HiAE: A High-Throughput Authenticated Encryption Algorithm for Cross-Platform Efficiency

Han Chen[1], Tao Huang[1], Phuong Pham[1] and Shuang Wu[1]

Huawei International Pte. Ltd., Singapore,
concyclics@gmail.com,{huangtao80,pham.phuong,Wu.Shuang}@huawei.com

Abstract. This paper addresses the critical challenges in designing cryptographic algorithms that achieve both high performance and cross-platform efficiency on ARM and x86 architectures, catering to the demanding requirements of next-generation communication systems, such as 6G and GPU/NPU interconnections. We propose HiAE, a high-throughput authenticated encryption algorithm optimized for performance exceeding 100 Gbps and designed to meet the stringent security requirements of future communication networks. HiAE leverages the stream cipher structure, integrating the AES round function for non-linear diffusion.
Our design achieves exceptional efficiency, with benchmark results from software implementations across various platforms showing over 180 Gbps on ARM devices in AEAD mode, making it the fastest AEAD solution on ARM chips.

Keywords: High-throughput · Authenticated Encryption · 5G/6G · AES-NI

1 Introduction

The growing demand for high-performance and secure cryptographic algorithms is driven by rapid advancements across various fields of modern communication and data transmission technologies. As data transmission rates continue to rise, particularly with the anticipated arrival of 6G, where speeds are expected to exceed 100 Gbps [LAL+19], the pressure on cryptographic systems to balance both security and performance becomes increasingly critical. This trend extends well beyond mobile communication networks, influencing diverse sectors such as high-performance data centers, AI and machine learning, Ethernet communication, software-defined networks (SDN), and Cloud Radio Access Networks (Cloud RAN).

In high-performance computing (HPC) environments, such as data centers, the demand for high-bandwidth communication between servers and storage devices is pushing the limits of interconnects, with speeds reaching 100 Gbps to 400 Gbps or more for data center backbone links. Similarly, in AI and machine learning, accelerators like Nvidia's NVLink, capable of transferring data at rates up to 1,800 GB/s, are driving the need for encryption solutions that can keep pace with the high-speed interconnects used in distributed training of large-language models [Corc]. Furthermore, with the growing deployment of 100GbE and 400GbE Ethernet networks in data centers and enterprise environments, cryptographic algorithms must perform efficiently at ultra-high throughput levels to ensure secure data exchange without compromising performance.

In SDN and Cloud RAN, the demand for high-speed encryption is particularly pronounced. These architectures rely on efficient data processing across distributed environments, where cryptographic algorithms are tasked with securing high-volume data flows.

The encryption solutions must operate efficiently on general-purpose CPUs, including both x86 and ARM platforms, which are the backbone of most computing infrastructures in these domains.

Modern hardware advancements, particularly Single Instruction Multiple Data (SIMD) instructions, have enabled significant performance gains in cryptographic algorithms. SIMD allows parallel execution of operations like XOR and modular arithmetic, leveraging instruction sets such as Intel's SSE/AVX/AVX2/AVX512 [Cor24] and ARM's NEON [Com24a]. Moreover, the widespread support for AES-NI [Cora] and NEON cryptographic extension [BS12], optimized for AES round functions, has made it a cornerstone for high-performance cryptographic primitives. However, while AES-based designs like AEGIS [WP14] and Rocca [SLN$^+$22] have demonstrated potential for 5G and early 6G requirements, they still face challenges in achieving cross-platform efficiency, especially as ARM architecture has been widely used for edge and mobile devices which require more on wireless communication ability.

1.1 Motivation

Many recent cryptographic designs have utilized SIMD instructions to achieve high performance, particularly on x86 platforms using AES-NI. AES-NI has become the foundation for many recent high-speed (authenticated) encryption algorithms like AEGIS, SNOW-V [EJMY18], and Rocca, which are tailored to take advantage of the parallelism and efficiency offered by these instructions. However, these designs often neglect the architectural differences between x86 and ARM, where SIMD instructions are implemented via NEON rather than AES-NI. This oversight results in inconsistent performance when deploying these algorithms on ARM-based devices, which dominate mobile and embedded systems.

The transition to 6G, with its demand for ultra-high data rates and reliance on SDN or cloud RAN, further emphasizes the need for cryptographic algorithms optimized for diverse platforms. While some existing designs achieve remarkable performance on x86—reaching or exceeding 100 Gbps—these same algorithms often perform suboptimally on ARM platforms due to differences in SIMD instruction sets and hardware support for AES round functions. This gap highlights the pressing need for a unified approach that ensures high and consistent performance across both architectures.

Addressing this challenge requires rethinking cryptographic design to leverage the unique capabilities of each platform while maintaining compatibility and efficiency. This motivates our work in developing a cross-platform cryptographic primitive that achieves competitive performance on both x86 and ARM architectures, meeting the stringent demands of 6G systems.

1.2 Contributions

This work addresses the challenges of designing cryptographic algorithms that achieve high performance and cross-platform efficiency for both ARM and x86 architectures, ensuring suitability for next-generation communication systems. The main contributions of this paper are as follows:

Cross-Platform Pipeline Optimization Analysis. We conduct a thorough analysis of the pipeline architectures of modern ARM and x86 processors, examining the nuances of their SIMD instruction sets and cryptographic capabilities. Our investigation focuses on key factors such as instruction throughput, latency, and execution unit utilization to identify architectural differences that influence performance. Based on this analysis, we derive the optimal ratio between AES round instructions and SIMD XOR instructions, ensuring efficient pipeline utilization tailored to the distinct characteristics of each platform.

A New Cross-Platform efficient AEAD Design. We design a new high-performance authenticated encryption with associated data (AEAD) algorithm, utilizing an innovative cross-platform structure as build block. We would like to highlight the following contributions in our design:

- **Cross-Platform High-Throughput Structure:** We introduce a novel XAXX structure that optimizes performance across both x86 and ARM platforms. This structure balances AES and XOR operations to achieve high instruction-level parallelism and efficient pipeline usage, adapting to the architectural differences between the platforms.
- **Efficient Update Function:** The update function efficiently integrates message blocks and optimizes AES operations while ensuring robust security. Compared to previous designs like Rocca, our design improves upon it by reducing the number of AES instructions needed to process a 128-bit plaintext from 3 to 2, minimizing data dependencies and carefully managing internal state updates. While the same level of security is maintained, our design boosts performance, bringing it closer to the maximum capabilities of available resources.

We also perform a comprehensive security analysis of our design, demonstrating its robustness against a wide range of attacks, including differential, linear, forgery, state recovery, and integral attacks.

Comprehensive Benchmarking Across Devices. We conducted extensive benchmarking of the proposed cipher across a wide range of devices, including both mobile processors and server-grade CPUs, to comprehensively evaluate its performance. Our experimental setup covered the latest ARM-based architectures to ensure a robust and diverse evaluation.

The results of our benchmarks demonstrate that the proposed cipher achieves exceptional performance. Specifically, on ARM CPUs, it achieves an impressive 97 Gbps, outperforming all previous designs on ARM architectures.

These results highlight the efficiency of our design and its ability to leverage the advanced instruction sets available in modern x86 and ARM processors. By achieving such high performance, our cipher not only ensures robust security but also meets the demanding throughput requirements of contemporary applications, making it a state-of-the-art solution in its category.

1.3 Organization of the Paper

The organization of the rest of this paper is as follows:

Section 2 discusses the AES instructions and SIMD optimizations, providing a cross-platform analysis. In Section 3, we present the detailed specification of HiAE. The design rationale behind our approach is elaborated in Section 4. Section 5 covers the security analysis, ensuring the robustness of our design. In Section 6, we evaluate the performance of the proposed methods. Finally, the paper concludes in Section 7.

2 AES Instructions and SIMD Optimization: Cross-Platform Analysis

AES is widely used for securing data confidentiality. To enhance the efficiency of AES computations, most modern processors integrate AES-specific cryptographic instructions into their SIMD instruction sets. This integration is especially notable in the two most widely adopted processor architectures: Intel's AES-NI (Advanced Encryption Standard New Instructions) [Cora] on x86 and the NEON cryptographic extension [Com24a] on

ARM. By leveraging these AES instructions, processors can execute AES round functions significantly faster than traditional software implementations.

2.1 The Implementation difference of AES Instructions between x86 and ARM architectures

The implementation of AES instructions varies between x86-64 and ARM processors. In x86-64 processors, the AES-NI set set provides several instructions for the AES round function, including both the internal rounds and the final round. These instructions include:

$$\begin{aligned}
\text{aesenc}(S, K) &= (\text{MixColumns} \circ \text{SubBytes} \circ \text{ShiftRows}(S)) \oplus K \\
\text{aesdec}(S, K) &= (\text{InvMixColumns} \circ \text{InvSubBytes} \circ \text{InvShiftRows}(S)) \oplus K \\
\text{aesenclast}(S, K) &= (\text{SubBytes} \circ \text{ShiftRows}(S)) \oplus K \\
\text{aesdeclast}(S, K) &= (\text{InvSubBytes} \circ \text{InvShiftRows}(S)) \oplus K
\end{aligned}$$

Conversely, the ARM NEON cryptographic extension separates the *MixColumns* operation, providing distinct instructions to handle this transformation to maintain the difference of internal and final round:

$$\begin{aligned}
\text{aese}(S, K) &= (\text{SubBytes} \circ \text{ShiftRows}(S \oplus K)) \\
\text{aesd}(S, K) &= (\text{InvSubBytes} \circ \text{InvShiftRows}(S \oplus K)) \\
\text{aesmc}(S) &= \text{MixColumns}(S) \\
\text{aesimc}(S) &= \text{InvMixColumns}(S)
\end{aligned}$$

The differences between ARM NEON and Intel AES-NI mostly are:

- AES-NI performs the XOR operation at the end of each round, while NEON executes the XOR operation at the beginning.
- AES-NI includes final round instructions that omit the *MixColumns* step, while NEON provides separate instructions for the *MixColumns* operation.

Consequently, to simulate an x86's AESENC instruction in ARM CPU, one more XOR instruction would be needed.

2.2 Analysis of the Pipeline, Parallelization and SIMD Throughput

When a CPU core processes instructions, it first fetches and decodes the instructions in the front-end, then routes them to different execution units in the back-end for computation. Performance is constrained by both the front-end decoding and the execution units in the back-end. Most modern processors feature a 4-wide or wider decode capability, which surpasses their throughput for AESR and XOR operations, where we use AESR to generalize the AES instructions in x86 and ARM platforms in a uniform way. Therefore, our focus in this work is on the back-end performance.

According to [Corb] and recent work [BBL+24], on latest x86 platforms, there are typically 3-4 execution units for different SIMD instructions, with usually 1-2 units dedicated to processing AESR instructions. In contrast, lightweight ARM processors, especially those designed for mobile devices, typically have only 2 SIMD units, with 1-2 of these supporting cryptographic instructions. This results in a weaker pipeline parallelism capability for XOR and AESR operations on ARM processors compared to their x86 counterparts. And for the high performance ARM processors for laptop and desktops, like Apple Silicon's Performance Core, there will be 4 SIMD units with fully support of AES.

To evaluate the performance of an instruction I, there are two main metrics:

- **Latency**: the number of clock cycles between the beginning to the return of its result, we marked as **Lat** below.
- **Throughput**: the number of instructions that can be processed in a given amount of time, we marked as **TP** below.

Pipeline Specifications. We summarize most popular ARM and x86 architectures and their AESR, XOR instructions latency and throughput as well as the pipeline unit occupied in Table 1. For x86 devices, we only remain the relevant units from $P_0 - P_6$, while for ARM devices, we category them to 2-SIMD, 3-SIMD, 4-SIMD and 6-SIMD platforms and number the SIMD unit as V_*. For Apple Silicon's Efficiency Cores, according to Apple Silicon CPU Optimization Guide [Inc24], ASIMD/FP units in E-Core can do 1 AES instructions per 2 cycles, so the throughput will be half of the number of execute units.

Conditions for Fully Utilizing the Pipeline. To fully utilize the pipeline, the ratio of AESR and XOR instructions must align with the pipeline's execution capacity. Let a represent the number of execution units capable of processing AESR instructions, b the total number of execution units, and $x : y$ the ratio of AESR to XOR instructions. Let C be a constant positive integer.

In order to fully utilize the pipeline, the following conditions must be satisfied:

- $x + y = C \cdot b$: This condition ensures that the total workload of SIMD instructions matches the pipeline's available execution capacity.
- $y : x \geq (b - a) : a$: This condition ensures that AESR instructions can run in parallel with XOR instructions. While most units handle XOR, only some support AESR. By substituting AESR with XOR, we maintain parallelism. However, too many AESR instructions can create contention, leaving XOR-only units underutilized.

We define the $a : b$ ratio as the AESR-SIMD ratio, which helps us better understand how different processors manage the balance of AESR and XOR instructions to fully utilize the SIMD pipeline.

2.3 Instruction Fusion on ARM Devices

Since some NEON instructions like `aese` and `aesmc` are often processed together, recent ARM architectures can accelerate certain instruction pairs in an operation called fusion. For example, instructions like `aesmc(aese(a, b))` and `eor(eor(a, b), c)` can benefit from this fusion to achieve higher IPC. These features should follow the optimization guide of each specific architecture, from Cortex-A77 (see Section 4.13 "Instruction Fusion" in [Com19b]) to later architectures.

3 The Specification of HiAE

Building on the analysis of AES and SIMD instructions across x86 and ARM platforms, we now turn to the specifications of the proposed authenticated encryption design. This section outlines the structural and operational details of the cipher, highlighting how the cross-platform optimizations discussed earlier are integrated into its design. In Section 4, we will delve deeper into the rationale behind these design choices, focusing on how they balance performance, security, and efficiency across diverse hardware architectures.

3.1 Notations

- $\mathrm{AESL}(x) = \mathrm{MixColumns} \circ \mathrm{SubBytes} \circ \mathrm{ShiftRows}(x)$
  Note: This differs from AESR, which is the generalized notation for the AES

Table 1: Comparison Pipeline of Different Architectures[AR19, Com22b, Com16, Com22a, Com23a, Com18, Com19a, Com19b, Com20, Com22c, Com23b, Com24b, Com21a, Com21b, Com22d, Com23c, Com22e, Com24d, Com22f, Com22g, Com24e, Com24c, Inc24]

Architecture	**Instructions**	**Lat**	**TP**	execution units					
x86 architecture processors				P_0	P_1	P_2	P_3	P_5	
Intel Haswell	AESENC	7	1					*	
	XOR	1	3	*	*			*	
Intel Skylake/Cascade Lake	AESENC	4	1	*					
	XOR	1	3	*	*			*	
Intel Ice Lake	AESENC	3	2	*	*				
	XOR	1	3	*	*			*	
Intel Alder Lake/Sapphire Rapids	AESENC	3	2	*	*				
	XOR	1	3	*	*			*	
AMD Zen 1/2/3/4/5	AESENC	4	2	*	*				
	XOR	1	4	*	*	*	*		
ARM architecture processors with 2 FP/SIMD units				V_0	V_1				
ARM Cortex-A55/A57	AESE/AESMC	3	1	*					
	XOR	3	2	*	*				
ARM Cortex-A510/A520	AESE/AESMC	3	2	*	*				
	XOR	3	2	*	*				
ARM Cortex-A75	AESE/AESMC	2	1	*					
	XOR	3	2	*	*				
ARM Cortex-A76	AESE/AESMC	2	1	*					
	XOR	2	2	*	*				
ARM Cortex-A77/A78	AESE/AESMC	2	2	*	*				
	XOR	2	2	*	*				
ARM Cortex-A715/A720/A725	AESE/AESMC	2	2	*	*				
	XOR	2	2	*	*				
ARM Neoverse-N2/N3	AESE/AESMC	2	2	*	*				
	XOR	2	2	*	*				
HiSilicon Taishan V110	AESE/AESMC	3	1	*					
	XOR	2	2	*	*				
ARM architecture processors with 3 FP/SIMD units				V_0	V_1	V_2			
Apple M1/M2 Generation E-Core	AESE/AESMC	5	1	*	*				
	XOR	2	2	*	*				
Apple M3/M3 Pro E-Core	AESE/AESMC	5	1	*	*				
	XOR	2	2	*	*				
Apple M3 Max E-Core	AESE/AESMC	5	1.5	*	*	*			
	XOR	2	3	*	*	*			
ARM architecture processors with 4 FP/SIMD units				V_0	V_1	V_2	V_3		
ARM Cortex-X1/X2/X3	AESE/AESMC	2	2	*	*				
	XOR	2	4	*	*	*	*		
ARM Cortex-X4	AESE/AESMC	2	4	*	*	*	*		
	XOR	2	4	*	*	*	*		
ARM Neoverse-V1/V2/V3	AESE/AESMC	2	4	*	*	*	*		
	XOR	2	4	*	*	*	*		
Apple M1/M2/M3 Generation P-Core	AESE/AESMC	3	4	*	*	*	*		
	XOR	2	4	*	*	*	*		
HiSilicon Taishan V120	AESE/AESMC	2	2	*	*				
	XOR	2	4	*	*	*	*		
ARM architecture processors with 6 FP/SIMD units				V_0	V_1	V_2	V_3	V_4	V_5
ARM Cortex-X925	AESE/AESMC	2	4	*	*		*	*	
	XOR	2	6	*	*	*	*	*	*

instruction applicable to both x86 and ARM platforms.

- $x \oplus y = \mathrm{XOR}(x, y)$
- S: The state of HiAE, which is composed of 16 blocks, i.e. $S = (S[0], S[1], ..., S[15])$, where $S[i] (0 \leq i \leq 15)$ are blocks and $S[0]$ is the first block. The i-th state block at round r is defined as $S^r[i]$.
- N: A 128-bit nonce
- AD_i: A 128-bit associated data block
- M_i: A 128-bit message block
- C_i: A 128-bit ciphertext
- $const_0$: A 128-bit constant, represented in hexadecimal as:

  `0x3243f6a8885a308d313198a2e0370734`
- $const_1$: A 128-bit constant, represented in hexadecimal as:

  `0x4a4093822299f31d0082efa98ec4e6c8`

  Remark: These two constants are derived from π in base 16.
- $X||0^*$: a 128-bit string of the concatenation of X and the complement of zeros.
- $|M|$ or $\mathrm{len}(M)$: the length in bit of the string M.

3.2 The Shift State Update Function

The update function of HiAE operates on the 16-block state S and a 128-bit data block X, producing a new state $S_{New} = UpdateFunction(S, X)$ according to the following process:

$$\begin{aligned}
S_{New}[15] &= AESL(S[0] \oplus S[1]) \oplus AESL(S[13]) \oplus X \\
S_{New}[14] &= S[15] \\
S_{New}[13] &= S[14] \\
S_{New}[12] &= S[13] \oplus X \\
S_{New}[11] &= S[12] \\
S_{New}[10] &= S[11] \\
S_{New}[9] &= S[10] \\
S_{New}[8] &= S[9] \\
S_{New}[7] &= S[8] \\
S_{New}[6] &= S[7] \\
S_{New}[5] &= S[6] \\
S_{New}[4] &= S[5] \\
S_{New}[3] &= S[4] \\
S_{New}[2] &= S[3] \oplus X \\
S_{New}[1] &= S[2] \\
S_{New}[0] &= S[1]
\end{aligned}$$

3.3 Specification of HiAE

HiAE is an authenticated encryption scheme with associated data specifically designed for ARM architecture, structured into four phases: initialization, processing of associated

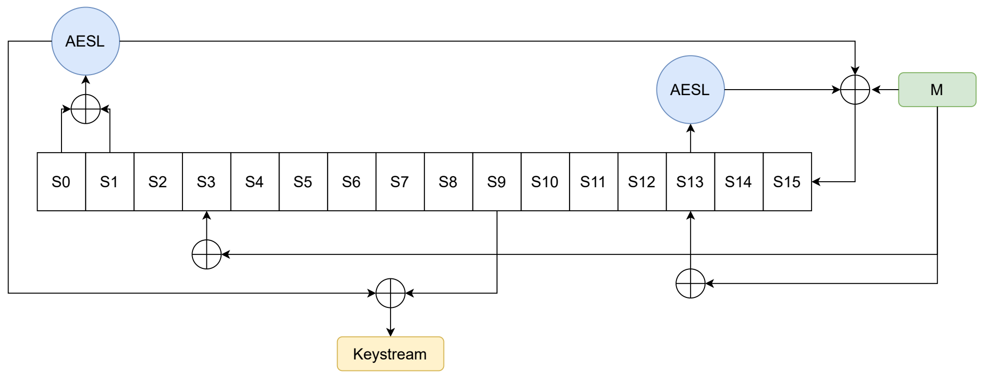

Figure 1: Overall schematic of HiAE

data, encryption, and finalization. It takes as input a 256-bit key $K = K_0||K_1$, $K_i \in F_2^{128}$, a 128-bit nonce $N \in F_2^{128}$, the associated data AD, and the message M. The output includes the ciphertext C where $|C| = |M|$ and a 128-bit tag T. Initially, the associated data AD and the message M are padded with 0 to ensure their lengths are multiples of 128, i.e, $AD||0^* = AD_0||...AD_{|AD|/128-1}$ and $M||0^* = M_0||...M_{|M|/128-1}$, where AD_i and M_i are 128-bit blocks. We describe the encryption and authentication process below, as illustrated in Figure 2 and Algorithm 1.

Initialization. The initialization of HiAE involves loading the key $K_0||K_1$ and the nonce N into the state, then executing 32 steps of the UpdateFunction with a constant used as the message, i.e., $S = UpdateFunction(S, const_0)$. Initially, the state is loaded as follow:

$$S^{-32}[0] = const_0, S^{-32}[1] = K_1, S^{-32}[2] = N, S^{-32}[3] = const_0,$$

$$S^{-32}[4] = 0, S^{-32}[5] = N \oplus K_0, S^{-32}[6] = 0, S^{-32}[7] = const_1,$$

$$S^{-32}[8] = N \oplus K_1, S^{-32}[9] = 0, S^{-32}[10] = K_1, S^{-32}[11] = const_0,$$

$$S^{-32}[12] = const_1, S^{-32}[13] = K_1, S^{-32}[14] = 0, S^{-32}[15] = const_0 \oplus const_1.$$

After 32 update rounds, two 128-bit keys are XORed with the state once more:

$$S^0[9] = S^0[9] \oplus K_0,$$

$$S^0[13] = S^0[13] \oplus K_1.$$

Processing the associated data. Following initialization, the associated data AD is used to update the state.
For $i = 0$ to $|AD|/128 - 1$:

$$S = UpdateFunction(S, AD_i)$$

This phase is skipped if the associated data is empty.

Encryption. After processing the associated data, at each step of the encryption, a 128-bit message block is used to update the state, and M_i is then encrypted to produce C_i. The last block of ciphertext is then truncated so that the length of the ciphertext is equal to the length of the message. The encryption phase is skipped if the message is empty.
For $i = 0$ to $|M|/128 - 1$:

$$C_i = AESL(S[0] \oplus S[1]) \oplus S[9] \oplus M_i$$

$$S = UpdateFunction(S, M_i)$$

Finalization. After encrypting all the message blocks, the authentication tag is generated through 32 update rounds. The lengths of the associated data and the message are used to update the state as:

$$S = UpdateFunction(S, len(AD)||len(M))$$

and the tag is computed as:

$$T = \bigoplus_{i=0}^{15} S[i].$$

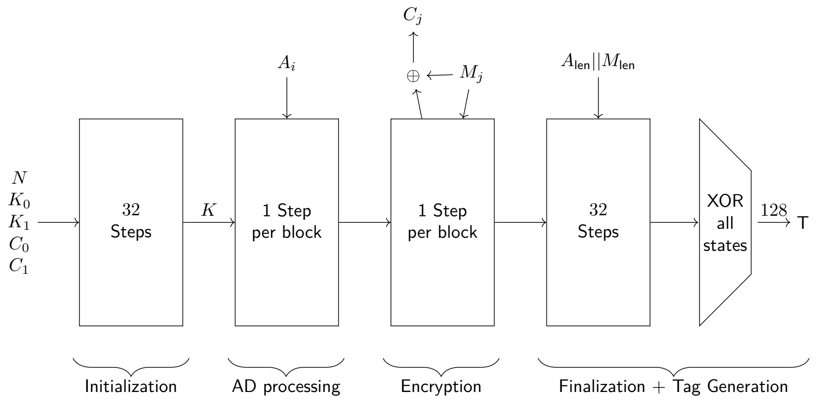

Figure 2: General process of HiAE.

3.4 Security Goal

HiAE is a nonce-based authenticated encryption scheme, whose security relies on the nonce-respecting setting. Specifically, each key and IV pair must be used to secure only one message. If verification fails, the scheme should not output the decrypted ciphertext or a new authentication tag.

We limit the maximum message length to less than 2^{64} bits, which aligns with modern cryptographic standards (e.g., NIST LWC [oSN25] $2^{50}-1$ bytes, NIST SP 800-38D [oSN07], $2^{64}-1$ bits) and design practices, such as those used in AEGIS (which also uses a maximum of 2^{64} bits). Additionally, a single key may be used to protect at most 2^{64} messages. These upper bounds are a precautionary measure designed to ensure the scheme remains secure against practical attacks and are not an absolute security requirement.

The design targets 256-bit security against key recovery and state recovery attacks, along with 128-bit security for integrity against forgery attempts. It is important to note that the encryption security assumes the attacker cannot successfully forge messages through repeated trials.

While attacks that exploit keystream bias may have a time complexity ranging between 2^{150} and 2^{256}, their practical impact on overall security is minimal. Furthermore, these attacks are not accelerated by quantum computing, and their complexity makes them infeasible in real-world scenarios (see Section 5.3 for a more detailed discussion). Consequently, we have not set the security goal for resisting such attacks at 2^{256}, reflecting a focus on balancing efficiency with robust security.

Finally, to address the threat posed by quantum computing, HiAE targets a security strength of 128 bits against key recovery attacks and forgery attacks in quantum setting.

Algorithm 1 The specification of HiAE

```
procedure HiAE_Encrypt($K_0, K_1, N, AD, M$)
    $S \leftarrow$ Initialization($N, K_0, K_1$)
    if $|AD| > 0$ then
        $S \leftarrow$ ProcessAD($S, AD$)
    end if
    if $|M| > 0$ then
        $S \leftarrow$ Encryption($S, M, C$)
        Truncate $C$
    end if
    $T \leftarrow$ Finalization($S, len(AD), len(M)$)
    return $(C, T)$
end procedure
procedure HiAE_Decrypt($K_0, K_1, N, AD, C, T$)
    $S \leftarrow$ Initialization($N, K_0, K_1$)
    if $|AD| > 0$ then
        $S \leftarrow$ ProcessAD($S, AD$)
    end if
    if $|C| > 0$ then
        $S \leftarrow$ Decryption($S, C, M$)
        Truncate $M$
    end if
    if $T =$ Finalization($S, len(AD), len(M)$) then
        return $M$
    else
        return $\perp$
    end if
end procedure
procedure Initialization($N, K_0, K_1$)
    $(S[0], S[1], S[2], S[3]) \leftarrow (const_0, K_1, N, const_0)$
    $(S[4], S[5], S[6], S[7]) \leftarrow (0, N \oplus K_0, 0, const_1)$
    $(S[8], S[9], S[10], S[11]) \leftarrow (N \oplus K_1, 0, K_1, const_0)$
    $(S[12], S[13], S[14], S[15]) \leftarrow (const_1, K_1, 0, const_0 \oplus const_1)$
    for $i = 0$ to $31$ do
        $S \leftarrow UpdateFunction(S, const_0)$
        $(S[9], S[13]) \leftarrow (S[9] \oplus K_0, S[13] \oplus K_1)$
    end for
    return $S$
end procedure
procedure ProcessAD($S, AD$)
    $d \leftarrow |AD|$
    for $i = 0$ to $d - 1$ do
        $S \leftarrow UpdateFunction(S, AD_i)$
    end for
    return $S$
end procedure
procedure Encryption($S, M, C$)
    $m \leftarrow |M|$
    for $i = 0$ to $m - 1$ do
        $C_i \leftarrow \text{AESL}(S[0] \oplus S[1]) \oplus S[9] \oplus M_i$
        $S \leftarrow UpdateFunction(S, M_i)$
    end for
    return $S$
end procedure
procedure Decryption($S, M, C$)
    $c \leftarrow |C|$
    for $i = 0$ to $c - 1$ do
        $M_i \leftarrow \text{AESL}(S[0] \oplus S[1]) \oplus S[9] \oplus C_i$
        $S \leftarrow UpdateFunction(S, M_i)$
    end for
    return $S$
end procedure
procedure Finalization($S, len(AD), len(M)$)
    for $i = 0$ to $31$ do
        $S \leftarrow UpdateFunction(S, (len(AD)||len(M))$
    end for
    $T \leftarrow 0$
    for $i = 0$ to $15$ do
        $T \leftarrow T \oplus S[i]$
    end for
    return $T$
end procedure
```

We do not claim security against online superposition queries to the cryptographic oracle attacks, as such attacks are highly impractical in real-world applications. The details of the security analysis, including the evaluation of key-committing attacks, are discussed in Section 5.

4 Design Rationale

4.1 Cross-Platform Efficient Structure

As introduced in Section 2, SIMD instructions exhibit different implementations and performance across processors. Recent AES-based cipher designs usually focus on x86 architecture, particularly using the **aesenc** instruction, differs significantly on ARM, where achieving the same single-round functionality typically requires three instructions instead of two. Conversely, implementing an ARM-based single-round function on x86 also involves one additional XOR instruction. For example:

$$\mathbf{aesenc}(x, y) = \mathbf{xor}(\mathbf{aesmc}(\mathbf{aese}(x, 0)), y),$$

$$\mathbf{aesmc}(\mathbf{aese}(x, y)) = \mathbf{aesenc}(\mathbf{xor}(x, y), 0).$$

Both approaches incur an additional XOR operation, which increases the latency and decreases overall efficiency (We provided the Rocca initialization assembly code on ARM and x86 in Figure B4, which demonstrates the inefficiency of Rocca and other x86-based ciphers using the `aesenc` instruction with conversion from x86 to ARM architectures. The ARM's implementation cost much more instructions than x86).

We initially attempt to avoid wasting XOR operations across architectures by using a structure with XOR operation both before and after the AES round function, like $F(a, b, c) = AESL(a \oplus b) \oplus c$. However, this structure results in a ratio of **aesenc:xor = 1:1** on x86 and **aese/aesmc:xor = 2:1** on ARM, which limits pipeline utilization for many processors with limited ratio of AES SIMD units. As discussed in Section 2, most x86 processors have an AESR-SIMD ratio of 1:3 and 2:3, while most ARM processors have an AESR-SIMD ratio of 1:2 and 1:1. Hence, this structure does not offer optimal ratios for 1:3 x86 and 1:2 ARM platforms.

To find a better structure, we begin with the optimal ratio for 1:3 x86 and 1:2 ARM platforms, which is **aesenc:xor = 1:2** on x86 and **aese/aesmc:xor = 1:1** on ARM, typically 1 AES round with 2 additional XOR. We test the initial idea of AES round as well as 2 additional XOR combinations below to evaluate the structure's efficiency across platforms. The results of the IPC (Instructions Per Cycle) and cycle per operation are shown in Table 2 where the structures are:

$$XAX(a, b, c) = AESL(a \oplus b) \oplus c$$

$$XAXX(a, b, c, d) = AESL(a \oplus b) \oplus c \oplus d$$

$$XXAX(a, b, c, d) = AESL(a \oplus b \oplus c) \oplus d$$

The test results show that for devices with a 2:3 ratio, the 'XAX' structure already fulfills the pipeline, as the cycle cost aligns with the instruction count and the IPC can reach the full capacity of the available SIMD units. However, for 1:3 devices, the bottleneck is the single AES unit while the other units—capable only of processing XOR—remain idle. In this scenario, the additional XOR operations do not increase the cycle cost and instead better utilize the pipeline, making the 'XAXX' and 'XXAX' structures equivalent in cycle cost. The most notable difference is observed on ARM devices. As discussed in Section 2.3, the 'XAXX' structure better exploits ARM's AES and XOR fusion, which unlocks much higher IPC and reduces cost compared to 'XAX'. Based on these observations, we propose

Table 2: Different Structure's IPC and cycle/op across architecture. As introduced in Section 2.3, some ARM processors feature 'instruction fusion,' which can process certain instruction pairs faster, such as aese+aesmc or two eor operations acting as a 3-operand XOR.)

Architecture and SIMD configs	Metrics	Structure		
		XAX	XAXX	XXAX
ARM, Taishan V110, 1:2	IPC	2.08	**3.00**	2.22
	cycle/op	1.46	**1.33**	1.81
ARM, Taishan V120, 2:4	IPC	3.03	**4.49**	2.98
	cycle/op	0.99	**0.88**	1.34

that the optimal cross-platform efficient structure is 'XAXX', which can be implemented as:

$$\textbf{x86: XAXX}(a,b,c,d) = \textbf{xor}(\textbf{aesenc}((\textbf{xor}(a,b),c),d))$$
$$\textbf{ARM: XAXX}(a,b,c,d) = \textbf{xor}(\textbf{xor}(\textbf{aesmc}(\textbf{aese}(a,b)),c),d).$$

4.2 On the Design of Update Function

After initially designing our structure with the lowest possible rate, a higher block count, and an optimal two AES rounds per encryption, we need to select several parameters of the structure to meet the security requirements outlined in Section 3.4. These include the number of inserted message blocks, the positions for injecting messages into the state, and the format of the keystream. As analyzed by Jean and Nikolic in [JN16], the most efficient structure has a rate between the number of $AESR$ and number of inserted messages of 2 with 12 state blocks, 4 **aesenc**, and 2 inserted messages. However, reducing the number of AES rounds to optimize compatibility with ARM architecture is one of our strategies. To achieve this, we develop the search strategies according to the following approach.

- Extend the candidate pool by increasing the number of state blocks from 13 to 16, while Jean and Nikolic limited their model to a maximum of 12 internal blocks. This approach introduces minimal overhead (or even no additional cost) on ARM architecture, thanks to its ample register capacity of up to 32.
- Limit the number of **aesenc/(aese+aesmc)**s in encryption to fewer than 4 and process only one message per encryption to maintain a rate of $AESL$ per processed message of at least 2. To enhance security while preserving efficiency, the message is XORed with multiple internal blocks to introduce more differences in the forgery attack model.

To align with the cross-platform structure outlined in Sect. 4.1, we constrain the structure of the new leftmost state block to the form $AESL(S[0] \oplus S[i]) \oplus M \oplus d$, where d can be either S_j or $AESL(S_j)$. Additionally, after updating, the structure of certain middle state blocks takes the form $c \oplus M$, where c may be S_m or $AESL(S_m)$, while keeping the total of AES in used is lower than 4.

After modeling the structure of our target design with the parameters in Table 3 using MILP, we found that only the pool of Candidates-2 meets the security requirement, ensuring at least 22 active S-boxes in the differential trails against forgery attacks. Also, updating the leftmost state block with XORing of two AESR instructions' output can let the 2 AESR instruction parallel without data dependency on 2 AESR instructions, which usually have long latency. Therefore, we chose our HiAE design with a 16-block internal

Table 3: Parameters and candidates of round functions. # IM is the number of state blocks to insert message.

Round function	# **aesenc**	**aesenc** at	# blocks	# IM	# candidates	#searched candidates
Candidates-1	2	d	13-16	2	5389	all
Candidates-2	2	d	13-16	3	34401	all
Candidates-3	2	c	13-16	2	5389	all
Candidates-4	2	c	13-16	3	68802	all

state, which offers highest balancing between performance for keystream generation and security, achieving at least 26 active S-boxes, and implemented the UpdateFunction as illustrated in Fig 1.

4.3 On the Loading of Nonce and Key

To avoid the cost of building a new circuit for processing the initialization states, we utilize the update function to diffuse the initial state. The nonce and the key are injected into five initial state blocks in the forms $K_0, K_1, N, N \oplus K_0$, and $N \oplus K_1$, while the remaining blocks are set to zero or constants. To ensure the security of the initialization phase against differential attacks, we generate a Mixed-Integer Linear Programming (MILP) model to determine the optimal block positions for $K_0, K_1, N, N \oplus K_0$, and $N \oplus K_1$. This ensures that the number of active S-boxes in the differential characteristic after 22 update function rounds exceeds 43. In our design, the initial state is arranged as follows:

$$S^{-32}[0] = const_0, S^{-32}[1] = K_1, S^{-32}[2] = N, S^{-32}[3] = const_0,$$

$$S^{-32}[4] = 0, S^{-32}[5] = N \oplus K_0, S^{-32}[6] = 0, S^{-32}[7] = const_1,$$

$$S^{-32}[8] = N \oplus K_1, S^{-32}[9] = 0, S^{-32}[10] = K_1, S^{-32}[11] = const_0,$$

$$S^{-32}[12] = const_1, S^{-32}[13] = K_1, S^{-32}[14] = 0, S^{-32}[15] = const_0 \oplus const_1.$$

Next, the initial state is updated through 32 rounds of the shift-state update function, using $S = UpdateFunction(S, const_0)$. Finally, two states $S^0[9]$ and $S^0[13]$, which are involved in the first step of the decryption, are XORed with the key to prevent key recovery attacks from state recovery. This method of protection has been analyzed for Rocca in [HII+22], stated as follows:

$$S^0[9] = S^0[9] \oplus K_0,$$

$$S^0[13] = S^0[13] \oplus K_1.$$

Since decryption begins by recovering the last state of the previous step as follows:

$$S^{Old}[15] = C \oplus S[9] \oplus AESL(S[13]),$$

the attackers cannot invert the initialization phase without knowing the key, even if the state after this phase is fully exposed.

After 16 update rounds, the nonce becomes integrated into the expressions of all state blocks, thereby increasing the difficulty of key-recovery attacks on the round-reduced initialization in [HII+22]. To provide evidence that all internal states are well diffused, meaning that all state blocks are expressed in terms of the nonce and the key after the initialization phase, we present the simplified expressions of the first eight state blocks after 18 rounds, ignoring any constants:

$$S^{-14}[0] = A(A(K_0 \oplus N))$$

$$S^{-14}[1] = A(N \oplus K_0) \oplus A(A(N))$$

$$\begin{aligned}
S^{-14}[2] &= A(N \oplus K_0) \oplus A(A(N \oplus K_0) \oplus A(A(N))) \\
S^{-14}[3] &= A(N \oplus K_1) \oplus A(A(N \oplus K_0) \oplus A(A(A(K_0 \oplus N))) \\
S^{-14}[4] &= A(N \oplus K_1) \oplus A(A(N \oplus K_0) \oplus A(A(K_0 \oplus N))) \\
S^{-14}[5] &= A(N \oplus K_1) \oplus A(A(A(N \oplus K_0) \oplus A(A(N)))) \\
S^{-14}[6] &= A(K_1) \oplus A(A(N \oplus K_1) \oplus A(A(N \oplus K_0) \oplus A(A(A(K_0 \oplus N))))) \\
S^{-14}[7] &= A(K_1) \oplus A(A(N \oplus K_1) \oplus A(A(N \oplus K_0) \oplus A(A(N))))).
\end{aligned}$$

The remaining 8 states are even more complex in terms of both the key and nonce.

4.4 On the Design of Keystream Generation and Intermediate Value Sharing from State Update Function

As our target to achieve high throughput, we try to reuse the intermediate value in the state update function to reduce the cost of the AES instruction. Among the candidates that remain after security filtering, we proceed to identify the positions for selecting $S[k]$ to form the keystream in the format $AESL(S[0] \oplus S[i]) \oplus S[k]$ with balancing efficiency of performance and security, where $AESL(S[0] \oplus S[i])$ an is shared from the state update function as discussed in Section 4.2. Among the candidates, $8 \leq k \leq 12$ providing enough security. We notice that in decryption phase, the $S[j]$ which inject with message in update function is close to $S[k]$ candidate and the message block should be calculated by keystream, which might cause a data dependency and hurt the pipeline parallel. We evaluated decryption performance on the Kunpeng-920 ARM chip. As shown in Table 4, setting $k = 9$ ensures sufficient parallelism when $j = 13$, resulting in $j - k = 4$ rounds of dependency-free parallelism during decryption. Therefore, we select state 9 for keystream generation.

Table 4: IPC for different $S[k]$ position on Keystream cross architectures on decryption

$k =$	8	9	10	11	12
ARM (Taishan V110)	2.30	**2.32**	2.32	2.28	1.89

As the structure above, we can share intermediate value between Keystream generation and state update function to reduce one AES instruction and one XOR instruction costs in both encryption phase and decryption phase and only use 2 AESL instructions each round. For encryption:

$$\begin{aligned}
S^{New}[15] =& AESL(S[0] \oplus S[1]) \oplus M \oplus AESL(S[13]) \\
C =& AESL(S[0] \oplus S[1]) \oplus M \oplus S[9] \\
\text{Shared Value :}& AESL(S[0] \oplus S[1]) \oplus M
\end{aligned}$$

For decryption:

$$\begin{aligned}
S^{New}[15] =& C \oplus S[9] \oplus AESL(S[13]) \\
M =& C \oplus S[9] \oplus AESL(S[0] \oplus S[1]) \\
\text{Shared Value :}& C \oplus S[9]
\end{aligned}$$

By leveraging value sharing, the one-round encryption instruction cost becomes `aesenc:xor` = 2:4 on x86 and `aese/aesmc:xor` = 4:5 on ARM, while maintaining a high throughput ratio. For a clearer comparison, we summarize recent work on AES-based ciphers and their corresponding ratios in Table 5, and benchmark these ciphers on 16KB x86 with `perf-tools` to compare the cycle and instruction cost cross-platform in Table 6. As discussed in Section 2, the HiAE's AESR-SIMD ratio on x86 devices is 1:3, which enables efficient pipeline utilization. The 4:5 ratio, with an additional XOR that can be fused as described in Section 2.3, also achieves high throughput on modern ARM devices. The Table 6 also demonstrates the cross-platform efficiency as Section 2.1 discussed that recent AES-based ciphers approximately double the instruction cost on ARM than x86 for the `aesenc` convert, while HiAE cost much fewer instructions on ARM.

Table 5: Comparison of AESR-SIMD ratio on different Ciphers (per 128-bit block)

Architecture	x86			ARM		
Cipher	aesenc	xor/and	ratio	aese/aesmc	xor/and	ratio
HiAE(this work)	2	4	1:3.00	4	5	1:2.25
Rocca	3	3.5	1:2.16	6	6.5	1:2.08
Rocca-S	4	2.5	1:1.62	8	6.5	1:1.81
AEGIS-128L	4	5	1:2.25	8	9	1:2.12

Table 6: IPC Benchmark for AES-based Ciphers on 16KB data Encryption

Architecture	ARM, Taishan V120		
Cipher	cycle/byte	ins/byte	IPC
HiAE(this work)	**0.260**	**0.884**	3.41
Rocca	0.413	1.383	3.35
Rocca-S	0.466	1.480	3.17
AEGIS-128L	0.479	1.810	**3.77**

4.5 Design Rationale for Using a 128-bit Tag

The impact of 128-bit tag length on authentication security has been a topic of consideration in the design of AEGIS [WP14], and was further emphasized in the analysis of `Rocca-S` [HII$^+$22]. While these works highlight the potential vulnerabilities associated with shorter tags, we have opted for a 128-bit tag in our design after careful consideration of real-world application constraints. This decision is motivated by the desire to balance security with practical efficiency.

While longer tags provide enhanced security, they also introduce additional overhead in terms of data transmission and storage. This increased burden could pose challenges for applications with resource limitations or high throughput demands. We believe that a 128-bit tag offers adequate security for the following reasons:

- **Forgery attacks must be conducted online.** Unlike offline attacks, where attackers can use parallelization, precomputation, or time-memory trade-offs, online attacks require performing individual authentication queries. This makes the process inherently slower and more constrained by the need to handle each query sequentially. In rare cases, multiple users might share the same key, allowing parallel submissions of queries to the server. However, such situations are extremely uncommon, and even if attackers could impersonate multiple users, submitting a large volume of queries would likely trigger the server's denial-of-service (DoS) defenses. As a result, we expect that the opportunities for attackers to accelerate forgery attempts through parallelism will be highly limited in practice.
- **128-bit authentication is resistant to foreseeable attacks.** The expected time to break 128-bit authentication is approximately 1.08×10^{22} years, assuming

each forgery attempt takes 1 nanosecond. This ensures that, within any realistic timeframe, 128-bit authentication remains unbreakable and provides robust security.

- **Even with extraordinary computational power, online attacks remain impractical.** For example, with 2^{250} computational power, an attacker would still need to conduct online queries for each forgery attempt. Without the ability to recover the key, this process remains infeasible and impractical.

In summary, while we acknowledge the concerns raised in the AEGIS and `Rocca-S` analyses regarding short tags, we maintain that a 128-bit tag provides robust security with minimal overhead on the data transmission and storage. This decision reflects our commitment to ensuring both the security and efficiency of our design, making it suitable for practical, real-world applications.

5 Security Analysis

Security has always been a primary concern for newly designed stream ciphers. In this section, we assess the resistance of HiAE to various cryptanalysis attacks.

5.1 Differential Attack on Initialization

In the initialization phase, the primary concern lies with the differential attack, which differences are injected in key. The security bound of this attack can be estimated bases on the minimum number of active S-boxes during initialization. To mitigate risks, it's crucial to determine the number of update rounds and strategically select state positions for injecting the nonce and key to ensure that a minimum of 43 active S-boxes, leading to a differential characteristic of at least $2^{-6\times 43} = 2^{-258}$, are generated after the initialization phase.

Table 7: The lower bound for the number of active S-boxes in the initialization phase where S_{sk} and S_{rk} mean active S-boxes in the single-key and related-key setting, respectively.

Rounds	12	13	14	15	16	17	18	19	20	21	22	23	24
No. of S_{sk}	27	28	28	28	28	28	29	29	30	35	35	40	52
No. of S_{rk}	8	12	16	19	23	27	37	37	53	58	58	61	68

5.2 Forgery Attack

5.2.1 Forgery Attack from Internal State

In a forgery attack, the goal is to identify a differential trail that allows attackers to arbitrarily choose differences in the associated data or message, with the expectation that these differences will lead to a collision in the internal state after several rounds. Therefore, these data or messages with the same length will lead to the same tag value after the finalization phase. The resistance to this type of attack can be effectively assessed using an automated approach with modellizing the cryptographic operators by MILP language [MWGP11]. Since HiAE is built upon the AES round function, demonstrating that the differential trail includes more than 22 active S-boxes is sufficient, given the tag length of 128 bits. With a MILP-based method, a minimum of 26 active S-boxes is determined, ensuring that HiAE offers 128-bit security against forgery attacks. More specify, the lower bound for number of active S-boxes of HiAE by injecting the difference up to these corresponding rounds are listed in Table 8.

Table 8: The lower bound for the number of active S-boxes to build forgery attack

Rounds	17	18	25	30	35	45
No. of S-boxes	48	32	30	26	26	26

5.2.2 Forgery Attack from Reduced Round Finalization

This attack explore the strength of finalization phase to agaisnt the forgery attack raised by the difference in associated data and message length when using the same associated data and message value but with different length. This due to using zero-padding to complete the block, i.e AD and $AD' = AD||0^*$, as well as M and $M' = M||0^*$ will result in identical states ater processing the associated data and the encryption steps when AD, AD' and M, M' occupy the same number of blocks, but $|AD||||M|$ and $|AD'||||M'|$ will have difference. We introduce a low Hamming weight difference to $\Delta|AD|$ or $\Delta|M|$ (e.g., `0x00000000000000010000000000000000`) and propagate it through as many rounds as possible until the probability of the differential trail drops below 2^{-128}. By modeling this attack using MILP, we find that after 10 rounds, the number of active S-boxes exceeds 22. Thus, we conclude that 32 rounds in the finalization phase provide sufficient security against this attack.

5.3 The Linear Bias on Keystream

To analyze the strength of our construction against a potential statistical attack, we also used the simplified truncated model referenced in [ENP19]. By examining 16 consecutive rounds, we found that at least 19 active S-boxes appears in the linear trail, which corresponds to a distinguishing attack with a time complexity of no less than 2^{114}. Although aiming for 22 active S-boxes would be preferable, it might need to compromise the performance, and we believe that 19 active S-boxes offer adequate protection against such attacks due to the significant disparity between the truncated and bitwise models highlighted in [ENP19]. Additionally, to determine if there exists a compatible linear trail aligning with the truncated model's best solution, we implemented the bitwise model, which removes additional constraints on S-box input and output masks, aside from trivial infeasibilities caused by zero input or output masks. Most bitwise transitions in the SubByte operator involve multiple active bits, rather than achieving the optimal square correlation transition of 2^{-6}, which occurs when only one bit is active in both the input and output masks. However, efforts to validate these transitions quickly lead to invalid linear characteristics of less than 25 active S-boxes. This observation which is more discussed in Appendix A suggests that the optimal square correlation of HiAE is significantly weaker than the value of 2^{-150}.

From a practical application perspective, we consider a square correlation of 2^{-150} to be an acceptable threshold for information leakage. In fact, for an attacker to extract even a single bit of information from a message, they would need to encrypt at least 2^{150} identical message blocks with independent key and nonce pairs, a scenario that is infeasible in real-world attacks. The same amount of data and computational complexity would be required to launch a distinguishing attack on the keystream using this linear approximation. Given these requirements, such an attack is also negligible in practical applications. Also, due to the large internal state size of 2048 bits, it is infeasible to adapt the classical linear bias attack on an LFSR stream cipher into a quantum linear bias attack with a lower complexity than the key-recovery attack achievable through Grover's algorithm, as analyzed in [Hos24].

5.4 State-recovery Attack

In a state-recovery attack, the attacker can query message-ciphertext pairs to gather information about the sequence of keystreams, aiming to eventually recover the internal state. Since each keystream contains 128 bits of information, the attacker would need at least 16 consecutive rounds of message queries to fully reconstruct the 16-block internal state. By assuming all messages are zero, the attacker gains knowledge of the subsequent sequence of keystreams:

$$
\begin{aligned}
&C_0 = A(S[0] \oplus S[1]) \oplus S[9]\\
&C_1 = A(S[1] \oplus S[2]) \oplus S[10]\\
&C_2 = A(S[2] \oplus S[3]) \oplus S[11]\\
&C_3 = A(S[3] \oplus S[4]) \oplus S[12]\\
&C_4 = A(S[4] \oplus S[5]) \oplus S[13]\\
&C_5 = A(S[5] \oplus S[6]) \oplus S[14]\\
&C_6 = A(S[6] \oplus S[7]) \oplus S[15]\\
&C_7 = A(S[7] \oplus S[8]) \oplus A(S[0] \oplus S[1]) \oplus A(S[13])\\
&C_8 = A(S[8] \oplus S[9]) \oplus A(S[1] \oplus S[2]) \oplus A(S[14])\\
&C_9 = A(S[9] \oplus S[10]) \oplus A(S[2] \oplus S[3]) \oplus A(S[15])\\
&C_{10} = A(S[10] \oplus S[11]) \oplus A(S[3] \oplus S[4]) \oplus A(A(S[0] \oplus S[1]) \oplus A(S[13]))\\
&C_{11} = A(S[11] \oplus S[12]) \oplus A(S[4] \oplus S[5]) \oplus A(A(S[1] \oplus S[2]) \oplus A(S[14]))\\
&C_{12} = A(S[12] \oplus S[13]) \oplus A(S[5] \oplus S[6]) \oplus A(A(S[2] \oplus S[3]) \oplus A(S[15]))\\
&C_{13} = A(S[13] \oplus S[14]) \oplus A(S[6] \oplus S[7]) \oplus A(A(S[3] \oplus S[4]) \oplus A(A(S[0] \oplus S[1]) \oplus A(S[13])))\\
&C_{14} = A(S[14] \oplus S[15]) \oplus A(S[7] \oplus S[8]) \oplus A(A(S[4] \oplus S[5]) \oplus A(A(S[1] \oplus S[2]) \oplus A(S[14])))\\
&C_{15} = A(S[15] \oplus A(S[0] \oplus S[1]) \oplus A(S[13])) \oplus A(S[8] \oplus S[9])\\
&\qquad \oplus A(A(S[5] \oplus S[6]) \oplus A(A(S[2] \oplus S[3]) \oplus A(S[15])))
\end{aligned}
$$

In the above system of equations, C_i values are known to the attacker, while S_i represents the original internal block states that need to be uncovered. However, we will show that by using the guess-and-determine method, the system of equations can not be solved after 2^{256} time complexity.

To avoid passing the time complexity of 2^{256}, attackers should not fully guess two state blocks. If the attacker guess a fully block of S_i, for i is one of the index in $\{9, 10, 11, 12, 13, 14, 15\}$, the information of $S_j \oplus S_{j+1}$, for respective j is one of number in $\{0, 1, 2, 3, 4, 5, 6\}$ is revealed by calculating the inversion of AES round value. However, from C_7 to C_{15}, at least five state blocks contribute to the keystream computations. As a result, having access to at most 256 bits of information is insufficient to uncover the remaining state blocks without also guessing the entirety of another state block. Therefore, we estimate that the time complexity of the guess-and-determine attack cannot be reduced below 2^{256}.

5.5 Integral Attack

Integral attacks are among the most effective methods for targeting round-reduced AES since the integral distinguisher for 4-round is first analyzed in [LIMS21]. To assess the security of our design, evaluating its resistance to integral attacks is crucial. Following the approach in [LIMS21], we begin by expressing the internal state in terms of the initial state, allowing for an analysis of these expressions. For clarity, let $S^r[i]$ represent the i^{th}-block state at the r iterations of the round function during the initialization phase. Additionally, we simplify the notation by omitting constants and using $A(S)$ to denote that S undergoes one AES round, where the constant is ignored and $A(S)$ could represent $A(X \oplus c)$ with c as a 128-bit constant. Using this approach, the internal state of first 11 rounds can be represented as follows:

$S^0[2] = N, S^0[5] = N, S^0[8] = N,$
$S^1[1] = N, S^1[4] = N, S^1[7] = N,$
$S^2[0] = N, S^2[3] = N, S^2[6] = N, S^2[15] = A(N),$
$S^3[2] = N, S^3[5] = N, S^3[14] = A(N), S^3[15] = A(N),$
$S^4[1] = N, S^4[4] = N, S^4[13] = A(N), S^4[14] = A(N),$
$S^5[0] = N, S^5[3] = N, S^5[12] = A(N), S^5[13] = A(N), S^5[15] = A(N) \oplus A(A(N)),$
$S^6[2] = N, S^6[11] = A(N), S^6[12] = A(N), S^6[14] = A(N) \oplus A(A(N)), S^6[15] = A(N) \oplus A(A(N)),$
$S^7[1] = N, S^7[10] = A(N), S^7[11] = A(N), S^7[13] = A(N) \oplus A(A(N)), S^7[14] = A(N) \oplus A(A(N)),$
$S^8[0] = N, S^8[9] = A(N), S^8[10] = A(N), S^8[12] = A(N) \oplus A(A(N)), S^8[13] = A(N) \oplus A(A(N)),$
$S^8[15] = A(N) \oplus A(A(N) \oplus A(A(N))),$
$S^9[8] = A(N), S^9[9] = A(N), S^9[11] = A(N) \oplus A(A(N)), S^9[12] = A(N) \oplus A(A(N)),$
$S^9[14] = A(N) \oplus A(A(N) \oplus A(A(N))), S^9[15] = A(N) \oplus A(A(N) \oplus A(A(N) \oplus A(A(N)))),$
$S^{10}[7] = A(N), S^{10}[8] = A(N), S^{10}[10] = A(N) \oplus A(A(N)), S^{10}[11] = A(N) \oplus A(A(N)),$
$S^{10}[13] = A(N) \oplus A(A(N) \oplus A(A(N))), S^{10}[14] = A(N) \oplus A(A(N) \oplus A(A(N) \oplus A(A(N)))),$
$S^{11}[15] = A(A(N) \oplus A(A(N) \oplus A(A(N)))),$
$S^{14}[15] = A(A(A(N) \oplus A(A(N) \oplus A(A(N))))).$

It is essential to consider the scenario where the nonce N takes on all possible 2^{128} values while using the same 256-bit key to guarantee the 256-bit security of our design. The most efficient integral attack result for round-reduced AES indicates that no integral distinguisher exists for 5 or more AES rounds [SLR+15]. Based on our analysis, after 14 rounds, the nonce N goes through at least 5 AES rounds in $S^{14}[15]$, thus preventing integral attacks. In summary, 32 initialization rounds for HiAE are secure against integral attacks.

5.6 Key Committing Attacks

In this section, we present an initial analysis suggesting that our approach of XORing a message or data into multiple states during each iteration may offer better resistance to key-committing attacks. While defending against key-committing attacks is not our primary focus—since implementing the necessary countermeasures could introduce significant performance trade-offs—we have prioritized security features that address the most common threats without compromising overall system performance. However, our preliminary analysis indicates that the current structure of HiAE demonstrates stronger resistance to these attacks compared to previous designs such as AEGIS and Rocca. We provide an initial comparison of the FROB and CMT-1/2 security of HiAE, using the same methods for analyzing the key-committing security of AEGIS, Rocca, and Tiaoxin as described in [DFI+24, TTI24]. In [DFI+24], AEGIS and Rocca-S were compromised in practical time within the FROB and CMT-1/2 models, and later, in [TTI24], Rocca was also shown to be vulnerable in both models.

The key-committing attacks aim to find a ciphertext-tag pair that can be decrypted using two different sets of key and nonce. Specifically, we focus on identifying (K_1, N_1, AD) and (K_2, N_2, AD^*) pairs, where $K_1 \neq K_2$, that produce identical ciphertext-tags. The FROB attack is a special case of CMT-1/2 attacks where $N_1 = N_2$. HiAE follows the generalized state updating process as described in Fig.1. Initially, two keys, K_1 and K_2, along with an initialization vector N or two vectors N_1 and N_2, are chosen. Consider that encrypting associated data AD and plaintext P_1 using K_1 and N_1 results in ciphertext-tag $C||T$. Let $H = H[0]||H[1]||...||H[14]||H[15]$ represent the internal state after processing the AD. Let $S = S[0]||S[1]||S[2]||...||S[14]||S[15]$ represent the internal state after processing K_2 and N, and before processing the associated data. We aim to find a suitable associated data AD^* such that AD^* transforms S into H. If such an AD^* exists and has the same length as AD, the pair (K_1, N_1, AD, P_1) and (K_2, N_2, AD^*, P_1) will produce the same ciphertext-tag pair $C||T$, enabling the attacker to perform key-committing attacks.

We first follow the method of presenting the internal states by the system of equations of the starting states presented in [DFI+24]. At iteration i, only one block of data, AD_i^*, is processed and XORed with three states $S[2], S[12]$, and $S[15]$, resulting in at least a 128-bit constraint: $H_{new}[2] \oplus H_{old}[3] = H_{new}[12] \oplus H_{old}[13] = AD_i^*$. As a result, we need at least 16 blocks of associated data to provide sufficient degrees of freedom for transforming from $S = H_0$ to $H = H_{15}$. After 16 steps, the state H is derived from S as follows:

$$
\begin{aligned}
H[0] =& A(S[0] \oplus S[1]) \oplus A(S[13]) \oplus AD_0^* \oplus AD_2^* \oplus AD_{12}^* \\
H[1] =& A(S[1] \oplus S[2]) \oplus A(S[14]) \oplus AD_1^* \oplus AD_3^* \oplus AD_{13}^* \\
H[2] =& A(S[2] \oplus S[3] \oplus AD_0^*) \oplus A(S[15]) \oplus AD_2^* \oplus AD_4^* \oplus AD_{15}^* \\
H[3] =& A(S[3] \oplus AD_0^* \oplus S[4] \oplus AD_1^*) \oplus A(A(S[0] \oplus S[1]) \oplus A(S[13]) \oplus AD_0^*) \oplus AD_3^* \oplus AD_5^* \\
H[4] =& A(S[4] \oplus AD_1^* \oplus S[5] \oplus AD_2^*) \oplus A(A(S[1] \oplus S[2]) \oplus A(S[14]) \oplus AD_1^*) \oplus AD_4^* \oplus AD_6^* \\
H[5] =& A(S[5] \oplus AD_2^* \oplus S[6] \oplus AD_3^*) \oplus A(A(S[2] \oplus S[3] \oplus AD_0^*) \oplus A(S[15]) \oplus AD_2^*) \oplus AD_5^* \oplus AD_7^* \\
H[6] =& A(S[6] \oplus AD_3^* \oplus S[7] \oplus AD_4^*) \oplus A(A(S[3] \oplus AD_0^* \oplus S[4] \oplus AD_1^*) \oplus A(A(S[0] \oplus S[1]) \oplus A(S[13]) \oplus AD_0^*) \oplus AD_3^*) \\
& \oplus AD_6^* \oplus AD_8^* \\
H[7] =& A(S[7] \oplus AD_4^* \oplus S[8] \oplus AD_5^*) \oplus A(A(S[4] \oplus AD_1^* \oplus S[5] \oplus AD_2^*) \oplus A(A(S[1] \oplus S[2]) \oplus A(S[14]) \oplus AD_1^*) \oplus AD_4^*) \\
& \oplus AD_7^* \oplus AD_9^* \\
H[8] =& A(S[8] \oplus AD_5^* \oplus S[9] \oplus AD_6^*) \oplus A(A(S[5] \oplus AD_2^* \oplus S[6] \oplus AD_3^*) \oplus A(A(S[2] \oplus S[3] \oplus AD_0^* \oplus A(S[15]) \oplus AD_2^*) \oplus AD_5^*) \\
& \oplus AD_8^* \oplus AD_{10}^* \\
H[9] =& A(S[9] \oplus AD_6^* \oplus S[10] \oplus AD_7^*) \oplus A(A(S[6] \oplus AD_3^* \oplus S[7] \oplus AD_4^*) \oplus A(A(S[3] \oplus AD_0^* \oplus S[4] \oplus AD_1^* \\
& \oplus A(A(S[0] \oplus S[1]) \oplus A(S[13]) \oplus AD_0^*) \oplus AD_3^*) \oplus AD_6^*) \oplus AD_9^* \oplus AD_{11}^* \\
H[10] =& A(S[10] \oplus AD_7^* \oplus S[11] \oplus AD_8^*) \oplus A(A(S[7] \oplus AD_4^* \oplus S[8] \oplus AD_5^*) \oplus A(A(S[4] \oplus AD_1^* \oplus S[5] \oplus AD_2^* \\
& \oplus A(A(S[1] \oplus S[2] \oplus A(S[14]) \oplus AD_1^*) \oplus AD_4^*) \oplus AD_7^*) \oplus AD_{10}^* \oplus AD_{12}^* \\
H[11] =& A(S[11] \oplus AD_8^* \oplus S[12] \oplus AD_9^*) \oplus A(A(S[8] \oplus AD_5^* \oplus S[9] \oplus AD_6^*) \oplus A(A(S[5] \oplus AD_2^* \oplus S[6] \oplus AD_3^* \\
& \oplus A(A(S[2] \oplus S[3] \oplus A(S[15]) \oplus AD_2^*) \oplus AD_5^*) \oplus AD_8^*) \oplus AD_{11}^* \oplus AD_{13}^* \\
H[12] =& A(S[12] \oplus AD_9^* \oplus S[13] \oplus AD_0^* \oplus AD_{10}^*) \oplus A(A(S[9] \oplus AD_6^* \oplus S[10] \oplus AD_7^*) \oplus A(A(S[6] \oplus AD_3^* \oplus S[7] \oplus AD_4^* \\
& \oplus A(A(S[3] \oplus S[4] \oplus A(A(S[0] \oplus S[1]) \oplus A(S[13]) \oplus AD_0^*) \oplus AD_3^*) \oplus AD_6^*) \oplus AD_9^*) \oplus AD_{12}^* \oplus AD_{14}^* \\
H[13] =& A(S[13] \oplus AD_0^* \oplus AD_{10}^* \oplus S[14] \oplus AD_1^* \oplus AD_{11}^*) \oplus A(A(S[10] \oplus AD_7^* \oplus S[11] \oplus AD_8^*) \oplus A(A(S[7] \oplus AD_4^* \oplus S[8] \oplus AD_5^* \\
& \oplus A(A(S[4] \oplus S[5] \oplus A(A(S[1] \oplus S[2]) \oplus A(S[14]) \oplus AD_1^*) \oplus AD_4^*) \oplus AD_7^*) \oplus AD_{10}^*) \oplus AD_{13}^* \oplus AD_{15}^* \\
H[14] =& A(S[14] \oplus AD_1^* \oplus AD_{11}^* \oplus S[15] \oplus AD_2^* \oplus AD_{12}^*) \oplus A(A(S[11] \oplus AD_8^* \oplus S[12] \oplus AD_9^*) \oplus A(A(S[8] \oplus AD_5^* \oplus S[9] \oplus AD_6^* \\
& \oplus A(A(S[5] \oplus S[6] \oplus A(A(S[2] \oplus S[3]) \oplus A(S[15]) \oplus AD_2^*) \oplus AD_5^*) \oplus AD_8^*) \oplus AD_{11}^*) \oplus AD_{14}^* \\
H[15] =& A(S[15] \oplus AD_3^* \oplus AD_{12}^* \oplus A(S[0] \oplus S[1]) \oplus A(S[13]) \oplus AD_3^* \oplus AD_{13}^* \oplus A(A(S[12] \oplus AD_9^* \oplus S[13] \oplus AD_{10}^*) \\
& \oplus A(A(S[9] \oplus AD_6^* \oplus S[10] \oplus AD_7^* \oplus A(A(S[6] \oplus S[7] \oplus A(A(S[3] \oplus S[4]) \oplus A(A(S[0] \oplus S[1]) \oplus A(S[13]) \oplus AD_0^*) \oplus AD_3^*) \\
& \oplus AD_6^*) \oplus AD_9^*) \oplus AD_{12}^*) \oplus AD_{15}^*.
\end{aligned}
$$

From the system of equations above, given the information of H and S, no block AD_i^* can be retrieved for free. Thus, guessing at least one 128-bit block is necessary to solve the system, which counters the time complexity of a generic attack.

The second method using differential technique which breaks the security of Rocca in the CMT-1/2 and FROB models as described in [TTI24], however, does not trivially apply to HiAE. After processing the initial state with different keys and nonces, the states S become fully distinct due to the diffusion in the initialization phase. By exploiting differences in the associated data (AD), which can be freely chosen by the attacker, [TTI24] successfully eliminated all differences in the internal state. The reason for this is that in Rocca, Rocca-S, and AEGIS, XORing one or two message blocks to only one or two states in each iteration weakens the constraint between messages and internal states. This issue, however, is avoided in HiAE. In HiAE, each iteration involves XORing a block of message (data) into three blocks of the state S_2, S_{12}, and S_{15}. Attempting to cancel out one block state using the difference from a block of message introduces new differences into other block states. Moreover, canceling multiple block states is equivalent to finding a collision for the 128-bit block, which is no lower than 2^{64}. Therefore, we believe that a differential attack on HiAE would have a complexity no less than 2^{64}. An example of this is shown in Fig. 3, where AD_0 to AD_9 are used to cancel the differences in S_{12} to S_3, and AD_{10} is used for S_{15} but this results in new differences in S_2 and S_{12}. We also used MILP model to validated our observation.

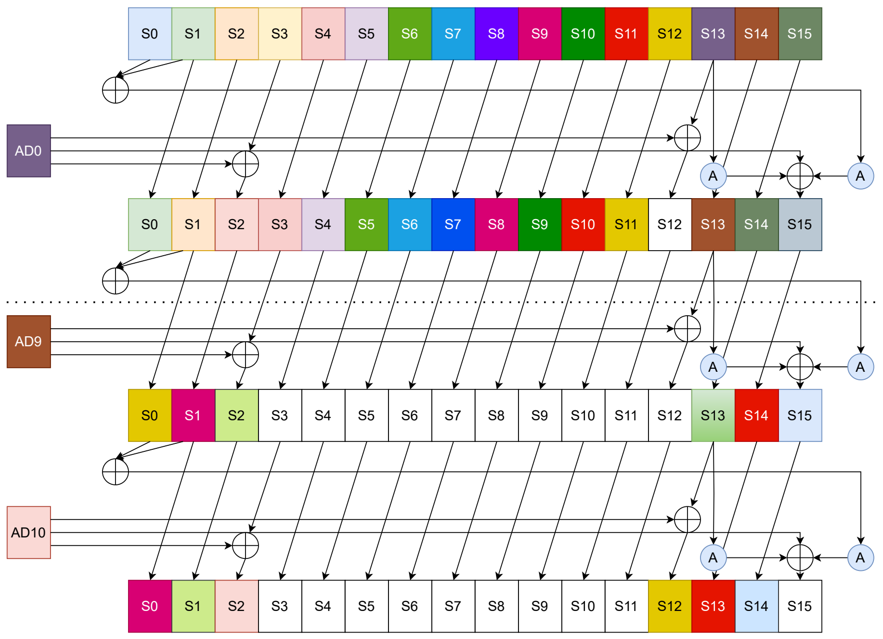

Figure 3: Example of using the differences of associated data to cancel the differences in the states

Our design employs a 128-bit tag, and the generic attack follows the birthday paradox algorithm, with a forgery time complexity bound of 2^{64}. Based on our analysis above, we believe that it is unlikely to find key-committing attacks with a complexity lower than that of the generic attack.

However, our design is not resistant to the CMT-3 attack, where everything is committed, and the attacker can use the same key and nonce for both queries. While this attack is a potential vulnerability, we consider this scenario unlikely in most real-world applications, as the majority of applications only require key-committing properties, as noted in [ADG+22, GLR17].

6 Performance Evaluation

6.1 Test Environment and Configuration

We conduct comprehensive benchmarking across 5 ARM processors, thedetail see Table 9 The experimental configurations are as follows:

- **Kunpeng Server Platform**: openEuler 22.03 with GCC 10.3
- **Mobile Devices**: Huawei Mate 50/60 smartphones with ArkCompiler 3.0.0.5
- **Router**: RK3568 development board with openWRT 22.03 OS and openWRT GCC 11.2.0

To ensure measurement consistency and cross-platform comparability, we implement the following unified configurations:

- Maximum optimization level (`-O3`) enabled for all cryptographic implementations
- Mobile device tests conducted in performance mode with CPU affinity binding to prime cores

Table 9: Processor Specifications

	Chip	Microarchitecture	Platform	ISA Support	Launch
ARM	HiSilicon Kunpeng 920	TaiShan V110	Server	ARMv8.2-A	2019-Q1
	HiSilicon Kunpeng 920X	TaiShan V120	Server	ARMv9.2-A	N/A
	HiSilicon Kirin 9000S	TaiShan V120	Mobile	ARMv9.2-A	2023-Q3
	Qualcomm Snapdragon 8+ Gen1	Cortex-X2	Mobile	ARMv9-A	2022-Q2
	Rockchip RK3568	Cortex-A55	Router	ARMv8.2-A	2020-Q4

6.2 Contrastive Ciphers Software Implementation

We conduct a comprehensive performance comparison among state-of-the-art AES-based authenticated encryption with only encryption and associated data (AEAD) schemes, including SNOW-V [EJMY18], AEGIS [WP14], Rocca [SLN+22], and Rocca-S [ABC+24]. For AEGIS, we only select AEGIS-128L for comparison as the optimal branch of the AEGIS family. For SNOW-V, the AEAD mode tested is SNOW-V-GCM. During the AEAD mode tests, the associated data (AD) length is set to 48 bytes. We also measure the performance of AES-256-CTR and AES-256-GCM for encryption and AEAD modes as a baseline.

For Rocca, AEGIS, and SNOW-V, we use their respective open-source implementations, which are provided in the papers or on the authors' websites. For Rocca-S, which has only slight differences from Rocca, we modify Rocca's open-source code based on the descriptions in the paper. These open-source implementations for x86 are then converted to ARM using SSE2NEON [DLT17]. Specifically, `_mm_aesenc_si128(x, y)` is replaced with `veorq_u8(vaesmcq_u8(vaeseq_u8(x, 0), y))`, and `_mm_xor_si128` is replaced with `veorq_u8`.

For AES-256-CTR and AES-256-GCM, we use the OpenSSL (3.0) implementation for ARM servers. For mobile phones, we implement AES-256 manually.

6.3 HiAE Software Implementation

For the implementation of HiAE, we note the shift-state structure involve overhead to compute the state index. To mitigate latency induced by the shift-state structure's dynamic indexing, we employ loop unrolling techniques: 16 rounds of state updates are unrolled to eliminate shift-state branching overhead, while both the 32-round initialization and finalization phases undergo full unrolling. A detailed analysis of these optimizations appears in Appendix B. On ARM NEON with the cryptographic extension, we utilize `vaesmcq_u8` and `vaeseq_u8` for AESR, and `veorq_u8` for XOR.

6.4 Evaluation and Result Analysis

We assess the performance of HiAE, with results presented in Table 10. HiAE achieves encryption speeds exceeding 97 Gbps on ARM devices when processing long messages. It outperforms all comparison algorithms across various ARM devices.

7 Conclusion

In this work, we addressed the challenges of designing cryptographic algorithms that achieve both high performance and cross-platform efficiency. With the growing demands of next-generation data transmission systems, particularly in 6G networks, we introduced HiAE — an advanced high-throughput authenticated encryption algorithm that balances performance and security across diverse computational platforms.

Table 10: Performance on ARM Chips (Gbps)

Encryption Only								AEAD							
Input Size (bytes)	16384	8192	4096	2048	1024	256	64	Input Size (bytes)	16384	8192	4096	2048	1024	256	64
Qualcomm Snapdragon 8+ Gen1															
HiAE (this work)	**120.85**	**115.01**	**104.50**	**88.67**	66.17	27.27	6.87	HiAE (this work)	**115.32**	**106.56**	92.80	74.06	52.88	19.20	4.68
Rocca	103.85	101.29	95.58	86.65	**72.99**	**37.53**	**13.16**	Rocca	99.72	99.37	**94.50**	**82.35**	**67.57**	**32.14**	**10.41**
Rocca-S	79.86	80.18	77.32	68.79	58.30	30.34	10.56	Rocca-S	83.45	79.96	73.97	66.30	54.04	26.08	8.44
AEGIS-128L	71.12	69.45	66.36	61.79	54.07	31.05	11.49	AEGIS-128L	68.41	66.77	62.30	56.52	46.88	23.05	8.04
SNOW-V	18.94	18.65	18.13	17.97	11.09	5.45	1.36	SNOW-V-GCM	5.55	5.54	5.50	5.43	5.29	4.60	3.02
AES-256-CTR	35.48	34.55	32.99	30.15	26.23	14.41	4.71	AES-256-GCM	7.51	7.36	7.36	7.21	6.96	5.72	3.17
Hisilicon Kirin 9000S															
HiAE (this work)	**102.24**	**98.02**	**89.01**	**75.30**	**57.51**	**23.89**	**5.67**	HiAE (this work)	**97.67**	**89.7**	**78.07**	**61.31**	42.86	15.30	3.99
Rocca	71.32	69.32	65.43	59.07	46.88	15.46	1.81	Rocca	70.82	68.26	63.77	56.25	**45.53**	**21.24**	**6.80**
Rocca-S	58.37	55.38	53.52	48.71	39.40	13.18	3.45	Rocca-S	58.49	55.65	52.26	44.37	36.79	17.01	5.39
AEGIS-128L	48.08	47.20	45.37	42.15	36.83	21.02	3.52	AEGIS-128L	47.57	46.09	43.34	29.62	31.97	15.62	2.23
SNOW-V	18.94	18.65	18.13	17.97	11.09	5.45	1.36	SNOW-V-GCM	5.55	5.54	5.50	5.43	5.29	4.60	3.02
AES-256-CTR	29.47	28.84	11.77	9.63	7.86	4.75	2.00	AES-256-GCM	6.50	6.39	6.09	6.40	6.19	5.16	3.10
HiSilicon Kunpeng 920															
HiAE (this work)	**56.14**	**55.06**	**52.20**	**47.10**	**39.47**	**20.13**	5.55	HiAE (this work)	**53.72**	**50.80**	**45.45**	**37.63**	**27.95**	10.99	2.92
Rocca	42.81	41.31	38.92	34.66	28.47	13.74	4.47	Rocca	41.17	38.28	33.85	27.42	19.79	7.44	2.13
Rocca-S	34.27	33.23	32.24	29.15	24.72	12.57	4.25	Rocca-S	33.64	31.70	28.37	23.43	17.46	6.83	1.99
AEGIS-128L	23.38	23.02	22.41	21.20	19.14	12.16	4.99	AEGIS-128L	23.11	22.51	21.44	19.55	16.78	8.95	3.08
SNOW-V	14.34	14.09	13.49	12.61	10.95	6.30	2.32	SNOW-V-GCM	4.92	4.90	4.86	4.78	4.63	3.89	2.38
AES-256-CTR	21.54	21.43	21.21	20.92	19.94	16.10	**9.09**	AES-256-GCM	14.66	14.60	14.41	14.45	14.10	**12.15**	**7.52**
HiSilicon Kunpeng 920X															
HiAE (this work)	**81.28**	**76.62**	**74.85**	**65.73**	**56.78**	28.89	8.37	HiAE (this work)	**75.08**	**71.10**	**63.09**	**52.33**	**38.69**	15.08	3.81
Rocca	48.59	47.19	44.48	39.87	33.06	16.31	5.39	Rocca	46.90	44.06	39.25	32.21	23.72	9.18	2.66
Rocca-S	43.00	41.76	39.38	35.34	29.33	14.51	4.81	Rocca-S	41.54	39.02	34.76	28.56	21.03	8.15	2.36
AEGIS-128L	41.81	41.07	39.49	36.72	32.21	18.55	6.86	AEGIS-128L	41.32	40.06	37.71	33.74	27.88	13.65	4.49
SNOW-V	18.70	18.17	17.26	15.64	13.17	6.77	2.30	SNOW-V-GCM	6.15	6.13	6.09	6.01	5.86	5.10	3.38
AES-256-CTR	39.16	39.21	38.61	37.52	36.72	**28.91**	**17.88**	AES-256-GCM	29.54	29.28	28.85	27.94	26.31	**16.65**	**11.34**
Rockchip RK3568															
HiAE (this work)	**15.40**	**14.97**	**14.88**	**14.20**	**12.81**	**8.04**	2.11	HiAE (this work)	**14.70**	**14.62**	**13.64**	**12.01**	**9.63**	**4.42**	1.13
Rocca	11.15	11.14	10.60	9.67	8.24	4.32	1.49	Rocca	10.83	10.55	9.58	8.11	6.19	2.56	0.74
Rocca-S	11.46	11.39	10.86	9.86	8.38	4.34	1.49	Rocca-S	11.09	10.81	9.79	8.28	6.30	2.58	0.74
AEGIS-128L	8.98	8.90	8.79	8.34	7.56	4.89	2.01	AEGIS-128L	8.88	8.83	8.50	7.87	6.82	3.80	1.37
SNOW-V	4.65	4.58	4.33	3.92	3.29	1.67	0.57	SNOW-V-GCM	2.00	1.99	1.99	1.97	1.93	1.71	1.19
AES-256-CTR	8.32	8.28	8.15	7.97	7.49	5.62	**2.89**	AES-256-GCM	5.63	5.63	5.55	5.50	5.26	4.20	**2.41**

Compared to prior AEAD designs aimed at high-throughput, such as SNOW-V, Rocca, and AEGIS, HiAE stands out with its 2048-bit internal state, utilizing 16 128-bit registers, while minimizing the number of AES instructions required. This design aligns well with the architectural capabilities of next-generation ARM and x86 processors, which support up to 32 128-bit registers.

Benchmark results from software implementations across various platforms demonstrate HiAE's exceptional efficiency. The algorithm achieved over 97 Gbps on ARM devices in AEAD mode, making it the fastest AEAD solution on ARM chips.

We will open source the implementation and call for more benchmark on various x86 and ARM chips.

References

[ABC$^+$24] Ravi Anand, Subhadeep Banik, Andrea Caforio, Kazuhide Fukushima, Takanori Isobe, Shisaku Kiyomoto, Fukang Liu, Yuto Nakano, Kosei Sakamoto, and Nobuyuki Takeuchi. An ultra-high throughput aes-based authenticated encryption scheme for 6g: Design and implementation. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *Computer Security – ESORICS 2023*, pages 229–248, Cham, 2024. Springer Nature Switzerland.

[ADG$^+$22] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 3291–3308. USENIX Association, 2022.

[AR19] Andreas Abel and Jan Reineke. uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 673–686, 2019.

[BBL+24] Augustin Bariant, Jules Baudrin, Gaëtan Leurent, Clara Pernot, Léo Perrin, and Thomas Peyrin. Fast aes-based universal hash functions and macs: Featuring lemac and petitmac. *IACR Transactions on Symmetric Cryptology*, 2024(2):35–67, 2024.

[BS12] Daniel J Bernstein and Peter Schwabe. Neon crypto. In *Cryptographic Hardware and Embedded Systems–CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings 14*, pages 320–339. Springer, 2012.

[Com16] ARM Community. Arm® cortex®-a57 software optimization guide, issue b, 2016.

[Com18] ARM Community. Arm® cortex®-a75 software optimization guide, verion 2.0, 2018.

[Com19a] ARM Community. Arm® cortex®-a76 software optimization guide, verion 10.0, 2019.

[Com19b] ARM Community. Arm® cortex®-a77 software optimization guide, issue 3.0, 2019.

[Com20] ARM Community. Arm® cortex®-a78 core software optimization guide, issue 3.0, 2020.

[Com21a] ARM Community. Arm® cortex®-x1 core software optimization guide, issue 4.0, 2021.

[Com21b] ARM Community. Arm® cortex®-x2 core software optimization guide, issue 5.0, 2021.

[Com22a] ARM Community. Arm® cortex®-a510 software optimization guide, issue 6.0, 2022.

[Com22b] ARM Community. Arm® cortex®-a55 software optimization guide, issue 4.0, 2022.

[Com22c] ARM Community. Arm® cortex®-a715 core software optimization guide, issue 4.0, 2022.

[Com22d] ARM Community. Arm® cortex®-x3 core software optimization guide, issue 4.0, 2022.

[Com22e] ARM Community. Arm® neoverse™ n2 software optimization guide, issue 5.0, 2022.

[Com22f] ARM Community. Arm® neoverse™ v1 software optimization guide, issue 6.0, 2022.

[Com22g] ARM Community. Arm® neoverse™ v2 core software optimization guide, issue 2.0, 2022.

[Com23a] ARM Community. Arm® cortex®-a520 core software optimization guide, issue 1.2, 2023.

[Com23b] ARM Community. Arm® cortex®-a720 core software optimization guide, issue 7.0, 2023.

[Com23c] ARM Community. Arm® cortex®-x4 core software optimization guide, issue 3.0, 2023.

[Com24a] ARM Community. Arm neon programming quick reference. https://community.arm.com/arm-community-blogs/b/operating-systems-blog/posts/arm-neon-programming-quick-reference, 2024. Accessed: 2024-07-18.

[Com24b] ARM Community. Arm® cortex®-a720 core software optimization guide, issue 3.0, 2024.

[Com24c] ARM Community. Arm® cortex®-x925 core software optimization guide, issue 3.0, 2024.

[Com24d] ARM Community. Arm® neoverse™ n3 core software optimization guide, issue 2.0, 2024.

[Com24e] ARM Community. Arm® neoverse™ v3 core software optimization guide, issue 2.0, 2024.

[Cora] Intel Corporation. Advanced encryption standard instructions (aes-ni). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html?wapkw=AES-NI. Accessed: 2024-07-17.

[Corb] Intel Corporation. Optimizing earlier generations of intel® 64 and ia-32 processor architectures, throughput, and latency. Accessed: 2024-07-25.

[Corc] NVIDIA Corporation. Nvidia nvlink: High-speed gpu interconnect. https://www.nvidia.com/en-sg/data-center/nvlink/. Accessed: 2024-11-21.

[Cor24] Intel Corporation. Intel® intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, 2024. Accessed: 2024-07-18.

[DFI+24] Patrick Derbez, Pierre-Alain Fouque, Takanori Isobe, Mostafizar Rahman, and André Schrottenloher. Key committing attacks against aes-based AEAD schemes. *IACR Trans. Symmetric Cryptol.*, 2024(1):135–157, 2024.

[DLT17] DLTcollab. sse2neon: A c/c++ header file for translating intel sse intrinsics into arm neon intrinsics. https://github.com/DLTcollab/sse2neon, 2017. Accessed: 2024-11-21.

[EJMY18] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new snow stream cipher called snow-v. *Cryptology ePrint Archive*, 2018.

[ENP19] Maria Eichlseder, Marcel Nageler, and Robert Primas. Analyzing the linear keystream biases in AEGIS. *IACR Trans. Symmetric Cryptol.*, 2019(4):348–368, 2019.

[GLR17] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 66–97. Springer, 2017.

[HII$^+$22] Akinori Hosoyamada, Akiko Inoue, Ryoma Ito, Tetsu Iwata, Kazuhiko Minematsu, Ferdinand Sibleyras, and Yosuke Todo. Cryptanalysis of rocca and feasibility of its security claim. *IACR Trans. Symmetric Cryptol.*, 2022(3):123–151, 2022.

[Hos24] Akinori Hosoyamada. Quantum algorithms for fast correlation attacks on lfsr-based stream ciphers. *IACR Cryptol. ePrint Arch.*, page 894, 2024.

[Inc24] Apple Inc. Apple silicon cpu optimization guide: 3.0, 2024. Accessed: 2025-01-07.

[JN16] Jérémy Jean and Ivica Nikolic. Efficient design strategies based on the AES round function. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 334–353. Springer, 2016.

[LAL$^+$19] Matti Latva-Aho, Kari Leppänen, et al. Key drivers and research challenges for 6g ubiquitous wireless intelligence. 2019.

[LIMS21] Fukang Liu, Takanori Isobe, Willi Meier, and Kosei Sakamoto. Weak keys in reduced AEGIS and tiaoxin. *IACR Trans. Symmetric Cryptol.*, 2021(2):104–139, 2021.

[MWGP11] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In Chuankun Wu, Moti Yung, and Dongdai Lin, editors, *Information Security and Cryptology - 7th International Conference, Inscrypt 2011, Beijing, China, November 30 - December 3, 2011. Revised Selected Papers*, volume 7537 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2011.

[oSN07] National Institute of Standards and Technology (NIST). Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac, 2007. Accessed: 2025-01-06.

[oSN25] National Institute of Standards and Technology (NIST). Nist lightweight cryptography (lwc), 2025. Accessed: 2025-01-06.

[SLN$^+$22] Kosei Sakamoto, Fukang Liu, Yuto Nakano, Shinsaku Kiyomoto, and Takanori Isobe. Rocca: An efficient AES-based encryption scheme for beyond 5G (full version). Cryptology ePrint Archive, Paper 2022/116, 2022.

[TTI24] Ryunouchi Takeuchi, Yosuke Todo, and Tetsu Iwata. Key recovery, universal forgery, and committing attacks against revised rocca: How finalization affects security. *IACR Trans. Symmetric Cryptol.*, 2024(2):85–117, 2024.

[WP14] Hongjun Wu and Bart Preneel. Aegis: A fast authenticated encryption algorithm. In *Selected Areas in Cryptography–SAC 2013: 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers 20*, pages 185–201. Springer, 2014.

A On the Linear Bias Keystream Attack

We have adapted the bitwise model proposed in [ENP19] for AEGIS to HiAE in order to identify the accurate linear differential characteristics from truncated linear differential trails. In our modified model, we focus on searching for valid bitwise trails, ensuring that all S-box linear transitions with non-zero probabilities are considered. After running the SMT model on all 11420 truncated trails listed in Table A11, no bitwise differential characteristics were found. This suggests that the square correlation of HiAE is not greater than 2^{-150}. Based on this, we conclude that HiAE has a security level of 150 bits against linear bias keystream attack.

Table A11: The list of truncated linear differential trails

No. of active S-boxes	19	20	21	22	23	24
No. of trails	369	0	1116	424	7922	1589

B Shift State Overhead in Update Loops

HiAE uses a shift-state update function. A naive implementation involves using an additional temporary variable to store the updated state, then shifting the 15 states and applying the updated state back from the temporary variable. However, this approach is clearly not optimal for the state update. For HiAE, only 3 states change each round, but with shift state, each round need to update all 16 states.

A more efficient approach is to keep most of the state registers unchanged and only update the state by computing index offset for each round. This approach is our initial implementation, which reduce HiAE's duplicate operations a lot, but still facing an overhead for computing offset.

To achieve the highest throughput as we expect, we need to remove all the overhead and remain only the computing for state update and encryption. Therefore, for the HiAE implementation, we unrolled both the initialization and finalization loops. This transformation entirely eliminate the offset overhead. Unrolling the loops also reduces the overhead from loop iterations and counter increments.

Our experiments show that this optimization leads to a significant performance improvement for short-length messages in HiAE. However, similar optimizations do not yield a noticeable performance boost for other ciphers like Rocca or AEGIS. This is because, in these ciphers, most of the state gets updated by the state update computing and do not involve any index overhead, and the impact of this optimization is minimal.

The experimental setup follows the same conditions described in Section 6, with results shown in Table B12. As illustrated, by eliminating the offset and extra memory communication costs, HiAE achieves a significant performance gain, especially for short messages. In these cases, initialization and finalization account for a larger portion of the total execution time compared to encryption. Additionally, on x86 devices, the offset overhead consumes a larger proportion of instructions, further amplifying the benefit of this optimization. For other cipher designs, which do not face the same issue, the performance remains largely unaffected. As a result, we retain the original implementation without unrolling for these ciphers, as detailed in Section 6.

Table B12: AEAD Performance difference with (*) and without unroll (Gbps)

Input Size (bytes)	16384	8192	4096	2048	1024	256	64
Hisilicon Kunpeng 920 (Taishan V110, launched in 2019)							
HiAE*	**53.72**	**50.80**	**45.45**	**37.63**	**27.95**	**10.99**	**2.92**
HiAE	53.47	50.54	44.45	35.6	25.52	9.47	2.48
Δ	+0.25	+0.26	+1.00	+2.03	+2.43	+1.52	+0.44
Δ%	+0.47%	+0.51%	+2.25%	+5.70%	+9.52%	+16.05%	+17.74%
Rocca*	41.26	38.52	34.14	27.86	20.37	7.78	2.24
Rocca	41.17	38.28	33.85	27.42	19.79	7.44	2.13
Δ	+0.09	+0.24	+0.29	+0.44	+0.58	+0.34	+0.11
Δ%	+0.22%	+0.63%	+0.86%	+1.60%	+2.93%	+4.57%	+5.16%
Intel Xeon W9 3495X (Sapphire Rapids, launched in 2023)							
HiAE*	**171.46**	**163.09**	**148.79**	**127.72**	**98.63**	**41.20**	**9.38**
HiAE	163.40	149.28	123.20	96.04	66.61	22.93	5.38
Δ	+8.06	+13.81	+25.59	+31.68	+32.02	+18.27	+4.00
Δ%	+4.93%	+9.25%	+20.77%	+32.99%	+48.07%	+79.68%	+74.35%
Rocca*	116.02	101.97	81.45	59.01	37.95	14.30	3.90
Rocca	117.38	103.30	83.93	59.73	38.54	14.59	4.00
Δ	-1.36	-1.33	-2.48	-0.72	-0.59	-0.29	-0.10
Δ%	-1.16%	-1.29%	-2.95%	-1.20%	-1.53%	-1.99%	-2.50%

```
.L8:
vmovdqa %xmm6, %xmm4
vmovdqa %xmm7, %xmm5
decq  %rax
vpxor %xmm7, %xmm1, %xmm6
vaesenc %xmm2, %xmm8, %xmm7
vaesenc %xmm3, %xmm2, %xmm8
vpxor %xmm10, %xmm3, %xmm2
vaesenc %xmm0, %xmm9, %xmm3
vpxor %xmm5, %xmm0, %xmm9
vaesenc %xmm4, %xmm1, %xmm0
vpxor %xmm11, %xmm4, %xmm1
jne  .L8








        ...
```

```
.L2:
mov      v2.16b, v5.16b
subs     x2, x2, #1
mov      v4.16b, v5.16b
aese     v1.16b, v5.16b
aesmc    v1.16b, v1.16b
aese     v6.16b, v5.16b
aesmc    v6.16b, v6.16b
aese     v2.16b, v16.16b
aesmc    v2.16b, v2.16b
aese     v4.16b, v18.16b
aesmc    v4.16b, v4.16b
eor      v21.16b, v1.16b, v16.16b
eor      v23.16b, v6.16b, v7.16b
eor      v20.16b, v19.16b, v18.16b
eor      v3.16b, v19.16b, v7.16b
eor      v2.16b, v2.16b, v17.16b
eor      v22.16b, v17.16b, v26.16b
eor      v4.16b, v4.16b, v0.16b
eor      v24.16b, v0.16b, v25.16b
         ...
bne      .L2
```

Figure B4: Rocca initialization Assembly Code on x86 (Left) and ARM (right)

```
.L5:
vmovdqa     (%rcx), %xmm0
movq        %r8, %rax
.L3:
leaq        1(%rax), %r8
leaq        13(%rax), %rdx
movq        %rax, %rsi
addq        $3, %rax
movq        %r8, %rcx
andl        $15, %edx
andl        $15, %eax
andl        $15, %esi
andl        $15, %ecx
salq        $4, %rdx
salq        $4, %rax
salq        $4, %rsi
salq        $4, %rcx
addq        %rdi, %rdx
addq        %rdi, %rax
addq        %rdi, %rcx
vmovdqa     (%rdx), %xmm6
vpxor       (%rcx), %xmm0, %xmm0
vaesenc     %xmm1, %xmm0, %xmm0
vaesenc     %xmm0, %xmm6, %xmm0
vmovdqa     %xmm0, (%rdi,%rsi)
vpxor       (%rax), %xmm1, %xmm0
vmovdqa     %xmm0, (%rax)
vpxor       (%rdx), %xmm1, %xmm0
vmovdqa     %xmm0, (%rdx)
cmpq        $32, %r8
jne         .L5
```

```
.L5:
ldr q0, [x0, x5]
.L3:
add x4, x2, 1
add x1, x2, 13
add x3, x2, 3
ubfiz x5, x4, 4, 4
ubfiz x6, x2, 4, 4
ubfiz x1, x1, 4, 4
ubfiz x3, x3, 4, 4
mov x2, x4
ldr q3, [x0, x5]
ldr q1, [x0, x1]
aese  v0.16b, v3.16b
aesmc v0.16b, v0.16b
aese  v1.16b, v4.16b
aesmc v1.16b, v1.16b
eor v0.16b, v0.16b, v1.16b
eor v0.16b, v0.16b, v2.16b
str q0, [x0, x6]
ldr q0, [x0, x3]
eor v0.16b, v0.16b, v2.16b
str q0, [x0, x3]
ldr q0, [x0, x1]
eor v0.16b, v0.16b, v2.16b
str q0, [x0, x1]
cmp x4, 32
bne .L5


        ...
```

Figure B5: HiAE initialization Assembly Code on x86 (Left) and ARM (right), offset overhead instructions are marked in red.