - d - 64$.

This gives $(r, c) = (960, 576)$, $(704, 832)$, and $(448, 1088)$ for $\text{APEX}_{1536}^{12}\text{-HASH-F-512}$, $\text{APEX}_{1536}^{16}\text{-HASH-F-768}$, and $\text{APEX}_{1536}^{20}\text{-HASH-F-1024}$, respectively. Their classical collision, preimage, and second-preimage claims are $(256, 512, 512)$, $(384, 768, 768)$, and $(512, 1024, 1024)$ bits. The corresponding quantum claims are $(128, 256, 256)$, $(192, 384, 384)$, and $(256, 512, 512)$ bits, as summarized in Table 24.

Each fixed initial value IV_d is derived deterministically from the ASCII seed

$$\text{seed}_d = \text{APEX-HASH-SPONGE-F} \mid \mathbf{v} = 1 \mid \mathbf{d} = \langle d \rangle \mid \mathbf{b} = \langle b \rangle \mid \mathbf{r} = \langle r \rangle \mid \mathbf{c} = \langle c \rangle \mid \mathbf{ns} = \langle n_s \rangle,$$

where each quantity in angle brackets is replaced by its decimal ASCII representation. To generate IV_d , we use the cryptographic hash function **SM3** [67] in a counter-based expansion. For $i = 1, \dots, m_d$, compute

$$Z_i = \text{SM3}(\text{seed}_d \parallel \text{BE32}(i)),$$

where $\text{BE32}(i)$ is the 32-bit big-endian encoding of the integer i . **SM3** accepts bit strings shorter than 2^{64} bits, uses a 512-bit message-block size, and returns a 256-bit digest. Since each **SM3** invocation contributes 256 bits, we set $m_d = \lceil c/256 \rceil$. The three profiles with digest lengths $d = 512, 768$, and 1024 have capacities $c = 576, 832$, and 1088 , and therefore use $m_d = 3, 4$, and 5 **SM3** invocations, respectively. The strings $Z_1, \dots, Z_{m_d}$ are concatenated in this order. The leftmost c bits are retained and their rightmost bit is set to zero to obtain IV_d . This procedure makes the initial values reproducible and binds each one to the complete parameter set (d, b, r, c, n_s) .

Following the multi-instance guidance for **Sponge-F** in [37], we choose the public final-block constant as $\theta = 0^{c-1} \parallel 1$. Distinct initial values of a common capacity width must satisfy $IV_i \neq IV_j \oplus \theta$, or equivalently $IV_i \oplus IV_j \neq \theta$. Otherwise, the same permutation input could be interpreted as an initialization call in one domain and a final-block call in another. Our rightmost-bit convention enforces this separation structurally: every IV_d ends in 0, so the XOR of any two same-width initial values ends in 0, whereas θ ends in 1. Although the three present profiles use different capacity widths, the convention also preserves the required separation if further domains or profiles of an existing width are introduced.

6.2.3 Step-Count Selection and Security Analysis

The collision-security analysis of **Sponge-F** reduces a collision to correlation intractability (CI) of the underlying permutation with respect to the relation $\mathcal{R}_{\text{coll}}$. In differential terms, this relation requires the input and output differences in the rightmost c capacity bits to be equal. We therefore study the differential probabilities of $\text{APEX}_{1536}^{n_s}$ under this endpoint constraint. For completeness, we also investigate the analogous linear setting, in which the input and output masks in the rightmost c bits are constrained to be equal, although this additional analysis is not required by the collision-CI argument. With minor modifications, the long-trail search techniques of [22, 30, 31, 36] can be applied to these restricted settings by enforcing the corresponding equality constraint at the two endpoints. The resulting differential and linear weights are reported in Tables 26 and 27, respectively.

The restricted trails form subsets of the unrestricted full-state trails considered in Tables 3 and 4. Consequently, the unrestricted bounds remain valid here, but can be loose: they may overestimate the adversary's capabilities and thereby underestimate the security of **APEX** under the **Sponge-F** interface. The restricted searches therefore provide a more precise analysis for the **APEX-HASH-F** instances.

In addition, we performed a restricted search for effective branch-level truncated differentials, constraining the input and output activity patterns in the rightmost $c/64$ capacity branches to be equal. For both $c = 512$ and $c = 576$, the maximum probabilities among effective trails are at most 2^{-576} over eight steps and 2^{-832} over nine steps.

Table 26 Differential long-trail weights for $\text{APEX}_{1536}^{n_s}$. The $b \rightarrow b$ row reproduces the unrestricted full-state differential bounds from Table 3. In every other row, the input and output differences in the rightmost c bits are constrained to be equal. An entry $w_0/w_{\neq 0}$ gives $-\log_2 p$ when this common difference is 0^c and nonzero, respectively, where p is an upper bound on the differential-trail probability. The symbol $\emptyset$ means that no such trail exists in the model.

$c \backslash n_s$	1	2	3	4	5	6	7	8	9	10	11	12	13
$b \rightarrow b$	10	37	152	192	283	376	435	511	582	622	697	756	830
512	10/20	37/94	152/197	192/268	317/327	376/386	450/454	511/511	601/604	641/663	732/739	825/790	901/849
576	10/20	37/94	152/197	261/268	317/327	376/386	450/454	511/511	601/604	641/663	732/739	825/790	901/849
768	$\emptyset/20$	$\emptyset/94$	222/224	261/268	337/337	428/396	487/454	541/511	617/604	691/663	767/739	825/808	901/867
832	$\emptyset/20$	$\emptyset/94$	222/222	261/261	337/337	428/396	487/454	541/511	617/604	691/663	767/739	825/808	901/867

Table 27 Linear long-trail weights for $\text{APEX}_{1536}^{n_s}$. The $b \rightarrow b$ row reproduces the unrestricted full-state linear bounds from Table 4. In every other row, the input and output linear masks in the rightmost c bits are constrained to be equal. An entry $w_0/w_{\neq 0}$ gives $-\log_2 |C_{\text{trail}}|$ when this common mask is 0^c and nonzero, respectively, where $|C_{\text{trail}}|$ is an upper bound on the absolute linear-trail correlation. The symbol $\emptyset$ means that no such trail exists in the model.

$c \backslash n_s$	1	2	3	4	5	6	7	8	9	10	11	12	13
$b \rightarrow b$	6	20	84	108	156	206	240	282	318	342	382	416	456
512	6/12	40/52	84/102	108/144	156/162	206/212	244/244	282/282	318/330	350/366	382/396	434/434	476/462
576	6/12	40/52	84/102	124/144	156/162	212/212	244/244	284/282	318/330	350/372	382/396	434/440	476/462
768	$\emptyset/12$	$\emptyset/72$	108/102	130/150	180/162	222/212	256/244	298/288	340/330	364/372	406/396	438/440	480/462
832	$\emptyset/12$	$\emptyset/72$	126/102	130/130	180/162	222/212	256/244	298/288	340/330	364/372	406/396	438/440	480/462

For each $\text{APEX}_{1536}^{n_s}$ -HASH-F- d instance, we adopt a conservative security target of 2^c , where $c = d + 64$. A differential trail with probability p has an estimated data complexity of approximately p^{-1} , whereas a linear trail with absolute correlation C has an estimated data complexity of approximately C^{-2} . We therefore require $W_D = -\log_2 p \geq c$ and $W_L = -\log_2 |C| \geq c/2$.

For $d = 512$, we have $c = 576$. Table 26 shows that the restricted differential weight first exceeds 576 after nine steps, while Table 27 shows that the restricted linear weight first exceeds 288 after nine steps. Moreover, the restricted branch-level truncated-differential analysis bounds the maximum probability among effective nine-step trails by 2^{-832} . We therefore obtain a minimum of nine steps. The longest distinguishers found in the other analyses cover no more than nine complete steps (see Section 3.3.3).

For $d = 768$, we have $c = 832$. The restricted differential weight first exceeds 832 after thirteen steps, whereas the restricted linear weight first exceeds 416 after twelve steps. We therefore obtain a minimum of thirteen steps. The longest distinguishers found in the other analyses cover no more than thirteen complete steps (see Section 3.3.3).

For $d = 1024$, we have $c = 1088$. We conservatively use the unrestricted full-state bounds in Tables 3 and 4. The differential weight first exceeds 1088 after seventeen steps, while the linear weight first exceeds 544 after sixteen steps. The resulting minimum is therefore seventeen steps. The longest distinguishers found in the other analyses cover no more than sixteen complete steps (see Section 3.3.3).

The minimum step counts satisfying our trail-complexity targets are thus 9, 13, and 17 for digest lengths 512, 768, and 1024, respectively. Since APEX_{1536} reaches full diffusion and fulfills the SAC after three steps (Section 3.3.3.6), we add three complete steps as an additional security margin. We finally select $n_s = 12, 16$, and 20 for APEX_{1536}^{12} -HASH-F-512, APEX_{1536}^{16} -HASH-F-768, and APEX_{1536}^{20} -HASH-F-1024, respectively.

6.3 Implementation

We provide implementations of the three HASH-F instances for x86-64 and 32-bit ARMv7-M platforms. For x86-64, these comprise a reference scalar C implementation, an optimized scalar C implementation, and vectorized C implementations using AVX2 and AVX-512 intrinsics. For ARMv7-M, we provide hand-optimized assembly implementations.

Implementation environment. The x86-64 benchmarks were performed under Linux on an Intel Xeon Platinum 8582C processor. The benchmark programs were compiled with GCC 15.3.0 using optimization level `-O3`. All execution times for the ARMv7-M implementations were determined using the cycle-accurate instruction-set simulator of Keil μ Vision v5.24.2.0, with a generic Cortex-M3 model as the target device. For comparison, we use SHA3-512 implementations from the eXtended Keccak Code Package (XKCP), a free and open-source collection of implementations of cryptographic schemes defined by the Keccak team, and from the OpenSSL 3.5.7 cryptographic library.

Implementation results. Table 28 reports the performance of the three HASH-F instances and SHA3-512 on x86-64 and ARMv7-M. Results for both platforms are given in CPU cycles/byte, and lower values indicate better performance. The label scalar C denotes a C implementation for which automatic vectorization is disabled, so only scalar instructions are used.

The HASH-F implementations achieve competitive throughput on both platforms. For the 512-bit digest case, APEX_{1536}^{12} -HASH-F-512 provides higher throughput than SHA3-512, especially for long messages. For a 1 MiB message, the fastest APEX_{1536}^{12} -HASH-F-512 implementation requires 3.33 cycles/byte on x86-64, compared with 6.95 cycles/byte for the fastest listed SHA3-512 implementation, corresponding to approximately $2.09\times$ higher throughput. On ARMv7-M, the corresponding values are 69.63 and 200.51 cycles/byte, respectively, giving approximately $2.88\times$ higher throughput.

Table 28 Performance of the three HASH-F instances and SHA3-512 on x86-64 and ARMv7-M. Results are given in CPU cycles/byte, and lower values indicate better performance.

Platform	Algorithm	Implementation	64 B	192 B	1 KiB	1536 B	64 KiB	1 MiB
x86-64	APEX_{1536}^{12} -HASH-F-512	Reference scalar C	25.81	16.79	13.90	13.33	13.12	13.11
		Optimized scalar C	13.69	9.06	7.61	7.33	7.22	7.22
		AVX2 intrinsic C	11.07	7.23	5.99	5.76	5.66	5.66
		AVX-512 intrinsic C	7.34	4.63	3.60	3.43	3.34	3.33
	APEX_{1536}^{16} -HASH-F-768	Reference scalar C	33.55	32.75	24.19	24.18	23.47	23.40
		Optimized scalar C	18.60	18.25	13.61	13.60	13.18	13.17
		AVX2 intrinsic C	14.51	14.16	10.51	10.50	10.18	10.18
		AVX-512 intrinsic C	9.42	8.71	6.29	6.26	6.02	6.02
	APEX_{1536}^{20} -HASH-F-1024	Reference scalar C	82.74	54.15	47.95	47.15	46.19	46.24
		Optimized scalar C	46.32	30.43	26.77	26.27	25.69	25.68
		AVX2 intrinsic C	35.32	23.41	20.75	20.38	19.97	19.98
		AVX-512 intrinsic C	21.97	14.22	12.36	12.12	11.83	11.82
	SHA3-512	XKCP reference scalar C	55.23	51.64	46.95	45.79	44.29	44.20
		XKCP optimized scalar C	8.38	12.03	8.03	7.62	6.95	6.95
		XKCP AVX2 asm	12.17	11.99	11.17	10.92	10.58	10.57
		XKCP AVX-512 asm	8.57	8.43	7.80	7.62	7.38	7.38
		OpenSSL asm	10.90	9.90	8.94	8.71	8.42	8.45
ARMv7-M	APEX_{1536}^{12} -HASH-F-512	ARMv7-M asm	143.41	91.46	74.23	71.33	69.75	69.63
	APEX_{1536}^{16} -HASH-F-768	ARMv7-M asm	189.84	175.67	128.78	128.34	123.78	123.72
	APEX_{1536}^{20} -HASH-F-1024	ARMv7-M asm	442.39	287.58	250.46	245.62	239.79	239.63
	SHA3-512	XKCP ARMv7-M asm	235.20	228.22	211.76	207.00	200.68	200.51

7 Block Ciphers and Tweakable Block Ciphers

In this section, we show that the APEX permutation family can also be readily used to construct block ciphers and tweakable block ciphers. Depending on the design requirements, the APEX step function can be combined with a suitable key schedule, an appropriate round-constant injection scheme, and, for a tweakable construction, a suitable tweak schedule. We present three representative instantiations based on APEX_{64}^8 and APEX_{256}^{15} : the ultralightweight small-state block cipher $\text{APEX}_{64}^8\text{-BC-128}$, the large-state block cipher $\text{APEX}_{256}^{15}\text{-BC-256}$, and the large-state tweakable block cipher $\text{APEX}_{256}^{15}\text{-TBC-256/128}$. These examples are not exhaustive. The other supported APEX state widths can be instantiated in the same manner to obtain additional block-cipher and tweakable-block-cipher profiles.

7.1 Specification

Table 29 summarizes the parameters of our proposed block cipher (BC) and tweakable block cipher (TBC) instances.

Table 29 Parameters of the three APEX-BC and APEX-TBC instances. All lengths are in bits.

Instance	Block length	Key length	Tweak length
$\text{APEX}_{64}^8\text{-BC-128}$	64	128	–
$\text{APEX}_{256}^{15}\text{-BC-256}$	256	256	–
$\text{APEX}_{256}^{15}\text{-TBC-256/128}$	256	256	128

7.1.1 The Block Cipher $\text{APEX}_{64}^8\text{-BC-128}$

The ultralightweight block cipher $\text{APEX}_{64}^8\text{-BC-128}$ operates on 64-bit blocks under a 128-bit key and consists of eight steps. It cyclically uses the five round constants $c_0, \dots, c_4$ (see Algorithm 3) and does not require a key schedule. The encryption and decryption procedures are formally specified in Algorithms 20 and 21, respectively.

Algorithm 20 Encryption for $\text{APEX}_{64}^8\text{-BC-128}$.
Input/Output: $P \in \mathbb{F}_2^{64}$, $K \in \mathbb{F}_2^{128}$ / $C \in \mathbb{F}_2^{64}$

```

 $(x_0, y_0) \in (\mathbb{F}_2^{32})^2 \leftarrow P$ 
 $(k_0, k_1, k_2, k_3) \in (\mathbb{F}_2^{32})^4 \leftarrow K$ 
 $(K_0, K_1) \leftarrow ((k_0, k_1), (k_2, k_3))$ 
for  $s \in [0, 7]$  do
   $Z_s \leftarrow (x_s, y_s) \oplus (s, 0) \oplus K_{s \bmod 2}$ 
   $(x_{s+1}, y_{s+1}) \leftarrow \mathcal{A}_{c_s \bmod 5}(Z_s)$ 
end for
 $C \leftarrow (x_8, y_8) \oplus K_0$ 
return  $C$ 

```

Algorithm 21 Decryption for $\text{APEX}_{64}^8\text{-BC-128}$.
Input/Output: $C \in \mathbb{F}_2^{64}$, $K \in \mathbb{F}_2^{128}$ / $P \in \mathbb{F}_2^{64}$

```

 $(k_0, k_1, k_2, k_3) \in (\mathbb{F}_2^{32})^4 \leftarrow K$ 
 $(K_0, K_1) \leftarrow ((k_0, k_1), (k_2, k_3))$ 
 $(x_8, y_8) \in (\mathbb{F}_2^{32})^2 \leftarrow C \oplus K_0$ 
for  $s = 7, \dots, 0$  do
   $Z_s \leftarrow \mathcal{A}_{c_s \bmod 5}^{-1}(x_{s+1}, y_{s+1})$ 
   $(x_s, y_s) \leftarrow Z_s \oplus (s, 0) \oplus K_{s \bmod 2}$ 
end for
 $P \leftarrow (x_0, y_0)$ 
return  $P$ 

```

7.1.2 The Block Cipher $\text{APEX}_{256}^{15}\text{-BC-256}$

The large-state block cipher $\text{APEX}_{256}^{15}\text{-BC-256}$ operates on 256-bit blocks under a 256-bit key and consists of fifteen steps. It uses the $\text{KeySchedule}_{256}^{15}$ algorithm specified in Algorithm 22, which generates sixteen 256-bit round keys, of which the last is used for final whitening, and cycles through all twenty-four constants $c_0, \dots, c_{23}$. Each step XORs one round key into the state, applies four parallel AFS-64 boxes with step-dependent constants, and then applies the four-branch linear layer $\mathcal{L}_4$ (see Algorithm 2). The encryption and decryption procedures are formally specified in Algorithms 23 and 24, respectively.

Algorithm 22 $\text{KeySchedule}_{256}^{15}$.
Input/Output: $K \in \mathbb{F}_2^{256}$ / $(K_0, \dots, K_{15}) \in (\mathbb{F}_2^{256})^{16}$

```

 $(k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7) \in (\mathbb{F}_2^{32})^8 \leftarrow K$ 
for  $s \in [0, 14]$  do
   $K_s \leftarrow ((k_0, k_1), (k_2, k_3), (k_4, k_5), (k_6, k_7))$ 
   $(k_0, k_1) \leftarrow (k_0 + k_1 + c_{2s \bmod 24}, k_1)$ 
   $(k_2, k_3) \leftarrow (k_2 \oplus k_3 \oplus s, k_3)$ 
   $(k_4, k_5) \leftarrow (k_4 + k_5 + c_{(2s+1) \bmod 24}, k_5)$ 
   $(k_6, k_7) \leftarrow (k_6 \oplus k_7 \oplus (s \ll 16), k_7)$ 
   $(k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7) \leftarrow (k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_0)$ 
end for
 $K_{15} \leftarrow ((k_0, k_1), (k_2, k_3), (k_4, k_5), (k_6, k_7))$ 
return  $(K_0, \dots, K_{15})$ 

```

Algorithm 23 Encryption for $\text{APEX}_{256}^{15}\text{-BC-256}$.
Input/Output: $P, K \in \mathbb{F}_2^{256}$ / $C \in \mathbb{F}_2^{256}$

```

 $(K_0, \dots, K_{15}) \leftarrow \text{KeySchedule}_{256}^{15}(K)$ 
 $S \leftarrow P$ 
for  $s \in [0, 14]$  do
   $((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)) \leftarrow S \oplus K_s$ 
  for  $i \in [0, 3]$  do
     $(x_i, y_i) \leftarrow \mathcal{A}_{c(4s+i) \bmod 24}(x_i, y_i)$ 
  end for
   $S \leftarrow \mathcal{L}_4((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3))$ 
end for
 $C \leftarrow S \oplus K_{15}$ 
return  $C$ 

```

Algorithm 24 Decryption for $\text{APEX}_{256}^{15}\text{-BC-256}$.
Input/Output: $C, K \in \mathbb{F}_2^{256}$ / $P \in \mathbb{F}_2^{256}$

```

 $(K_0, \dots, K_{15}) \leftarrow \text{KeySchedule}_{256}^{15}(K)$ 
 $S \leftarrow C \oplus K_{15}$ 
for  $s = 14, \dots, 0$  do
   $((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)) \leftarrow \mathcal{L}_4^{-1}(S)$ 
  for  $i \in [0, 3]$  do
     $(x_i, y_i) \leftarrow \mathcal{A}_{c(4s+i) \bmod 24}^{-1}(x_i, y_i)$ 
  end for
   $S \leftarrow ((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)) \oplus K_s$ 
end for
 $P \leftarrow S$ 
return  $P$ 

```

7.1.3 The Tweakable Block Cipher $\text{APEX}_{256}^{15}\text{-TBC-256/128}$

The tweakable block cipher $\text{APEX}_{256}^{15}\text{-TBC-256/128}$ follows the same fifteen-step structure as $\text{APEX}_{256}^{15}\text{-BC-256}$ and likewise operates on 256-bit blocks under a 256-bit key. It uses the $\text{KeySchedule}_{256}^{15}$ algorithm specified in Algorithm 22. The only additional operation is the injection of a 128-bit tweak: beginning with zero-based step 1, the tweak is XORed into the leftmost 128 state bits every two steps, namely before steps 1, 3, $\dots$, 13. All other operations, including final whitening, are unchanged. The encryption and decryption procedures are specified in Algorithms 25 and 26, respectively.

Algorithm 25 Encryption for $\text{APEX}_{256}^{15}\text{-TBC-256/128}$.
Input/Output: $P, K \in \mathbb{F}_2^{256}$, $T \in \mathbb{F}_2^{128}$ / $C \in \mathbb{F}_2^{256}$

```

 $(t_0, t_1, t_2, t_3) \in (\mathbb{F}_2^{32})^4 \leftarrow T$ 
 $(K_0, \dots, K_{15}) \leftarrow \text{KeySchedule}_{256}^{15}(K)$ 
 $S \leftarrow P$ 
for  $s \in [0, 14]$  do
   $((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)) \leftarrow S \oplus K_s$ 
  if  $s \bmod 2 = 1$  then
     $(x_0, y_0) \leftarrow (x_0, y_0) \oplus (t_0, t_1)$ 
     $(x_1, y_1) \leftarrow (x_1, y_1) \oplus (t_2, t_3)$ 
  end if
  for  $i \in [0, 3]$  do
     $(x_i, y_i) \leftarrow \mathcal{A}_{c(4s+i) \bmod 24}(x_i, y_i)$ 
  end for
   $S \leftarrow \mathcal{L}_4((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3))$ 
end for
 $C \leftarrow S \oplus K_{15}$ 
return  $C$ 

```

Algorithm 26 Decryption for $\text{APEX}_{256}^{15}\text{-TBC-256/128}$.
Input/Output: $C, K \in \mathbb{F}_2^{256}$, $T \in \mathbb{F}_2^{128}$ / $P \in \mathbb{F}_2^{256}$

```

 $(t_0, t_1, t_2, t_3) \in (\mathbb{F}_2^{32})^4 \leftarrow T$ 
 $(K_0, \dots, K_{15}) \leftarrow \text{KeySchedule}_{256}^{15}(K)$ 
 $S \leftarrow C \oplus K_{15}$ 
for  $s = 14, \dots, 0$  do
   $((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)) \leftarrow \mathcal{L}_4^{-1}(S)$ 
  for  $i \in [0, 3]$  do
     $(x_i, y_i) \leftarrow \mathcal{A}_{c(4s+i) \bmod 24}^{-1}(x_i, y_i)$ 
  end for
  if  $s \bmod 2 = 1$  then
     $(x_0, y_0) \leftarrow (x_0, y_0) \oplus (t_0, t_1)$ 
     $(x_1, y_1) \leftarrow (x_1, y_1) \oplus (t_2, t_3)$ 
  end if
   $S \leftarrow ((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)) \oplus K_s$ 
end for
 $P \leftarrow S$ 
return  $P$ 

```

7.2 Design Rationale and Security Analysis

7.2.1 Security Claim

We make no security claim against related-key attacks. Related-key security is a separate and stronger notion than the standard single-key model, since it grants the adversary access to evaluations under keys satisfying specified relations [68, 69]. The practical relevance of this stronger access model depends on the deployment environment. If

resistance to related-key attacks is required, the key schedule can be modified for the intended class of key relations and the resulting construction analyzed separately. For $\text{APEX}_{256}^{15}\text{-TBC-256/128}$, we claim 128-bit security in the related-tweak setting, provided that the total number of plaintext/ciphertext–tweak queries under any fixed key does not exceed 2^{128} . The secret key remains fixed and unknown, while the adversary may choose public tweaks satisfying adversarially specified relations.

7.2.2 $\text{APEX}_{64}^8\text{-BC-128}$

The ultralightweight block cipher $\text{APEX}_{64}^8\text{-BC-128}$ is built directly from the APEX_{64} step function. To reduce encryption and decryption latency, we follow the schedule-free key-alternation strategy of CRAX-S [30]. The 128-bit master key is divided into two 64-bit half-keys. The first and second half-keys are used alternately in the even- and odd-numbered steps, respectively, and the first half-key is also used for final whitening. This arrangement avoids both expanded-key generation and round-key storage. To further reduce the software cost, the cipher cyclically uses only the five constants $c_0, \dots, c_4$. This five-constant cycle is combined with the injected step index to prevent direct slide repetitions. Since the constants repeat with period five while the two half-keys alternate with period two, a repeated constant is paired with the opposite half-key in the latter part of the cipher. The step index further distinguishes the keyed steps.

Since $\text{APEX}_{64}^8\text{-BC-128}$ is built directly from the APEX_{64} core, its security analysis relies on the cryptanalytic results for APEX_{64} presented in Section 3.3. The longest distinguisher found in the analyses of that section covers no more than three complete steps of APEX_{64} . One APEX_{64} step reaches full diffusion under the dependency-matrix criterion, although this is only a weak structural condition, while two steps satisfy the strict avalanche criterion as shown in Section 3.3.3.6. Full key-to-state diffusion is also achieved after two steps. We therefore choose the step count using the heuristic rule $n_s = \lceil 2 + 2 + 1.2 \cdot 3 \rceil = 8$. The first two terms allocate two steps on each side of the three-step distinguisher core, corresponding to the number of steps required for full key-to-state diffusion. The factor 1.2 adds a conservative 20% margin to the longest distinguisher found, rounded upward. This gives the final choice $n_s = 8$.

7.2.3 $\text{APEX}_{256}^{15}\text{-BC-256}$ and $\text{APEX}_{256}^{15}\text{-TBC-256/128}$

The key schedule and tweak-injection strategy of $\text{APEX}_{256}^{15}\text{-BC-256}$ and $\text{APEX}_{256}^{15}\text{-TBC-256/128}$ follow the lightweight scheduling strategy of TRAX-L [30]. The 256-bit key state is updated with modular additions, XORs, step-dependent material, and a cyclic rotation of its eight 32-bit words. We use the complete set of 24 APEX constants in this update. The resulting schedule produces one 256-bit subkey for every step and a final whitening key. The block cipher and tweakable block cipher share this key schedule and the same APEX_{256} step function. Their security analysis in the single-key setting therefore builds on the cryptanalytic results for APEX_{256} in Section 3.3.

The tweakable instance has no separate tweak schedule. The same 128-bit tweak is injected into the first two 64-bit branches before the odd, zero-based steps $1, 3, \dots, 13$. The tweak is first injected at step 1 rather than step 0, since an injection at step 0 could otherwise be trivially cancelled using chosen plaintexts. A comparison of candidate injection cadences using the related-tweak long-trail model led us to retain an interval of two steps, which attains larger differential weights in fewer steps. This spacing forces tweak-induced activity through a double-AFS-64 segment before the next injection and thereby increases the accumulated long-trail weight. This choice is also consistent with the rationale used for TRAX-L [30].

We analyze this schedule by adapting the long-trail search techniques of [22, 30, 31, 36] to the related-tweak setting. The plaintext difference is unrestricted, while one fixed nonzero 128-bit tweak difference is reused at every tweak-injection step. Table 30 gives the resulting related-tweak differential long-trail weights. For $\text{APEX}_{256}^{15}\text{-TBC-256/128}$, we claim 128-bit related-tweak security under a limit of 2^{128} plaintext/ciphertext–tweak queries per key. As shown in Table 30, the related-tweak differential weight exceeds 128 after seven steps, where it reaches 148. We therefore select the step count using the heuristic rule $n_s = \lceil 3 + 3 + 1.2 \cdot 7 \rceil = 15$. The two three-step terms allocate the number of steps required for full key-to-state diffusion on both sides of the seven-step distinguisher core, while the factor 1.2 adds a conservative 20% margin.

For $\text{APEX}_{256}^{15}\text{-BC-256}$, the cryptanalysis of APEX_{256} in Section 3.3 finds no distinguisher covering more than seven complete steps (Table 3 gives differential weights 236 and 266 for seven and eight steps, respectively). Using a heuristic similar to the one above, we select the step count of $\text{APEX}_{256}^{15}\text{-BC-256}$ as $n_s = \lceil 3 + 3 + 1.2 \cdot 7 \rceil = 15$.

Table 30 Related-tweak differential long-trail weights for $\text{APEX}_{256}^{n_s}$. An entry w establishes an upper bound 2^{-w} on the probability of a related-tweak differential trail.

n_s	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
w	0	10	10	20	67	121	148	158	158	185	234	246	293	320	325	354

7.3 Implementation

Implementation environment. All execution times for the 32-bit ARM implementations were determined using the cycle-accurate instruction-set simulator of Keil μ Vision v5.24.2.0, with a generic Cortex-M3 model as the target device. The optimized C implementations were compiled with ARMCC 5.06 at optimization level -O2 with optimization for execution time. All execution times for the 8-bit AVR implementations were determined using the cycle-accurate instruction-set simulator of Microchip Studio 7, with the ATmega128 microcontroller as the target device. The corresponding C implementation was compiled with AVR-GCC 5.4.0 at optimization level -O2.

Table 31 Comparison of the implementation results of $\text{APEX}_{64}^8\text{-BC-128}$, SPECK and SIMON [11], CRAX-S [30], RECTANGLE [70], SPARX [22], and HIGHT [71], all with a 64-bit block and a 128-bit key. ROM and RAM consumption are measured in bytes, and the time for processing one 64-bit block is given in clock cycles. The APEX ARM results use optimized C, whereas the APEX AVR results use handwritten assembly. The results for the other ciphers are those reported in [30].

Platform	Cipher	Enc. 64-bit			Dec. 64-bit			Key schedule		
		ROM	RAM	Time	ROM	RAM	Time	ROM	RAM	Time
32-bit ARM (Cortex-M3)	$\text{APEX}_{64}^8\text{-BC-128}$	196	24	200	200	24	203	0	0	0
	SPECK	340	132	184	448	132	254	48	132	514
	CRAX-S	196	36	239	202	36	239	0	0	0
	SIMON	100	200	557	120	200	596	244	200	864
	RECTANGLE	412	232	714	484	232	843	157	232	1106
	SPARX	644	224	932	748	224	1065	756	224	620
	HIGHT	442	160	2500	450	160	2644	352	160	740
8-bit AVR (ATmega128)	$\text{APEX}_{64}^8\text{-BC-128}$	580	20	1043	578	20	1039	0	0	0
	SPECK	542	132	997	706	132	1139	178	132	1401
	CRAX-S	584	20	1257	582	20	1249	0	0	0
	SIMON	354	200	1973	364	200	1803	254	200	2911
	RECTANGLE	230	232	1823	230	232	1843	218	232	1832
	SPARX	712	224	1529	790	224	1676	642	224	844
	HIGHT	286	160	2770	288	160	2768	418	160	1191

Table 32 Performance comparison of $\text{APEX}_{256}^{15}\text{-BC-256}$, $\text{APEX}_{256}^{15}\text{-TBC-256/128}$, TRAX-L [30], and SATURNIN-256/256 [72] on a 32-bit ARM Cortex-M3. Results are given in clock cycles. The APEX results use optimized C. The SATURNIN-256/256 results were obtained from the latest implementation available in the [lightweight-crypto repository](#).

Instance	Key schedule	Encryption	Decryption
$\text{APEX}_{256}^{15}\text{-BC-256}$	697	1553	1650
$\text{APEX}_{256}^{15}\text{-TBC-256/128}$	697	1760	1777
TRAX-L	925	2435	2464
SATURNIN-256/256	233	2757	2725

Implementation results. As shown in Table 31, $\text{APEX}_{64}^8\text{-BC-128}$ is among the fastest and smallest 64-bit block ciphers in the comparison. Relative to CRAX-S, it retains a similar ROM and RAM footprint and the same absence of a separate key-schedule cost, while reducing the encryption and decryption times by approximately 15–17% on both platforms. Its closest competitor is SPECK: once the round keys are available, SPECK encrypts slightly faster, whereas $\text{APEX}_{64}^8\text{-BC-128}$ decrypts faster, uses substantially less RAM, and avoids the cost of generating round keys. It is therefore particularly attractive when only a small number of blocks is processed under one key.

The results for the 256-bit designs are summarized in Table 32. $\text{APEX}_{256}^{15}\text{-BC-256}$ encrypts and decrypts one block in 1553 and 1650 cycles, respectively, making it approximately 1.77 and 1.65 times faster than SATURNIN-256/256. Adding tweak support introduces only a modest overhead: $\text{APEX}_{256}^{15}\text{-TBC-256/128}$ requires 697 cycles for key scheduling and 1760 and 1777 cycles for encryption and decryption. These three operations are approximately 1.33, 1.38, and 1.39 times faster, respectively, than those of TRAX-L.

8 Conclusion

Currently, how to systematically design and accurately analyze cryptographic primitives that achieve an extreme and compact balance between security and implementation performance has become a core challenging problem in the field of ARX cryptography. In this paper, we have presented APEX, a scalable family of cryptographic permutations built around the AFS-64 ARX-box. Its uniform step function supports state widths ranging from 64 to 1536 bits, providing a common foundation for both lightweight and high-throughput symmetric-key designs. Using this family, we specified small-state hash functions, extendable-output functions, and AEAD schemes, along with an ultralightweight block cipher, a large-state block cipher, and a large-state tweakable block cipher. We further developed large-state HASH-F instances tailored to the requirements of the Next-Generation Commercial Cryptographic Algorithms program.

The proposed constructions are supported by a unified design rationale and a comprehensive security analysis. We derived differential and linear long-trail bounds for the unrestricted permutations and adapted the corresponding searches to mode- and construction-specific constraints, including rate-restricted, same-capacity, and related-tweak settings. Together with our analyses of other relevant attack vectors, these results justify the selected step counts and support the claimed security levels.

Software implementations on 8-bit AVR, 32-bit ARMv7-M, and x86-64 platforms show that APEX can be implemented efficiently across markedly different architectures. In particular, the hand-optimized assembly and vectorized implementations demonstrate that the regular AFS-based structure scales from resource-constrained embedded processors to high-throughput SIMD software on x86-64. Taken together, these results demonstrate the versatility of APEX as a permutation foundation for a broad range of symmetric primitives.

Appendix A Details of integral cryptanalysis of the APEX permutations

Table A1 A partial list of the best integral distinguishers for the APEX permutation family. ? denotes a possibly nonbalanced 32-bit word, **B** means that all 32 bits are balanced, and **b** means that some of the 32 bits are balanced. A superscript denotes consecutive repetitions of the corresponding 32-bit-word symbol. One APEX- b step for $b \geq 128$ comprises an AFS-64 layer (four rounds) and the linear layer; one APEX-64 step is one AFS-64 application.

Instance	Inactive bit	Number of steps/rounds	Balanced pattern
APEX ₆₄	63	1 step +1 round	b?
APEX ₆₄ ⁻¹	30	1 step +1 round	?b
APEX ₁₂₈	63	3 steps +4 rounds	??BB
APEX ₁₂₈ ⁻¹	30 or 94	3 steps +0 rounds	BB??
APEX ₂₅₆	64 or 192	4 steps +4 rounds	? ⁴ B ⁴
APEX ₂₅₆ ⁻¹	30 or 158	4 steps +0 rounds	B ⁴ ? ⁴
APEX ₃₈₄	64 or 256	4 steps +4 rounds	? ⁶ B ⁶
APEX ₃₈₄ ⁻¹	30 or 222	4 steps +0 rounds	B ⁶ ? ⁶
APEX ₅₁₂	64 or 320	4 steps +4 rounds	? ⁸ B ⁸
APEX ₅₁₂ ⁻¹	30 or 286	4 steps +0 rounds	B ⁸ ? ⁸
APEX ₆₄₀	64 or 384	4 steps +4 rounds	? ¹⁰ B ¹⁰
APEX ₆₄₀ ⁻¹	30 or 350	4 steps +0 rounds	B ¹⁰ ? ¹⁰
APEX ₇₆₈	64 or 448	4 steps +4 rounds	? ¹² B ¹²
APEX ₇₆₈ ⁻¹	30 or 414	4 steps +0 rounds	B ¹² ? ¹²
APEX ₈₉₆	64 or 512	4 steps +4 rounds	? ¹⁴ B ¹⁴
APEX ₈₉₆ ⁻¹	30 or 478	4 steps +0 rounds	B ¹⁴ ? ¹⁴
APEX ₁₀₂₄	64 or 576	4 steps +4 rounds	? ¹⁶ B ¹⁶
APEX ₁₀₂₄ ⁻¹	30 or 542	4 steps +0 rounds	B ¹⁶ ? ¹⁶
APEX ₁₁₅₂	64 or 640	4 steps +4 rounds	? ¹⁸ B ¹⁸
APEX ₁₁₅₂ ⁻¹	30 or 606	4 steps +0 rounds	B ¹⁸ ? ¹⁸
APEX ₁₂₈₀	64 or 704	4 steps +4 rounds	? ²⁰ B ²⁰
APEX ₁₂₈₀ ⁻¹	30 or 670	4 steps +0 rounds	B ²⁰ ? ²⁰
APEX ₁₄₀₈	64 or 768	4 steps +4 rounds	? ²² B ²²
APEX ₁₄₀₈ ⁻¹	30 or 734	4 steps +0 rounds	B ²² ? ²²
APEX ₁₅₃₆	64 or 832	4 steps +4 rounds	? ²⁴ B ²⁴
APEX ₁₅₃₆ ⁻¹	30 or 798	4 steps +0 rounds	B ²⁴ ? ²⁴

Appendix B Details of impossible-differential and zero-correlation linear cryptanalysis of the APEX permutations

B.1 Impossible differentials

α , β , γ , and δ denote arbitrary nonzero 64-bit differences.

For APEX₁₂₈, a four-step impossible-differential distinguisher is given by

$$\begin{array}{lll} \text{input : } 0 & \alpha & \downarrow \\ \text{step 1 : } \gamma & 0 & \downarrow \\ \text{step 2 : } \not\downarrow & & \\ \text{step 3 : } \delta & 0 & \uparrow \\ \text{step 4 : } \beta & \beta & \uparrow \end{array}$$

For every supported APEX_b with $b \geq 256$, a four-step impossible-differential distinguisher is given by the following pattern, where each side of $\parallel$ contains $n_b/2$ branches and the ellipses denote zero differences in all omitted branches:

$$\begin{array}{lll} \text{input : } 0 & 0 \cdots 0 & \parallel \alpha & 0 \cdots 0 & \downarrow \\ \text{step 1 : } 0 & 0 \cdots \gamma & \parallel 0 & 0 \cdots 0 & \downarrow \\ \text{step 2 : } \not\downarrow & & \parallel & & \\ \text{step 3 : } 0 & 0 \cdots 0 & \parallel 0 & \delta \cdots 0 & \uparrow \\ \text{step 4 : } \beta & 0 \cdots 0 & \parallel 0 & 0 \cdots 0 & \uparrow \end{array}$$

B.2 Zero-correlation linear cryptanalysis

α , β , γ , and δ denote arbitrary nonzero 64-bit linear masks.

For APEX₁₂₈, a four-step zero-correlation linear distinguisher is given by

$$\begin{array}{lll} \text{input : } \alpha & 0 & \downarrow \\ \text{step 1 : } 0 & \gamma & \downarrow \\ \text{step 2 : } \not\downarrow & & \\ \text{step 3 : } 0 & \delta & \uparrow \\ \text{step 4 : } \beta & \beta & \uparrow \end{array}$$

For every supported APEX_b with $b \geq 256$, a four-step zero-correlation linear distinguisher is given by the following pattern, where each side of $\parallel$ contains $n_b/2$ branches and the ellipses denote zero masks in all omitted branches:

$$\begin{array}{lll} \text{input : } \alpha & 0 \cdots 0 & \parallel 0 & 0 \cdots 0 & \downarrow \\ \text{step 1 : } 0 & 0 \cdots 0 & \parallel \gamma & 0 \cdots 0 & \downarrow \\ \text{step 2 : } \not\downarrow & & \parallel & & \\ \text{step 3 : } \delta & 0 \cdots 0 & \parallel 0 & 0 \cdots 0 & \uparrow \\ \text{step 4 : } 0 & 0 \cdots 0 & \parallel \beta & 0 \cdots 0 & \uparrow \end{array}$$

Acknowledgements

This work was supported by National Cyber Security–National Science and Technology Major Project (2026ZD1500600, 2026ZD1500604). Zhiguang Yan thanks the organizers and lecturers of the 14th Advanced Cryptography Training Workshop on Symmetric-Cipher Design and Analysis, hosted by the Chinese Association for Cryptologic Research, for their valuable lectures and discussions.

References

- [1] Wheeler, D.J., Needham, R.M.: TEA, a tiny encryption algorithm. In: Fast Software Encryption – FSE 1994. Lecture Notes in Computer Science, vol. 1008, pp. 363–366. Springer, Berlin, Heidelberg (1994). https://doi.org/10.1007/3-540-60590-8_29
- [2] Needham, R.M., Wheeler, D.J.: TEA extensions. Technical report, University of Cambridge Computer Laboratory (October 1997). <https://www.movable-type.co.uk/scripts/xtea.pdf>
- [3] Bernstein, D.J.: The Salsa20 family of stream ciphers. In: New Stream Cipher Designs: The eSTREAM Finalists. Lecture Notes in Computer Science, vol. 4986, pp. 84–97. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68351-3_8
- [4] Bernstein, D.J.: ChaCha, a Variant of Salsa20. Workshop Record of SASC 2008 (2008). <https://cr.yp.to/chacha/chacha-20080128.pdf>
- [5] Bernstein, D.J.: The Poly1305-AES message-authentication code. In: Fast Software Encryption – FSE 2005. Lecture Notes in Computer Science, vol. 3557, pp. 32–49. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11502760_3
- [6] Nir, Y., Langley, A.: ChaCha20 and Poly1305 for IETF protocols. Request for Comments 8439, Internet Research Task Force (June 2018). <https://doi.org/10.17487/RFC8439>
- [7] Langley, A., Chang, W.-T., Mavrogiannopoulos, N., Strombergson, J., Josefsson, S.: ChaCha20-Poly1305 cipher suites for transport layer security (TLS). Request for Comments 7905, Internet Engineering Task Force (June 2016). <https://doi.org/10.17487/RFC7905>
- [8] Rescorla, E.: The transport layer security (TLS) protocol version 1.3. Request for Comments 9846, Internet Engineering Task Force (March 2026). <https://doi.org/10.17487/RFC9846>
- [9] Thomson, M., Turner, S.: Using TLS to secure QUIC. Request for Comments 9001, Internet Engineering Task Force (May 2021). <https://doi.org/10.17487/RFC9001>
- [10] Nir, Y.: ChaCha20, Poly1305, and their use in the internet key exchange protocol (IKE) and IPsec. Request for Comments 7634, Internet Engineering Task Force (August 2015). <https://doi.org/10.17487/RFC7634>
- [11] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK lightweight block ciphers. In: Proceedings of the 52nd Annual Design Automation Conference, pp. 1–6. ACM, New York, NY, USA (2015). <https://doi.org/10.1145/2744769.2747946>
- [12] Hong, D., Lee, J.-K., Kim, D.-C., Kwon, D., Ryu, K.H., Lee, D.: LEA: A 128-bit block cipher for fast encryption on common processors. In: Information Security Applications – WISA 2013. Lecture Notes in Computer Science, vol. 8267, pp. 3–27. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-05149-9_1
- [13] Korean Agency for Technology and Standards: 128-Bit Block Cipher LEA. Korean Industrial Standard KS X 3246 (2016). https://www.rra.go.kr/ko/reference/kcsList_view.do?nb_seq=1923&nb_type=6
- [14] Cui, T., Wang, M., Fan, Y., Hu, K., Fu, Y., Huang, L.: Ballet: A software-friendly block cipher. Journal of Cryptologic Research 6(6), 704–712 (2019) <https://doi.org/10.13868/j.cnki.jcr.000335>
- [15] Lai, X., Massey, J.L.: A proposal for a new block encryption standard. In: Advances in Cryptology – EURO-CRYPT ’90. Lecture Notes in Computer Science, vol. 473, pp. 389–404. Springer, Berlin, Heidelberg (1991). https://doi.org/10.1007/3-540-46877-3_35
- [16] Chinese Association for Cryptologic Research: Announcement of the Evaluation Results of the National Cryptographic Algorithm Design Competition (2020). <https://www.cacrnet.org.cn/site/content/854.html>

- [17] Aumasson, J.-P., Henzen, L., Meier, W., Phan, R.C.-W.: SHA-3 Proposal BLAKE. Submission to the NIST SHA-3 Competition, version 1.3 (2010). <https://www.aumasson.jp/blake/blake.pdf>
- [18] Chang, S.-j.H., Perlner, R., Burr, W.E., Turan, M.S., Kelsey, J., Paul, S., Bassham, L.E.: Third-round report of the SHA-3 cryptographic hash algorithm competition. NIST Interagency/Internal Report 7896, National Institute of Standards and Technology (2012). <https://doi.org/10.6028/NIST.IR.7896>
- [19] Aumasson, J.-P., Neves, S., Wilcox-O’Hearn, Z., Winnerlein, C.: BLAKE2: Simpler, smaller, fast as MD5. In: Applied Cryptography and Network Security – ACNS 2013. Lecture Notes in Computer Science, vol. 7954, pp. 119–135. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38980-1_8
- [20] Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family. SHA-3 Algorithm Submission Version 1.3, National Institute of Standards and Technology (2010). <https://www.schneier.com/academic/skein/>
- [21] Daemen, J., Rijmen, V.: The wide trail design strategy. In: Cryptography and Coding. Lecture Notes in Computer Science, vol. 2260, pp. 222–238. Springer, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-45325-3_20
- [22] Dinu, D., Perrin, L., Udovenko, A., Velichkov, V., Großschädl, J., Biryukov, A.: Design strategies for ARX with provable bounds: SPARX and LAX. In: Advances in Cryptology – ASIACRYPT 2016. Lecture Notes in Computer Science, vol. 10031, pp. 484–513. Springer, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53887-6_18
- [23] Mishra, S., Sadhya, D.: LiARX: A lightweight cipher based on the LTS design strategy of ARX. In: Information Systems Security. Lecture Notes in Computer Science, vol. 12553, pp. 185–197. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65610-2_12
- [24] AlTawy, R., Rohit, R., He, M., Mandal, K., Yang, G., Gong, G.: sLiSCP: Simeck-based permutations for lightweight sponge cryptographic primitives. In: Selected Areas in Cryptography – SAC 2017. Lecture Notes in Computer Science, vol. 10719, pp. 129–150. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-72565-9_7
- [25] AlTawy, R., Rohit, R., He, M., Mandal, K., Yang, G., Gong, G.: sLiSCP-light: Towards hardware-optimized sponge-specific cryptographic permutations. ACM Transactions on Embedded Computing Systems **17**(4), 81–18126 (2018) <https://doi.org/10.1145/3233245>
- [26] Yang, G., Zhu, B., Suder, V., Aagaard, M.D., Gong, G.: The Simeck family of lightweight block ciphers. In: Cryptographic Hardware and Embedded Systems – CHES 2015. Lecture Notes in Computer Science, vol. 9293, pp. 307–329. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48324-4_16
- [27] AlTawy, R., Gong, G., He, M., Mandal, K., Rohit, R.: Spix: An Authenticated Cipher. NIST Lightweight Cryptography Round-2 Candidate Specification. 27 September 2019 (2019). <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/spix-spec-round2.pdf>
- [28] AlTawy, R., Gong, G., He, M., Jha, A., Mandal, K., Nandi, M., Rohit, R.: Spoc: An Authenticated Cipher. NIST Lightweight Cryptography Round-2 Candidate Specification. 26 September 2019 (2019). <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/spoc-spec-round2.pdf>
- [29] National Institute of Standards and Technology: Status report on the second round of the NIST lightweight cryptography standardization process. NIST Interagency/Internal Report 8369, National Institute of Standards and Technology (2021). <https://doi.org/10.6028/NIST.IR.8369>
- [30] Beierle, C., Biryukov, A., Santos, L.C., Großschädl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q.: Alzette: A 64-bit ARX-box (feat. CRAX and TRAX). In: Advances in Cryptology – CRYPTO 2020. Lecture Notes in Computer Science, vol. 12172, pp. 419–448. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_15
- [31] Beierle, C., Biryukov, A., Santos, L.C., Großschädl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q.:

- Lightweight AEAD and hashing using the Sparkle permutation family. *IACR Transactions on Symmetric Cryptology* **2020**(S1), 208–261 (2020) <https://doi.org/10.13154/tosc.v2020.iS1.208-261>
- [32] Dobraunig, C., Eichlseder, M., Mendel, F., Schl  ffer, M.: Ascon v1.2: Lightweight authenticated encryption and hashing. *Journal of Cryptology* **34**(3), 33 (2021) <https://doi.org/10.1007/s00145-021-09398-9>
- [33] Turan, M.S., McKay, K., Chang, D., Bassham, L.E., Kang, J., Waller, N.D., Kelsey, J.M., Hong, D.: Status report on the final round of the NIST lightweight cryptography standardization process. NIST Interagency/Internal Report 8454, National Institute of Standards and Technology (2023). <https://doi.org/10.6028/NIST.IR.8454>
- [34] Daemen, J., Hoffert, S., Peeters, M., Assche, G.V., Keer, R.V.: Xoodyak, a lightweight cryptographic scheme. *IACR Transactions on Symmetric Cryptology* **2020**(S1), 60–87 (2020) <https://doi.org/10.13154/tosc.v2020.iS1.60-87>
- [35] Fan, T., Feng, W., Wei, Y.: The design and cryptanalysis of a large-state lightweight cryptographic S-box. *Journal of Xidian University* **50**(4), 170–179 (2023) <https://doi.org/10.19665/j.issn1001-2400.2023.04.017>
- [36] Yan, Z., Wei, Y., Rodr  guez-Aldama, R., Pasalic, E.: AFS: A Family of ARX-Based Large-State S-boxes with Exceptional Properties. *Cryptology ePrint Archive*, Paper 2026/1672 (2026). <https://eprint.iacr.org/2026/1672>
- [37] Guo, C., Hu, K., Jiang, S., Fan, Y., Fu, Y., Preneel, B., Wang, M.: Permutation-Based Hash from Non-Idealized Assumptions: Adding Feed-Forward to Sponge. *IACR Cryptology ePrint Archive*, Report 2025/1006; full version of CRYPTO 2026 (2026). <https://eprint.iacr.org/2025/1006>
- [38] Institute of Commercial Cryptography Standards, China: Announcement on Launching the Next-generation Commercial Cryptographic Algorithms Program (NGCC) (2025). <https://www.niccs.org.cn/niccs/home/content/1937426388645056512/content.1937426388645056512.html>
- [39] Beyne, T., Rijmen, V.: Differential cryptanalysis in the fixed-key model. In: *Advances in Cryptology – CRYPTO 2022, Part III. Lecture Notes in Computer Science*, vol. 13509, pp. 687–716. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15982-4_23
- [40] Todo, Y.: Structural evaluation by generalized integral property. In: *Advances in Cryptology – EUROCRYPT 2015, Part I. Lecture Notes in Computer Science*, vol. 9056, pp. 287–314. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_12
- [41] Hu, K., Sun, S., Wang, M., Wang, Q.: An algebraic formulation of the division property: Revisiting degree evaluations, cube attacks, and key-independent sums. In: *Advances in Cryptology – ASIACRYPT 2020, Part I. Lecture Notes in Computer Science*, vol. 12491, pp. 446–476. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64837-4_15
- [42] Hu, K., Yap, T.: Perfect monomial prediction for modular addition. *IACR Transactions on Symmetric Cryptology* **2024**(3), 177–199 (2024) <https://doi.org/10.46586/tosc.v2024.i3.177-199>
- [43] Beierle, C., Biryukov, A., Santos, L.C., Gro  sch  dl, J., Moradi, A., Perrin, L., Shahmirzadi, A.R., Udovenko, A., Velichkov, V., Wang, Q.: Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing Using the Sparkle Permutation Family. NIST Lightweight Cryptography Finalist Specification, version v1.2. Version v1.2, 17 May 2021 (2021). <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/sparkle-spec-final.pdf>
- [44] Wagner, D.: The boomerang attack. In: *Fast Software Encryption – FSE ’99. Lecture Notes in Computer Science*, vol. 1636, pp. 156–170. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48519-8_12
- [45] Cui, T., Chen, S., Fu, K., Wang, M., Jia, K.: New automatic tool for finding impossible differentials and zero-correlation linear approximations. *Science China Information Sciences* **64**(2), 129103 (2021) <https://doi.org/10.1007/s11432-018-1506-4>

- [46] Sasaki, Y., Todo, Y.: New impossible differential search tool from design and cryptanalysis aspects: Revealing structural properties of several ciphers. In: *Advances in Cryptology – EUROCRYPT 2017, Part III. Lecture Notes in Computer Science*, vol. 10212, pp. 185–215. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_7
- [47] Biham, E., Biryukov, A., Dunkelman, O., Richardson, E., Shamir, A.: Initial observations on Skipjack: Cryptanalysis of Skipjack-3XOR (invited talk). In: *Selected Areas in Cryptography – SAC 1998. Lecture Notes in Computer Science*, vol. 1556, pp. 362–376. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48892-8_27
- [48] Webster, A.F., Tavares, S.E.: On the design of S-boxes. In: *Advances in Cryptology – CRYPTO '85. Lecture Notes in Computer Science*, vol. 218, pp. 523–534. Springer, Berlin, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_41
- [49] Biryukov, A., Wagner, D.A.: Slide attacks. In: *Fast Software Encryption – FSE '99. Lecture Notes in Computer Science*, vol. 1636, pp. 245–259. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48519-8_18
- [50] Hirose, S.: Sequential hashing with minimum padding. In: *NIST Workshop on Lightweight Cryptography 2016* (2016). National Institute of Standards and Technology
- [51] Chakraborti, A., Datta, N., Nandi, M., Yasuda, K.: Beetle family of lightweight and secure authenticated encryption ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2018**(2), 218–241 (2018) <https://doi.org/10.13154/tches.v2018.i2.218-241>
- [52] Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Duplexing the sponge: Single-pass authenticated encryption and other applications. In: *Selected Areas in Cryptography – SAC 2011. Lecture Notes in Computer Science*, vol. 7118, pp. 320–337. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28496-0_19
- [53] Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: *Sponge Functions*. ECRYPT Hash Workshop (2007)
- [54] Vaudenay, S., Vizár, D.: Under Pressure: Security of CAESAR Candidates beyond Their Guarantees. *Cryptology ePrint Archive*, Paper 2017/1147 (2017). <https://eprint.iacr.org/2017/1147>
- [55] Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. In: *Advances in Cryptology – CRYPTO '90. Lecture Notes in Computer Science*, vol. 537, pp. 2–21. Springer, Berlin, Heidelberg (1991). https://doi.org/10.1007/3-540-38424-3_1
- [56] Matsui, M.: Linear cryptanalysis method for DES cipher. In: *Advances in Cryptology – EUROCRYPT '93. Lecture Notes in Computer Science*, vol. 765, pp. 386–397. Springer, Berlin, Heidelberg (1994). https://doi.org/10.1007/3-540-48285-7_33
- [57] Knudsen, L.R.: Truncated and higher order differentials. In: *Fast Software Encryption – FSE '94. Lecture Notes in Computer Science*, vol. 1008, pp. 196–211. Springer, Berlin, Heidelberg (1995). https://doi.org/10.1007/3-540-60590-8_16
- [58] Knudsen, L.R.: *DEAL: A 128-Bit Block Cipher*. NIST AES Proposal (1998)
- [59] Biham, E., Biryukov, A., Shamir, A.: Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. In: *Advances in Cryptology – EUROCRYPT '99. Lecture Notes in Computer Science*, vol. 1592, pp. 12–23. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48910-X_2
- [60] Bogdanov, A., Rijmen, V.: Linear hulls with correlation zero and linear cryptanalysis of block ciphers. *Designs, Codes and Cryptography* **70**(3), 369–383 (2014) <https://doi.org/10.1007/s10623-012-9697-z>
- [61] Daemen, J., Knudsen, L.R., Rijmen, V.: The block cipher SQUARE. In: *Fast Software Encryption – FSE '97. Lecture Notes in Computer Science*, vol. 1267, pp. 149–165. Springer, Berlin, Heidelberg (1997). <https://doi.org/10.1007/BFb0052343>

- [62] Knudsen, L.R., Wagner, D.: Integral cryptanalysis. In: Fast Software Encryption – FSE 2002. Lecture Notes in Computer Science, vol. 2365, pp. 112–127. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45661-9_9
- [63] Balasch, J., Ege, B., Eisenbarth, T., Gérard, B., Gong, Z., Güneysu, T., Heyse, S., Kerckhof, S., Koeune, F., Plos, T., Pöppelmann, T., Regazzoni, F., Standaert, F.-X., Assche, G.V., Keer, R.V., Oldenzeel, L., Maurich, I.: Compact implementation and performance evaluation of hash functions in ATtiny devices. In: Smart Card Research and Advanced Applications – CARDIS 2012. Lecture Notes in Computer Science, vol. 7771, pp. 158–172. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37288-9_11
- [64] Bernstein, D.J., Kölbl, S., Lucks, S., Massolino, P.M.C., Mendel, F., Nawaz, K., Schneider, T., Schwabe, P., Standaert, F.-X., Todo, Y., Viguier, B.: Gimli: A cross-platform permutation. In: Cryptographic Hardware and Embedded Systems – CHES 2017. Lecture Notes in Computer Science, vol. 10529, pp. 299–320. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_15
- [65] Institute of Commercial Cryptography Standards: Submission Requirements for Cryptographic Hash Algorithms. Next-generation Commercial Cryptographic Algorithms Program (2025). <https://www.niccs.org.cn/niccs/Notice/1975892908773478400/52tbw9IQ.pdf>
- [66] Institute of Commercial Cryptography Standards: Evaluation Criteria for Cryptographic Hash Algorithms. Next-generation Commercial Cryptographic Algorithms Program (2025). <https://www.niccs.org.cn/niccs/Notice/1975892908773478400/crIRB1ZY.pdf>
- [67] Standardization Administration of China: Information Security Technology—SM3 Cryptographic Hash Algorithm. GB/T 32905-2016 (2016)
- [68] Biham, E.: New types of cryptanalytic attacks using related keys. *Journal of Cryptology* **7**(4), 229–246 (1994) <https://doi.org/10.1007/BF00203965>
- [69] Bellare, M., Kohno, T.: A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications. In: Advances in Cryptology—EUROCRYPT 2003. Lecture Notes in Computer Science, vol. 2656, pp. 491–506. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_31
- [70] Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: RECTANGLE: A bit-slice lightweight block cipher suitable for multiple platforms. *Science China Information Sciences* **58**(12), 1–15 (2015) <https://doi.org/10.1007/s11432-015-5459-7>
- [71] Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B.-S., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S.: HIGHT: A new block cipher suitable for low-resource device. In: Cryptographic Hardware and Embedded Systems – CHES 2006. Lecture Notes in Computer Science, vol. 4249, pp. 46–59. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11894063_4
- [72] Canteaut, A., Duval, S., Leurent, G., Naya-Plasencia, M., Perrin, L., Pornin, T., Schrottenloher, A.: Saturnin: A suite of lightweight symmetric algorithms for post-quantum security. *IACR Transactions on Symmetric Cryptology* **2020**(S1), 160–207 (2020) <https://doi.org/10.46586/tosc.v2020.iS1.160-207>