IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 29 June 2023, accepted 18 July 2023, date of publication 24 July 2023, date of current version 31 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3298026

RESEARCH ARTICLE

AVX-Based Acceleration of ARIA Block Cipher Algorithm

TAE-HEE YOO[1], JUSSI KIVILINNA[2], AND CHOONG-HEE CHO[3], (Member, IEEE)
[1]Department of Cloud, Kakao Enterprise Corporation, Seongnam 13494, South Korea
[2]Oura Health, 90590 Oulu, Finland
[3]Department of Computer Science and Engineering, Sahmyook University, Seoul 01795, South Korea

Corresponding author: Choong-Hee Cho (cch@syu.ac.kr)

This paper was supported by the Samyook University Research Fund in 2023.

ABSTRACT Block cipher algorithms encrypt sensitive personal, financial, and confidential information to prevent unauthorized access. The ARIA is a general block cipher algorithm with an involutional SPN structure optimized for lightweight environments and hardware implementation. This study focuses on implementing ARIA in the crypto-subsystem of the Linux kernel because it has yet to be implemented despite being recognized as a global standard. This study improves the practicality of ARIA by implementing it in the Linux kernel with reasonable performance and attempts to reduce CPU cycles for substitution and diffusion operations while alleviating the lack of ARIA-specific instructions in existing CPUs. To achieve this, the study implemented the AVX, AVX2, and AVX512 versions of ARIA that can operate in parallel in addition to two types of ARIA-specific substitution functions using AES-NI and GFNI. We implemented an accelerated version of ARIA that performs up to 10.6 times better than the generic version. The optimization of the affine transformation in AES-NI based ARIA has been shown to reduce the required cycle count by 32.2%. Moreover, ARIA demonstrated competitive speeds when compared to other algorithms, such as Camellia, that are implemented in the Linux kernel.

INDEX TERMS ARIA algorithm, Linux kernel, AVX, AES-NI, GFNI.

I. INTRODUCTION

As personal, financial, and confidential data are increasingly being shared and stored online, and their protection from unauthorized access has become increasingly important. Block cipher algorithms encrypt this sensitive information, rendering it unreadable by anyone without an appropriate decryption key. In addition, with the development of data analysis technology, real-time data encryption and decryption are becoming increasingly important as the amount of data collected on networks increases rapidly. This growth in data collection has increased the need for high-performance block cipher algorithms that can encrypt and decrypt large amounts of data quickly and efficiently.

ARIA [1] is a general-purpose block cipher algorithm with an Involutional Substitution Permutation Network (ISPN) structure with the same encryption and decryption processes optimized for lightweight environments and hardware implementations. ARIA is a derivative of the Advanced Encryption Standard (AES) [2] that incorporates its structure and design principles with modifications to enhance security. Both algorithms perform substitution and diffusion in each round and have the same input (128-bit) and key sizes (128-bit, 192-bit, 256-bit). AES is an SPN with various encryption and decryption processes. AES uses one substitution function, S_1, for encryption and its inverse, S_1^{-1}, for decryption. ARIA also uses S_1 and S_1^{-1} but additionally uses other substitution functions S_2 and S_2^{-1}, for a total of four substitution functions. Unlike AES, ARIA uses all four substitution functions for encryption and decryption. This indicates that although AES instructions can be utilized to implement ARIA, some of them, S_2 and S_2^{-1}, must be implemented differently.

In this study, we first analyzed open-source operating systems to understand the implementation and usage of ARIA. This is because, generally, block cipher algorithms

The associate editor coordinating the review of this manuscript and approving it for publication was Wei Liu.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

are components of the operating system and are accessible to users. Including ARIA in open-source Linux, an operating system standard, can enhance user accessibility, as it can be integrated into various commercial devices that run on Linux. Block cipher algorithms, like ARIA, are typically incorporated into user-space libraries, such as OpenSSL and GNUTLS, and are used in the context of Layer 7 within a network. However, these algorithms can also find significant utility in a networking, specifically from Layer 2 to Layer 4 (L2 to L4). In the Linux, the functionalities associated with these layers are implemented within the kernel. In situations where a block cipher algorithm, such as ARIA, is not inherently integrated into the cryptographic subsystem in the kernel of Linux, the range of algorithm selection options available to the user can be limited. It is important to note that block cipher algorithms can serve not only networking purposes within the Linux kernel but can also contribute to file system and storage operations. Therefore, it is essential to implement algorithms that are fundamentally demanded in both the user-space and the kernel. Moreover, when implementing ARIA in the kernel, it is crucial to ensure commercially viable performance. Since encryption/decryption is inherently a time-consuming operation, it should be implemented using various methods (such as utilizing physical acceleration features and adopting parallel techniques) to maximize performance as much as possible. Considering that Linux is commonly deployed in CPU-based environments, the implementation should be optimized for CPU-based performance. Thus, we investigate security-related Linux implementations to explore the incorporation of ARIA algorithm in the kernel with reasonable performance by utilizing not only the fundamental implementation of ARIA but also the acceleration capabilities supported by the CPU.

Transport Layer Security (TLS) in Linux, which ensures end-to-end data integrity and confidentiality, comprises the TLS handshake step in OpenSSL of the user space and the encryption/decryption process in the kernel implementation of TLS (kTLS). The kTLS in the Linux kernel operates on the Upper Layer Protocol (ULP), which runs on top of the TCP layer. Figure 1 shows the implementation of the block cipher algorithms in the kernel. The crypto subsystem can be utilized in any part of the kernel, and Fig. 1 illustrates its usage in the network subsystem. SM4 [3], [4] and Camellia [5], [6] are among the algorithms incorporated into the crypto subsystem of the kernel. Each generic version of the algorithm can be considered the simplest implementation in the kernel using the C programming language. The other versions could be optimized based on instructions provided by the CPU. Similar to SM4 and Camellia, ARIA is a standardized algorithm [7] and TLS standard [8]. However, unlike other algorithms, ARIA does not have a generic kernel implementation although it has long been established as a standard. Therefore, in this study, we applied ARIA to the crypto subsystem in the kernel and enhanced its practicality by maximizing its performance.

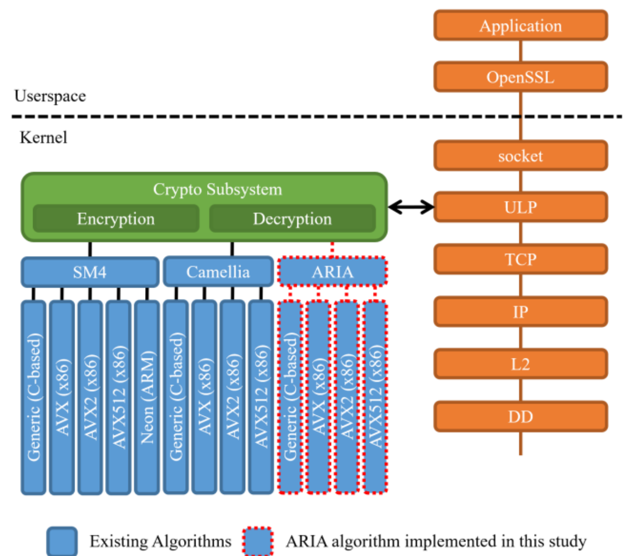


FIGURE 1. Implementations of block cipher algorithms in the kernel.

Implementing ARIA in the kernel presents three challenges. The first is decreasing the number of CPU cycles during substitution and diffusion operations, which have a high overhead, and the second is that current CPUs do not have ARIA-specific instructions. This study makes two contributions towards solving these problems. First, we implemented the Advanced Vector eXtensions (AVX) [9], AVX2, and AVX512 versions of ARIA that can work in parallel with the generic version. Second, we implemented ARIA-specific substitution functions using the AES-New Instructions (AES-NI) [10] and Galois Field New Instructions (GFNI) [11] and applied these codes to the main branch of the Linux kernel. The two contributions presented are novel and have not been previously reported. These performance-optimized implementations enhance the practicality and performance of ARIA in the kernel, making it more suitable for modern cryptography applications that require fast and efficient encryption and decryption of large amounts of data.

The remainder of this paper is organized as follows. Section II presents a comprehensive overview of block cipher cryptography and its associated algorithms and presents techniques that enhance their performance. Section III presents a comprehensive analysis of the relevant literature on block cipher algorithms and their historical development in the Linux operating system. Section IV proposes a high-performance ARIA implementation in a Linux kernel. The performance of the proposed implementation is thoroughly evaluated in Section V. Finally, conclusions are drawn in Section VI, and future research avenues are identified.

II. BACKGROUND

A block cipher [12], [13] operates on fixed-size data blocks, typically of 64 or 128 bits at once, using a secret key. It encrypts plaintext into ciphertext by applying mathemat-

ical operations such as substitution and permutation. Block ciphers are used to encrypt and decrypt data transmitted over networks, files, and disk drives, sensitive information in payment systems, data stored on mobile devices and embedded systems, and to verify firmware authenticity during the boot process.

Several well-known algorithms exist for block ciphers, including the AES [2], which is regarded as one of the most secure and efficient block cipher algorithms. The US government has adopted AES, which is utilized in various applications such as wireless networks, disk encryption, and secure communication protocols. The Data Encryption Standard (DES) is an older block cipher algorithm that utilizes a key length of 56 bits and a block size of 64 bits. However, owing to its short key length, it has been largely replaced by AES and triple DES (3DES), which is a variation of the DES algorithm that uses a key length of 168 bits by applying the DES algorithm thrice. Although 3DES is considered more secure than DES, it is slower and less efficient than AES. Twofish and Serpent are highly secure encryption algorithms, which allow for the use of key lengths up to 256 bits and block sizes of 128 bits. Although both were finalists in the NIST AES competition, neither was chosen as the AES standard. Rijndael [2] won the competition and became AES, resulting in Twofish and Serpent being less widely adopted in practice.

Although other algorithms exist, the SM4 and Camellia block cipher algorithms applied to the Linux kernel should be thoroughly examined. The SM4 algorithm is a relatively new symmetric-key block cipher approved by the Chinese government as a standard encryption algorithm in 2012 and has also been internationally considered as the ISO standard. Camellia is a block cipher algorithm that has been adopted as the standard encryption algorithm by the Japanese government and is recommended by the European Union Agency for Network and Information Security (ENISA) as a secure and efficient alternative to AES. It has also been recognized as an international IETF standard [5]. Although, SM4 and Camellia have similar key lengths and block sizes to algorithms such as Twofish and Serpent, they are specifically optimized for use in embedded systems and resource-constrained devices. Figure 1 shows the generic versions of each code applied to the Linux kernel for Camellia and SM4 in 2006 and 2018, respectively.

The generic version of encryption and decryption performed in the software has performance limitations. One approach to resolve this problem is to use special instruction sets, specifically designed to accelerate encryption and decryption processes, such as GFNI, AES-NI, and AVX, which are supported by x86-based Intel CPUs. This not only improves the overall speed of the algorithm, but also reduces the computational load on the system, leading to lower power consumption and increased battery life in portable devices. The GFNI is an instruction set that enhances the performance of cryptographic and security applications. This provides **vgf2p8affineqb** for affine transformations and **vgf2p8affineinvqb** for affine-inverse transformations. AES-NI, introduced by Intel in 2008, is another set of instructions specifically designed to improve AES algorithm performance. This includes **vaesenc** for the AES encryption rounds, **vaesenclast** for the final encryption round, and **vaesdec** and **vaesdeclast** for decryption.

To achieve high performance in the substitution and diffusion operations, tasks should be completed within minimal CPU cycles. Parallel processing of multiple tasks reduces the required number of cycles. AVX is a set of instructions for x86 processors developed by Intel and later adopted by AMD. This allows for Single Instruction and Multiple Data (SIMD) operations, which can significantly improve the performance of specific computations. AVX also includes new instructions for floating point, integer, and data shuffling operations. However, applying AVX to optimize the performance of the block cipher algorithm requires the x86 assembly language, which is a significant challenge. The scarcity of AVX code in the kernel indicates the complexity and difficulty of its implementation. Furthermore, the implementation of an AVX-based code must be tailored to the size of the register utilized, and the application of AVX should be based on the specific characteristics of the algorithm.

III. RELATED WORK

In cryptography, the acceleration of block cipher algorithms has received considerable attention owing to the growing need for efficient and secure data encryption in various applications. One widely adopted approach for accelerating block cipher algorithms is to utilize hardware acceleration methods such as Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), and GPU acceleration. These hardware-based methods provide a high level of performance optimization and have been the subject of numerous studies [14], [15], [16], [17], [18], [19], [20], [21], [22], [23]. In the same vein of research, the ARIA algorithm has also been the focus of optimization studies using parallel implementation with ARMv8 processors and GPUs (Nvidia GTX 3060) [24], as well as performance optimization based on low-power embedded processors (8-bit AVR microcontrollers) [25]. These studies demonstrate the increasing interest in exploring a variety of hardware-based approaches for accelerating block cipher algorithms in cryptography.

Previous research on block cipher algorithms primarily focused on achieving the best performance through hardware acceleration methods, such as FPGAs, ASICs, and GPU acceleration. However, algorithms utilizing GPUs and FPGAs are generally designed to operate within user space rather than kernel space. Consequently, it is not straightforward to leverage GPUs and FPGAs within the kernel for high-performance computation of these algorithms. In addition, most general users who utilize block cipher algorithms operate them in CPU-based environments. To implement a versatile and commercially viable block cipher algorithm, it is essential to develop an algorithm that can be seamlessly utilized by a large number of users in a Linux

environment, without performance constraints. Therefore, unlike previous studies, this research aims to optimize the performance of the ARIA algorithm by aligning it with the instruction set supported by the CPU in user environments for block cipher algorithms. Our research primarily focuses on software-based optimization techniques, specifically targeting CPU-based environments, rather than hardware-based approaches. In detail, various software-based methods achieve acceleration, including table optimization, slicing techniques, parallelism, and special instruction sets. Researchers aim to significantly enhance the performance of block cipher algorithms by combining these techniques, and numerous studies have been conducted on this topic.

AES algorithms have been extensively studied. In [18], various optimized designs and implementations of the AES algorithm are presented to improve its performance by 50 times by implementing fast algorithms, Intel AES-NI extended instruction sets, and parallel execution on CUDA and GPU. A hardware-acceleration solution for AES encryption in the Linux kernel for ARM processors has been proposed [26]. It utilizes the NEON instruction set, which enhances the performance of AES encryption on ARM processors. Another study proposed an efficient and high-performance AES-based authenticated encryption algorithm that utilizes AES-NI instructions, nonces, and optional associated data with an optimized low-area implementation in ASIC hardware [19]. In [27], a bit-slice implementation of AES is presented to improve the performance of different microprocessors by evaluating the impact of architecture and optimizing the maximum utilization of superscalars and SIMD, such as Streaming SIMD Extensions (SSE). In [28], a method to resist cold-boot attacks on disk drive encryption is proposed by implementing AES in the Linux kernel in a nonstandard form. This method uses an SSE that maintains a secret key within the processor, resulting in an acceptable performance penalty with a brief security analysis.

In addition to AES, studies related to other block cipher algorithms, such as SM4 and Camellia, are tailored to the characteristics of these algorithms. One study optimized the S-box implementation and derived the most compact representations of bit slicing to improve the performance of AES and SM4 [29]. In [30], a faster implementation of the SM4 block cipher algorithm is proposed using bit-slice technology, resulting in an average increase of 80 – 120% in the encryption and decryption speeds. In [4], the poor performance of a constant-time SM4 implementation is compared to AES and performance improvement techniques using GFNI (with AVX and AVX512) from Intel and NEON from ARM are proposed. In [31] and [32], a constant-time and efficient SM4 is achieved using AES-NI by exploiting the similarity between the GF(2^8) fields of SM4 and AES. Reference [33] presents an optimized implementation of SM4 on various microcontrollers, RISC-V processors, and ARM processors with parallel computation. A byte-sliced AES-NI/AVX implementation of Camellia, presented in [34], demonstrates exceptional performance at a rate of 5.32 cycles per byte on an Intel Sandy-Bridge processor. In addition, they experimented with different slicing techniques (e.g., bit slicing, byte slicing, and word slicing) to improve the performance of block ciphers. The performance of six block algorithms, including Camellia, was analyzed in [35]. Camellia from Libgrypt cryptography library is the second fastest common cipher after AES; however, its performance is 7 – 11 times worse than that of AES even with hardware acceleration support.

Algorithms such as those presented in [34] and [36], which are different from AES, utilize the AES-IN instruction set for other algorithms than AES. Inspired by these studies, we propose an architecture that includes a method utilizing AES-NI instructions to accelerate the ARIA. The next section explains the basic concepts of ARIA and the proposed performance improvements.

IV. ARIA ALGORITHM

A. OVERVIEW

This section introduces ARIA, following the notation used in [1]. It was jointly developed by academics, research institutes, and government agencies in South Korea and is designed to resist all types of block cipher attacks. It has been intensely evaluated for its stability, efficiency, and data processing speed by the University of Leuven in Belgium, the host of NESSIE (New European Schemes for Signatures, Integrity, and Encryption) [37]. In addition, it is designed to resist all known attacks and has a security level similar to AES. In software implementation, ARIA shows faster performance than Camellia and is comparable to AES, indicating its high level of security and efficiency [37]. The input and output sizes of ARIA were fixed at 128 bits and three key sizes were available:128, 192, and 256 bits. The number of rounds performed during encryption/decryption varies depending on the key size and is 12, 14, or 16 rounds (13, 15, or 17 rounds, including the final additional round) for 128-, 192-, or 256-bit keys, respectively.

The algorithm consists of two parts. The first part involves generating a round key for each round from a given primary key, which is performed before encryption/decryption. In this process, four 128-bit values (W_1, W_2, W_3, and W_4) are generated through a combination of the primary key, round functions (odd/even), and three fixed 128-bit values of the inverse of π. Then, as a round key generation process, the round key ek_n required for each n-th round is generated using W_1, W_2, W_3, and W_4. However, the details of the preliminary encryption/decryption, including the round key generation process, are outside the scope of this study. Hence, further details are not provided in this paper.

The second part focuses on the encryption/decryption process that involves repeated round functions. Figure 2 illustrates the logic within a single-round function of ARIA for encryption. Because the ARIA is an ISPN, each round function is the same for encryption and decryption. However, the key used in each round differed depending on the

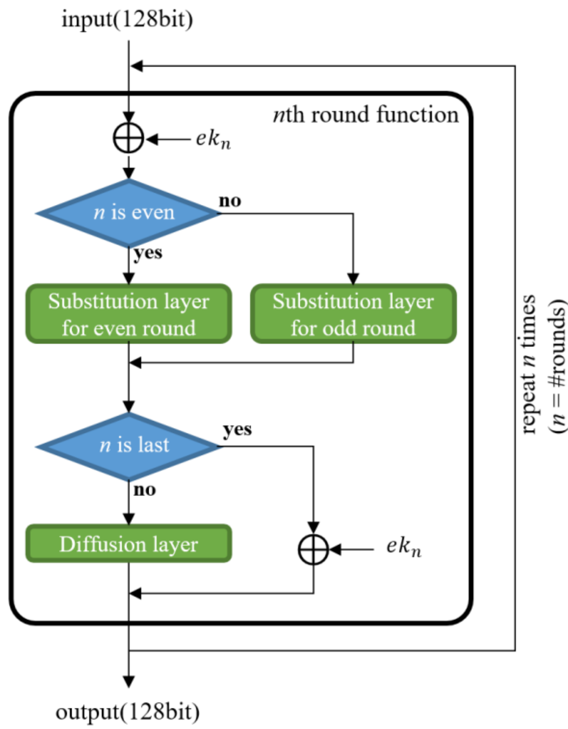


FIGURE 2. An overview of the ARIA algorithm represented in terms of the encryption process.

process. Each round function comprises a round key addition, substitution layer, and diffusion layer. The round-key addition results from XORing a 128-bit input with a 128-bit round key. The substitution and diffusion layers are the most time-consuming operations within the round function; hence, reducing the number of CPU cycles used in these layers is crucial for improving performance.

1) SUBSTITUTION LAYER

This layer uses four S-boxes (S_1, S_2, S_1^{-1} and S_2^{-1}), whose order depends on the even and odd rounds. For the even rounds, S_1, S_2, S_1^{-1} and S_2^{-1} were repeated in the given order, and similarly, for the odd rounds, S_1^{-1}, S_2^{-1}, S_1 and S_2 were repeated. Each S-box is defined as a combination of an affine transformation (e.g., A_{S_1} for S_1) and an inverse function $\mathrm{inv}(x) = x^{-1}$ over $GF(2^8)$. Here, GF represents the Galois Field. S_1 and S_2 are defined as follows:

$$S_1(x) = Ax^{-1} \oplus a = A_{S_1}(\mathrm{inv}(x)), \tag{1}$$

where,

$$\mathrm{inv}(x) = x^{-1}; \quad x \in GF(2^8),$$

$$A_{S_1} := \begin{pmatrix} 1&0&0&0&1&1&1&1\\ 1&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&1&1&1&0&0&0&1\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0\\ 0&0&0&1&1&1&1&1 \end{pmatrix} \begin{pmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 1\\ 1\\ 0\\ 0\\ 0\\ 1\\ 1\\ 0 \end{pmatrix}$$

$$\begin{aligned} S_2(x) &= Bx^{247} \oplus b \\ &= B\left(x^{-8}\right) \oplus b \\ &= B\left(x^{8} \cdot x^{-1}\right) \oplus b \\ &= BC\left(x^{-1}\right) \oplus b \\ &= Dx^{-1} \oplus b \\ &= A_{S_2}(\mathrm{inv}(x)), \end{aligned} \tag{2}$$

where,

$$A_{S_2} := \begin{pmatrix} 0&1&0&1&0&1&1&1\\ 0&0&1&1&1&1&1&1\\ 1&1&1&0&1&1&0&1\\ 1&1&0&0&0&0&1&1\\ 0&1&0&0&0&0&1&1\\ 1&1&0&0&1&1&1&0\\ 0&1&1&0&0&0&1&1\\ 1&1&1&1&0&1&1&0 \end{pmatrix} \begin{pmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 0\\ 1\\ 0\\ 0\\ 0\\ 1\\ 1\\ 1 \end{pmatrix}$$

S_1^{-1} and S_2^{-1} are also defined as:

$$S_1^{-1}(x) = (Ex \oplus e)^{-1} = \mathrm{inv}\left(A_{S_1^{-1}}(x)\right), \tag{3}$$

where,

$$A_{S_1^{-1}} := \begin{pmatrix} 0&0&1&0&0&1&0&1\\ 1&0&0&1&0&0&1&0\\ 0&1&0&0&1&0&0&1\\ 1&0&1&0&0&1&0&0\\ 0&1&0&1&0&0&1&0\\ 0&0&1&0&1&0&0&1\\ 1&0&0&1&0&1&0&0\\ 0&1&0&0&1&0&1&0 \end{pmatrix} \begin{pmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 1\\ 0\\ 1\\ 0\\ 0\\ 0\\ 0\\ 0 \end{pmatrix}$$

$$S_2^{-1}(x) = (Fx \oplus f)^{-1} = \mathrm{inv}\left(A_{S_2^{-1}}(x)\right), \tag{4}$$

where,

$$A_{S_2^{-1}} := \begin{pmatrix} 0&0&0&1&1&0&0&0\\ 0&0&1&0&0&1&1&0\\ 0&0&0&0&1&0&1&0\\ 1&1&1&0&0&0&1&1\\ 1&1&1&0&1&1&0&0\\ 0&1&1&0&1&0&1&1\\ 1&0&1&1&1&1&0&1\\ 1&0&0&1&0&0&1&1 \end{pmatrix} \begin{pmatrix} x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 0\\ 0\\ 1\\ 1\\ 0\\ 1\\ 0\\ 0 \end{pmatrix}$$

From a practical perspective, there are two approaches for obtaining the results of S-boxes: using pre-calculated tables and calculating them on demand (i.e., using the on-the-fly method). To elaborate, considering that S-box outputs 8-bits

for every 8-bit input, it can be stored as a Lookup Table (LUT) by pre-calculating and mapping 256 values (2^8). Alternatively, S-box results can be calculated as required. An S-box is generally implemented as an LUT to simplify the implementation process and achieve faster processing time. However, this method requires more resources, such as memory, and does not allow for parallel processing techniques such as SIMD.

2) DIFFUSION LAYER

The results of the 16 S-boxes (16 bytes) were used as input for the diffusion layer. As described in [1], ARIA supports 8- and 32-bit implementations. In a basic 8-bit processor implementation, the input is multiplied by a 16×16 binary matrix in $GF\left(2^8\right)^{16}$. Moreover, ARIA provides implementations of the diffusion layer for 32-bit processors, not just for 8-bit processors [1].

$$M_1 \cdot P \cdot M_1 \cdot M, \tag{5}$$

where

$$M_1 = \begin{pmatrix} I & I & I & 0 \\ I & 0 & I & I \\ I & I & 0 & I \\ 0 & I & I & I \end{pmatrix}, \quad P = \begin{pmatrix} I & 0 & 0 & 0 \\ 0 & P_1 & 0 & 0 \\ 0 & 0 & P_2 & 0 \\ 0 & 0 & 0 & P_3 \end{pmatrix},$$

$$P_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad P_2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix},$$

$$P_3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \quad M = \begin{pmatrix} T & 0 & 0 & 0 \\ 0 & T & 0 & 0 \\ 0 & 0 & T & 0 \\ 0 & 0 & 0 & T \end{pmatrix},$$

$$T = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

In the 8-bit processor implementation, the result can only be obtained after all the calculations are performed in the diffusion layer. However, the 16 bytes obtained from (1) or (2) can be partially calculated using (5), enabling parallel processing such as SIMD, and contributing to faster processing speeds.

B. ARIA IMPLEMENTATION FOR PERFORMANCE IMPROVEMENT

This section describes the steps involved in porting the ARIA encryption algorithm to a Linux kernel. We followed the guidelines outlined in [38] and implemented the ARIA code as a module within the cryptographic subsystem of the Linux kernel, as shown in Fig. 1. The implementation is based on a 32-bit architecture.

1) ACCELERATION OF THE SUBSTITUTION LAYER

We utilized AES-NI, GFNI, and AVX instructions on Intel x86-based CPUs to improve performance. Specifically, we used the **vgf2p8affineqb** instruction for affine transformations such as $A_{S_1^{-1}}$ and $A_{S_2^{-1}}$, and **vgf2p8affineinvqb** for inverse affine transformations such as (1) and (2). These instructions enabled us to efficiently perform inverse transformations and affine calculations. GFNI support is becoming increasingly widespread; a growing number of CPUs are being equipped with this feature. However, for CPUs without GFNI support, alternative methods for performing Steps (1)–(4) must be provided. AES-NI instructions are slower than GFNI for our use-case but can be used as an alternative for S-box operations.

As ARIA has the same S-boxes (S_1 and S_1^{-1}) as AES, **vaesenclast** and **vaesdeclast** can be used to compute S_1 and S_1^{-1}, respectively. However, it is important to consider the following when using these instructions: Both **vaesenclast** and **vaesdeclast** contain additional operations for AES round processing, including the **ShiftRows** operation, which shifts rows. As ARIA does not require **ShiftRows**, if **InvShiftRows** is performed on the results from **vesenclast** and **vesdeclast**, we can obtain pure S_1 and S_1^{-1} results as follows.

$$f_{\text{InvShiftRows}}\left(f_{\text{vaesenclast}}(x)\right) = \widehat{A}_{S_1}\left(\widehat{\text{inv}}(x)\right), \tag{6}$$

$$f_{\text{InvShiftRows}}\left(f_{\text{vaesdeclast}}(x)\right) = \widehat{\text{inv}}\left(\widehat{A}_{S_1^{-1}}(x)\right). \tag{7}$$

$\widehat{A}(x)$ is defined as $A(x)$ concatenated 16 times, $A(x) \,||\, \ldots$ (16 times $A(x)$), and this definition is used to capture the relationship between 16-byte and 8-bit operations.

The second consideration is the calculation of S_2 and S_2^{-1} using only ARIA, which is not included in AES. Because there are no instructions supporting S_2 and S_2^{-1}, (3) and (4) must be implemented. For (3), $\text{inv}(x)$ must be calculated followed by A_{S_2}. Similarly, for (4), $A_{S_2^{-1}}$ must be calculated followed by $\text{inv}(x)$. We confirmed that calculating inv() using **vaesenclast** and **vaesdeclast** consumed fewer cycles than directly calculating inv(). For (2), $\text{inv}(x)$ was obtained using (6).

$$\widehat{A}_{S_1^{-1}}\left(f_{\text{InvShiftRows}}\left(f_{\text{vaesenclast}}(x)\right)\right) = \widehat{\text{inv}}(x) \tag{8}$$

By performing $\widehat{A}_{S_2}$ with (8), $\widehat{S}_2(x)$ can be obtained as follows:

$$\begin{aligned} &\widehat{A}_{S_2}\left(\widehat{A}_{S_1^{-1}}\left(f_{\text{InvShiftRows}}\left(f_{\text{vaesenclast}}(x)\right)\right)\right) \\ &= \widehat{A}_{S_2}\left(\widehat{A}_{S_1^{-1}}\left(\widehat{A}_{S_1}\left(\text{inv}(x)\right)\right)\right) \\ &= \widehat{A}_{S_2}\left(\widehat{\text{inv}}(x)\right) \\ &= \widehat{S}_2(x) \end{aligned} \tag{9}$$

Unlike (8) and (9), to obtain the result of (4), $A_{S_2^{-1}}$ and A_{S_1} are sequentially performed:

$$\widehat{A}_{S_1}\left(\widehat{A}_{S_2^{-1}}(x)\right) \tag{10}$$

In addition, if A_{S_1} is eliminated using (7), $S_2^{-1}(x)$ is obtained as follows:

$$f_{\text{InvShiftRows}}\left(f_{\text{vaesdeclast}}\left(\widehat{A}_{S_1}\left(\widehat{A}_{S_2^{-1}}(x)\right)\right)\right)$$

$$
\begin{aligned}
&= \widehat{\text{inv}}\left(\widehat{A}_{S_1^{-1}}\left(\widehat{A}_{S_1}\left(\widehat{A}_{S_2^{-1}}(x)\right)\right)\right) \\
&= \widehat{\text{inv}}\left(\widehat{A}_{S_2^{-1}}(x)\right) \\
&= \widehat{S}_2^{-1}(x)
\end{aligned} \tag{11}
$$

AES-NI instructions were used to reduce the number of CPU cycles. Two affine transformations and one AES-NI instruction (either **vaesenclast** or **vaesdeclast**) are performed, as shown in Equations (9) and (11). For further optimization, the two affine transformations can be combined. To calculate S_2, the two transformations A_{S_2} and $A_{S_1^{-1}}$ in Equation (9) are combined and referred to as $\overline{A_{S_2}}$. The results are as follows:

$$
\begin{aligned}
\overline{A_{S_2}} &:= A_{S_2} A_{S_1^{-1}} \\
&= D(Ex \oplus e) \oplus b \\
&= DEx \oplus (De \oplus b) \\
&= DEx \oplus g \\
&= Gx \oplus g \\
&= \begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1
\end{pmatrix}
\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \\
&\quad \oplus \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.
\end{aligned} \tag{12}
$$

Similarly, for calculating S_2^{-1}, A_{S_1} and $A_{S_2^{-1}}$ in (9) were combined and referred to as $\overline{A_{S_2^{-1}}}$, and the following results were obtained:

$$
\begin{aligned}
\overline{A_{S_2^{-1}}} &:= A_{S_1} A_{S_2^{-1}} \\
&= A(Fx \oplus f) \oplus a \\
&= AFx \oplus Df \oplus a \\
&= AFx \oplus g \\
&= Hx \oplus h \\
&= \begin{pmatrix}
1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}
\end{aligned}
$$

$$
\oplus \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{13}
$$

Therefore, equations (9) and (11) can be redefined to require fewer cycles.

$$
\overline{\widehat{A}}_{S_2}\left(f_{\text{InvShiftRows}}\left(f_{\text{vaesenclast}}(x)\right)\right) = \widehat{S}_2(x), \tag{14}
$$

$$
f_{\text{InvShiftRows}}\left(f_{\text{vaesdeclast}}\left(\overline{\widehat{A}_{S_2^{-1}}}(x)\right)\right) = \widehat{S}_2^{-1}(x) \tag{15}
$$

2) PARALLEL PROCESSING WITH AVX

In the generic version of ARIA, we utilized parallel processing to improve performance beyond the limits of GFNI or AES-NI. As one of the SIMD instruction sets, AVX [9] supports the simultaneous computation of multiple values using a single instruction. AVX primarily uses 128-bit registers and the number of usable registers differs for each AVX version. The AVX, AVX2, and AVX512 versions process 16, 32, and 64 blocks in parallel, respectively. It should be noted that while multiple values can be calculated simultaneously, calculation of each value must require the same operation.

Figure 3 shows the parallel processing of 16 128-bit inputs in a single round using AVX. AVX can store data in 16 128-bit registers. Therefore, the first step is to store sequentially incoming 16 inputs to perform encryption or decryption in parallel.

Once 16 inputs were gathered, they were sliced into bytes. The same S-box operation was performed on a single 128-bit register in the substitution layer. The byte-sliced 16 inputs are loaded into memory, and as shown in Fig. 3, the top eight data points are used to calculate the substitution layer and M matrix of the diffusion layer (diff_m block in Fig. 3). Whereas, the remaining eight data points are calculated later in the same process. Only eight data points were processed simultaneously, because there were 16 AVX registers to divide and contain two operated values. The first operation in a round function is round key addition. To support byte slicing, each round key is filled with a byte of a particular index in the relevant round key. After round key addition, the related S-box operations were performed on each 128-bit slice of the data. After operations (6) are performed, only those that require S_2 operations are multiplied by $\overline{A_{S_2}}$, related to (14). On the other hand, only those that require S_2^{-1} operations are multiplied by $\overline{A_{S_2^{-1}}}$ before operations (7) are performed, related to (15). An LUT for the S-box (8)-bit input) can be implemented using **vpshufb** of the AVX. As suggested in [34], we constructed two LUT maps by separating one affine transformation into high and low LUT maps as follows:

$$
\overline{A_{S_2}} = \overline{A_{S_2}^{\text{low}}}(x_0, x_1, x_2, x_3) \oplus \overline{A_{S_2}^{\text{high}}}(x_4, x_5, x_6, x_7) \tag{16}
$$

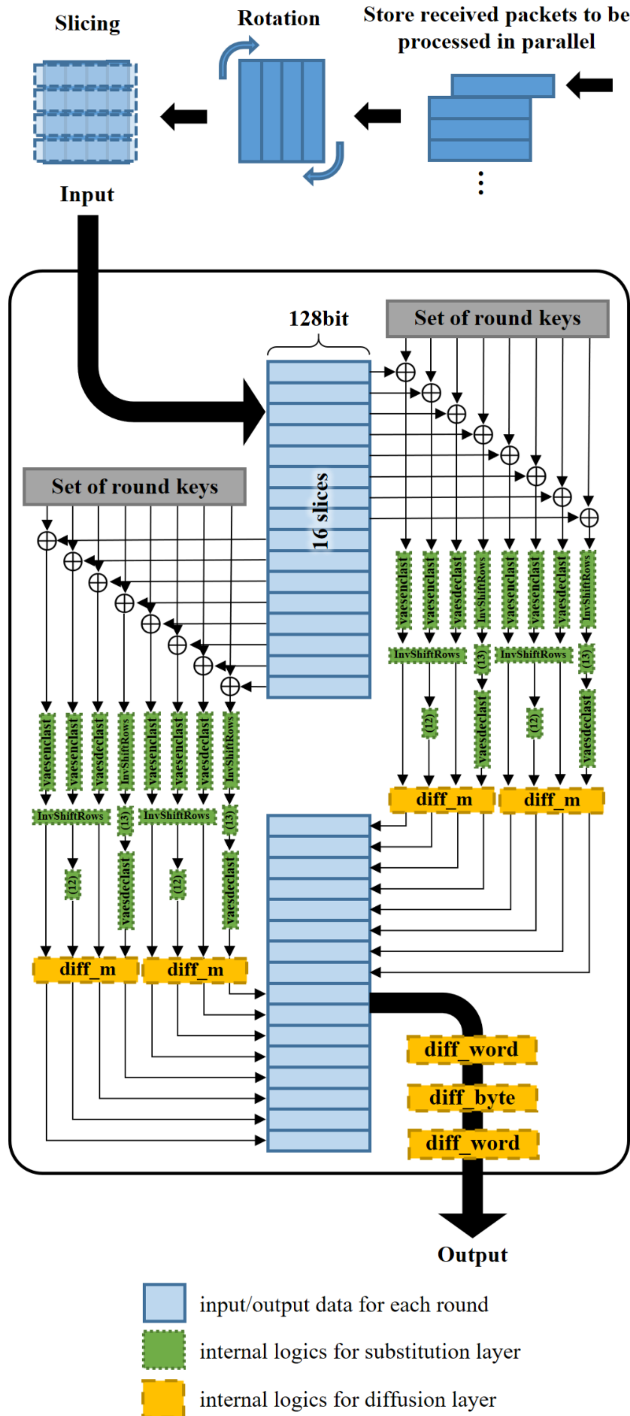


FIGURE 3. Detailed operations inside one round function to perform parallel processing using AVX.

$$A_{\overline{S_2^{-1}}} = A_{\overline{S_2^{-1}}}^{\text{low}}(x_0, x_1, x_2, x_3) \oplus A_{\overline{S_2^{-1}}}^{\text{high}}(x_4, x_5, x_6, x_7) \quad (17)$$

This significantly reduces the size of the maps by input into 4-bit elements. The results of $\overline{A_{S_2}}$ and $A_{\overline{S_2^{-1}}}$ can be obtained by XORing the results of the high and low maps calculated later. This completes the calculations related to the round substitution layer and the calculations related to the diffusion layer are initiated. diff_m, diff_word, and diff_byte shown in Fig. 3 represent M, M_1, and P of (5), respectively. Four diff_ms were calculated for each 32-bit, and a total of 16 slices of data were reconstructed. For the 16 pieces of 16-byte data, diff_word, diff_byte, and diff_word operations were performed sequentially. The diff_byte operation corresponding to P is implemented to skip by adjusting the position of the input byte of the last diff_word.

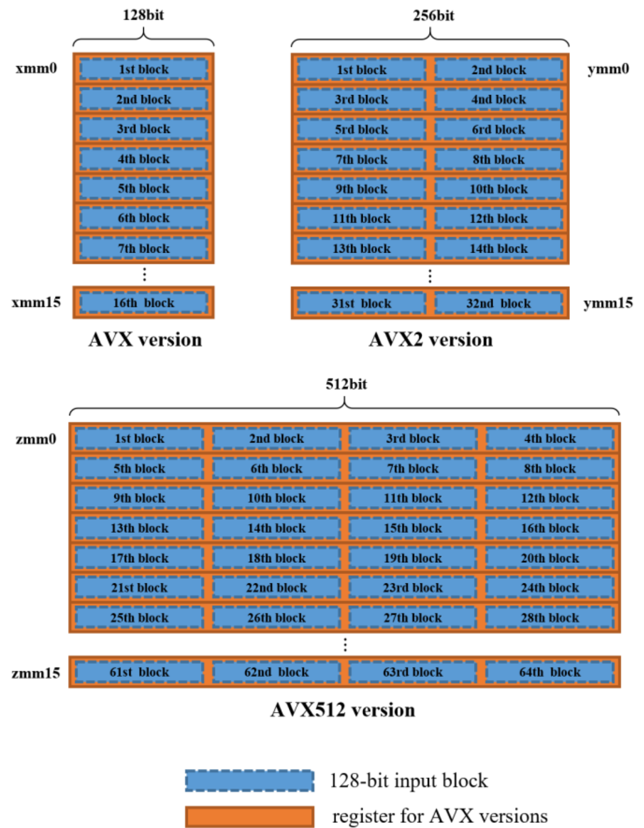


FIGURE 4. Example of receiving maximum input blocks for each version of AVX.

3) IMPLEMENTATION WITH AVX2 AND AVX512

We also implemented the AVX2 and AVX512 versions of ARIA, which can handle more input blocks in parallel than the basic AVX version. AVX2 can handle 32 blocks simultaneously with its 16 256-bit YMM registers, and AVX512 can handle 64 blocks simultaneously with its 32 512-bit ZMM registers. The number of available registers differed for each AVX version.

As shown in Fig. 4, the basic AVX version can process input blocks equal to 16 128-bit XMM registers. However, because of the need for key storage and processing or temporary data storage, the input data must be divided into halves, each half consisting of eight 16-byte blocks.

Each register used by AVX2 accommodates two 128-bit blocks. However, similar to the basic AVX version, half of the input data were processed before and the remainder is

processed subsequently. If the processed input data were $<$ 256 bytes but greater than 128 bytes, they were processed using the basic AVX version. If the processed input data were less than 128 bytes in size, they were processed using a generic version.

The AVX512 version of the ARIA utilizes 16 of the 32 available 512-bit ZMM registers that accommodate the four blocks. This approach minimizes the memory storage and loading overhead by processing half of the input data and using some of the registers for key or temporary data storage. Although AVX512 can accommodate up to 2048 bytes of data, processing 2048 bytes of data simultaneously is rare. Given that most of the latest CPUs supporting AVX512 also provide GFNI, the implementation of AES-NI for the AVX512 version of ARIA has been deemed unnecessary. Thus, we have only included the implementation of GFNI in this version.

The AVX-based versions of the ARIA algorithm use parallel processing with a minimum data size that can be processed in parallel. If the data to be processed is less than this minimum size, a hybrid method is used that partially utilizes parallel processing, and the remaining data is processed using the generic algorithm. The SM4 algorithm-based AVX code references the cumulative input size and pads the data if necessary to enable parallel processing. The same approach can be adopted for ARIA to process data in parallel based on the cumulative input size, resulting in no performance degradation even with different cumulative input sizes. This improvement can be considered for future work.

4) CONSTANT-TIME IMPLEMENTATION

Maintenance of constant-time execution for block cipher algorithms like ARIA is crucial to ensure the security and robustness of the encryption process. This involves ensuring consistent execution time regardless of the input data or secret key used, preventing timing-based side-channel attacks. Such attacks exploit variations in execution time to reveal sensitive information about the encryption and decryption key. Therefore, careful evaluation of the ARIA implementation is necessary to ensure constant-time execution and prevent such attacks, thereby strengthening the overall security and reliability of the encryption and decryption process.

Our code is designed with multiple security measures to safeguard against data-dependent execution paths that can leave systems vulnerable to timing attacks. We achieve this by avoiding data-dependent branching statements and data-dependent sub-logics such as the substitution layer and diffusion layer. Moreover, our code leverages SIMD instructions, such as **vaesenclast**, **vaesdeclast**, and **vpxor**, to enable the processing of multiple data elements simultaneously in constant-time [39]. This feature results in a fixed execution time for function calls, operations, and memory allocations that remain independent of input data. Constant-time operations like bit and XOR operations (e.g., **vpand**, **vpandn**, **vpxor**) help minimize data dependencies.

V. EXPERIMENT

In this section, we describe the performances of the generic [40], AVX [41], AVX2 [41], and AVX512 [42] versions of ARIA implemented in the Linux kernel. Our experimental results confirm that the proposed ARIA implementation is suitable for commercial applications requiring block ciphering. As outlined in the Introduction, our goal is to minimize the number of CPU cycles required when employing ARIA for encryption and decryption. Thus, the experimental results presented in this section highlight the required number of cycles, which are the key performance metrics. We present the results for all the ARIA implementations and demonstrate the results of reducing the number of affine transformations from (9) and (11) to (14) and (15). Furthermore, we compared the performance of ARIA with that of the Camellia algorithm, which has already been implemented in the Linux kernel, similar to our ARIA implementation.

A. EXPERIMENTAL SETUP

The kernel contains modules that support algorithm testing. In particular, a performance test module exists in the cryptographic subsystem, crypto/tcrypt.c [43]. The tcrypt module was introduced into the Linux kernel to provide a standardized method for evaluating the performance of different cryptographic algorithms, which can help improve the design and implementation of future algorithms. Using tcrypt, we can check the relationship between the operations and the cycles. In the block cipher algorithm, an operation indicates the process of encrypts or decrypts a single block.

Our experiments did not require high-performance equipment, as only the CPU was needed to support the AVX series with AES-NI and GFNI. The experimental setup comprised a machine equipped with an i3-12100 CPU, 16 GB of memory, and 256 GB of secondary SSD storage. The tests were conducted on Ubuntu 22.04 Linux with kernel version 6.1. Our implementation adhered to the ECB and CTR modes specified in [7]; the results are presented based on the ECB mode. Two test data block sizes, 1024 bytes and 4096 bytes, were used for encryption and decryption. Both 128-bit and 256-bit keys were used in the tests.

B. PERFORMANCE COMPARISON OF DIFFERENT IMPLEMENTATIONS OF ARIA

The ARIA implemented in this study has six versions, each designed to accommodate various instruction sets supported by a CPU. The generic version is used in the absence of both AVX and GFNI/AES-NI, or when only AVX is present. The second version utilizes AVX and AES-NI when both are available but GFNI is not. The third version prioritizes GFNI when both GFNI and AES-NI are present and utilizes AVX. The fourth version employs AVX2 and AES-NI. The fifth version utilized AVX2 and GFNI. Finally, the sixth version employs AVX512 and GFNI. Before the experiment, we expected the results to show that the generic version would exhibit the lowest performance owing to its lack of

parallel processing and reliance on sequential data processing. Among the versions supporting the AVX technology, we expected AVX512 to demonstrate the highest performance, followed by AVX2 and AVX. We anticipated that GFNI would outperform AES-NI in terms of operational efficiency, because it requires fewer cycles for the same operation.

Figure 5 compares the performance of the six AVX versions. As expected, the performance of the versions utilizing AES-NI or GFNI was significantly better than that of the generic algorithm, as confirmed by the results. In terms of speed, AVX with AES-NI, AVX with GFNI, AVX2 with AES-NI, AVX2 with GFNI, and AVX512 with GFNI were approximately 4.2, 5.3, 5.8, 8.1, and 10.6 times faster than the generic version, respectively. Furthermore, for AVX, using GFNI improved the performance by approximately 21.2% compared to AES-NI, and for AVX2, using GFNI improved the performance by 26.8% to 31.3% compared to the use of AES-NI. These results indicate that a CPU that supports AVX, AES-NI, or GFNI is crucial to achieve maximum performance with ARIA.

C. PERFORMANCE COMPARISON OF OPTIMIZED AFFINE TRANSFORMATIONS USING AES-NI

The substitution layer is one of the time-consuming operations within the round function. Therefore, we conducted an analysis to understand the specific operations and their cycle consumption. In this section, we confirm that the affine transformation is the most cycle-intensive operation through the experimental results. By leveraging the AES-NI instruction set, dedicated to AES, we can able to improve performance by reducing the cycle count of the affine transformation. We compare the performance before and after optimization by reducing the number of affine transformations when AES-NI is used. We minimized the affine transformation computations required to compute S-boxes S_2 and S_2^{-1}, as expressed in (12) and (13) of Section IV-B.

Table 1 compares the number of cycles for the operations performed in the ARIA_SBOX_8WAY function before and after optimizing the affine transformation. Based on AVX, each round processed 16 blocks, and the ARIA_SBOX_8WAY function, which calculated eight blocks simultaneously, was called twice per round. Table 1 shows that the affine transformation is the most cycle-intensive operation in this function. Before optimization, an affine transformation was performed twice on four of the eight blocks to calculate S_2 and S_2^{-1}. Optimization was performed once for each relevant block. After optimization, the number of cycles decreased by 32.2%. Figure 6 shows the proportion of operations within the ARIA_SBOX_8WAY function. Before optimization, the proportion of cycles for the two affine transformations was approximately 54%; after optimization, the proportion of cycles for a single affine transformation decreased to approximately 40%. Hence, opti-

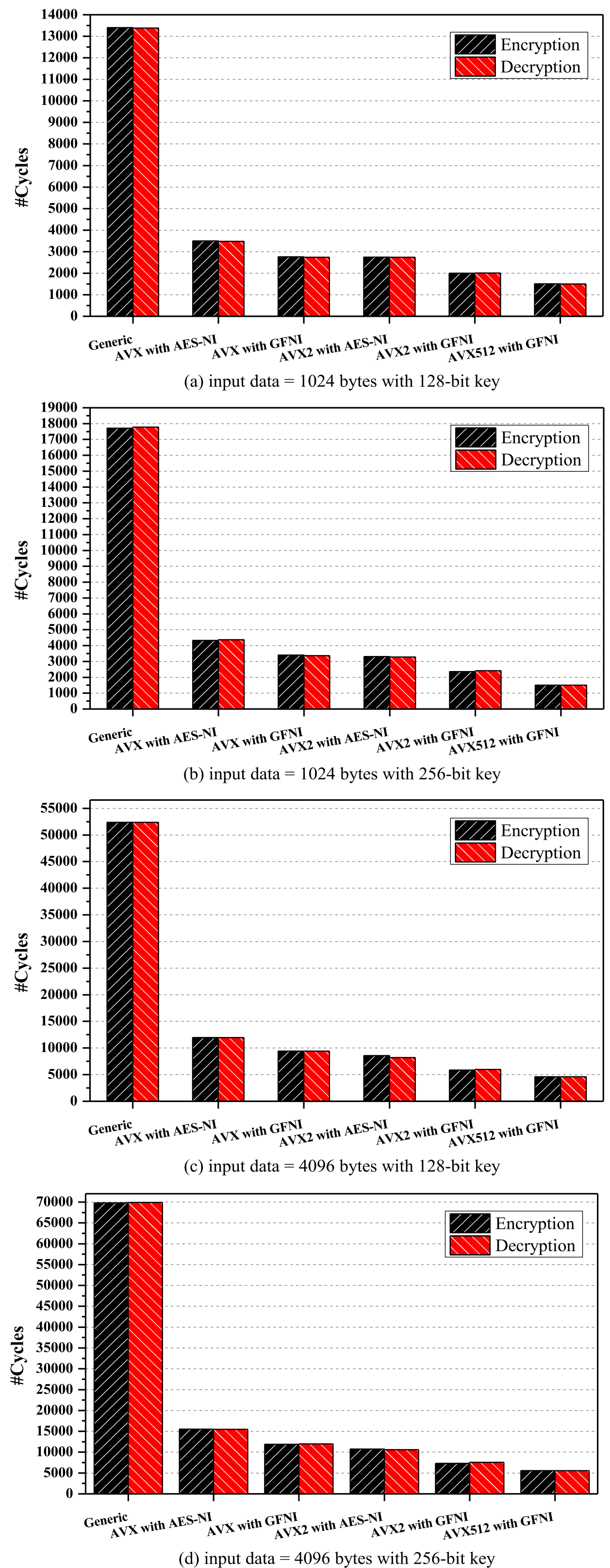


FIGURE 5. Performance Comparison of six ARIA Implementations: generic, AVX with AES-NI, AVX with GFNI, AVX2 with AES-NI, AVX2 with GFNI, and AVX512 with GFNI, considering an example of receiving maximum input blocks for each version of AVX.

TABLE 1. The number of cycles in the ARIA_SBOX_8WAY function based on AVX.

Operation	#cycles per operation (A)	Before Optimization		After Optimization	
		#Operations called (B)	Cycles for all operations (AB)	#Operations called (D)	#Cycles for all operations (AD)
AES-NI for Encryption (**vaesenclast**)	1	4	4	4	4
AES-NI for Decryption (**vaesenclast**)	1	4	4	4	4
AES Inverse Shift Rows (**vpshufb**)	1	8	8	8	8
Affine Transformation (filter_8bit)	4	8	32	4	16
Etc. (**vpxor**, **vmovdqa**)	-	-	11	-	8
Total Cycles	-	-	59	-	40
Total Cycles per one Round	-	-	118	-	80

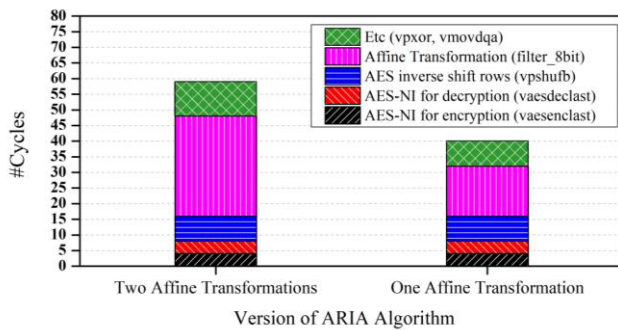
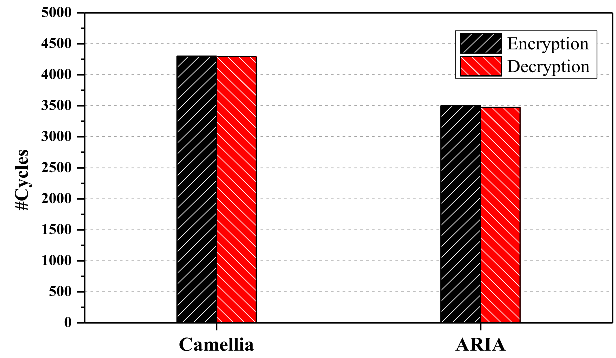


FIGURE 6. Comparison before and after optimizing the number of affine transformations when using AES-NI.

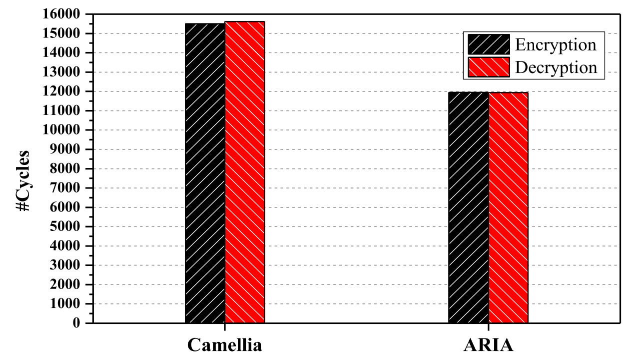
mizing the affine transformation is crucial for accelerating ARIA using AES-NI.

D. PERFORMANCE COMPARISON OF ARIA AND CAMELLIA

While comparing the number of cycles between block algorithms with different characteristics may not always accurately reflect performance differences, our findings demonstrate that Camellia, a widely used block cipher algorithm in the Linux kernel, and our implementation of ARIA exhibit comparable performance. This indicates the validity of our implementation and suggests that it does not demonstrate inferior performance when compared. We evaluate the performance matrix between algorithms based on the number of cycles. By comparing the number of cycles required for a single encryption or decryption operation on a given dataset, we can assess performance differences between algorithms. To ensure consistent performance measurements in the same environment, we measure performance using typical data block sizes of 1024 bytes and 4096 bytes in a CPU environment that supports AVX and AES-NI. Additionally, during the experiments, the key size for each algorithm is



(a) input data = 1024 bytes with 128-bit key



(a) input data = 4096 bytes with 128-bit key

FIGURE 7. Performance comparison of ARIA and Camellia Algorithms.

fixed at 128 bits. The metric values for each algorithm are presented as the average of results obtained from conducting the experiments more than 1000 times in the same environment, aiming to reduce variances.

According to the results in Fig. 7, ARIA was 1.24 times faster than Camellia when tested with 1024-byte data blocks and 1.3 times faster with 4096-byte data blocks. The results show that ARIA we implemented is well-optimized for performance, similar to other block algorithms in the Linux kernel.

VI. CONCLUSION

In this paper, we present a method for implementing a standardized ARIA algorithm in a crypto subsystem. Our implementation of the algorithm comprised a generic version developed in C, as well as several optimized versions that leveraged parallel processing through the use of AVX instructions. Specifically, for systems that support AES-NI, we present an optimized affine transformation approach to accelerate the processing of two unique S-boxes among the four ARIA S-boxes. In addition, when AES-NI and GFNI were supported simultaneously, the faster GFNI was used to accelerate ARIA. In conclusion, our experimental results demonstrate that ARIA can deliver optimal performance in encryption and decryption tasks depending on the instruction set supported by the CPU. The future work of our research

is to mitigate the performance loss due to Generic-based processing of the remaining data when the data to be processed is less than the minimum data size for parallel processing. A potential strategy for this is to refer to the cumulative amount of given data and insert padding into the input data to make it a suitable size for parallel processing with AVX. We expect that the ARIA applied to the kernel will be utilized effectively when a block cipher algorithm is required.

ACKNOWLEDGMENT

The authors would like to thank the Cloud Team, Kakao Enterprise, for the informative discussions.

REFERENCES

[1] D. Kwon, "New block cipher: ARIA," in *Proc. Int. Conf. Inf. Secur. Cryptol.* Berlin, Germany: Springer, 2003, pp. 432–445.
[2] J. Daemen and V. Rijmen, *The Design of Rijndael: AES—The Advanced Encryption Standard*. Berlin, Germany: Springer, 2002.
[3] *Information Technology-Security Techniques-Encryption Algorithms-Part 3: Block Ciphers-Amendment 1: SM4*, document ISO/IEC 18033-3:2010/AMD 1:2021, Accessed: Nov. 27, 2021. [Online]. Available: https://www.iso.org/standard/81564.html
[4] W. Guo, "Efficient constant-time implementation of SM4 with Intel GFNI instruction set extension and arm NEON coprocessor," Cryptol. ePrint Arch., 2022, pp. 1–14. [Online]. Available: https://eprint.iacr.org/2022/1154
[5] M. Matsui, J. Nakajima, and S. Moriai, *A Description of the Camellia Encryption Algorithm*, document RFC 3713, 2004.
[6] K. Aoki, "Camellia: A 128-bit block cipher suitable for multiple platforms—Design and analysis," in *Proc. Int. Workshop Sel. Areas Cryptogr.* Berlin, Germany: Springer, 2000, pp. 39–56.
[7] J. Lee, *A Description of the ARIA Encryption Algorithm*, document RFC 5794, 2010.
[8] W. Kim, *Addition of the ARIA cipher suites to Transport Layer Security (TLS)*, document RFC 6209, 2011.
[9] *Intel R Advanced Vector Extensions Programming Reference*. Accessed: Feb. 9, 2023. [Online]. Available: https://software.intel.com/sites/default/files/4f/5b/36945
[10] K. Akdemir, "Breakthrough AES performance with Intel AES new instructions," Intel Corp., Santa Clara County, CA, USA, White Paper 217, Jun. 2010.
[11] D. Towner, "Galois field new instructions (GFNI)," Intel Corp., Santa Clara County, CA, USA, White Paper, Jul. 2021.
[12] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Boca Raton, FL, USA: Chapman & Hall/CRC, 2008.
[13] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 5th ed. Boston, MA, USA: Prentice-Hall, 2010.
[14] A. J. Elbirt, "An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 4, pp. 545–557, Aug. 2001.
[15] U. Farooq and M. F. Aslam, "Comparative analysis of different AES implementation techniques for efficient resource usage and better performance of an FPGA," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 29, no. 3, pp. 295–302, Jul. 2017.
[16] M. Mozaffari-Kermani and A. Reyhani-Masoleh, "Efficient and high-performance parallel hardware architectures for the AES-GCM," *IEEE Trans. Comput.*, vol. 61, no. 8, pp. 1165–1178, Aug. 2012.
[17] S. Morioka and A. Satoh, "An optimized S-box circuit architecture for low power AES design," in *Proc. Int. Workshop Cryptogr. Hardw. Embed. Sys.* Berlin, Germany: Springer, 2003, pp. 1–15.
[18] G.-L. Guo, Q. Qian, and R. Zhang, "Different implementations of AES cryptographic algorithm," in *Proc. IEEE IEEE 17th Int. Conf. High Perform. Comput. Commun. 7th Int. Symp. Cyberspace Saf. Secur., IEEE 12th Int. Conf. Embedded Softw. Syst.*, Aug. 2015, pp. 1848–1853.
[19] A. Bogdanov, "ALE: AES-based lightweight authenticated encryption," in *Proc. Int. Workshop Fast Softw. Encryption*. Berlin, Germany: Springer, 2014, pp. 1–20.
[20] O. D. Arne, "Fast software AES encryption," in *Proc. Fast Softw. Encryption, 17th Int. Workshop (FSE)*. Seoul, South Korea: Springer, Feb. 2010.
[21] B. Rashidi, "Low-cost and two-cycle hardware structures of PRINCE lightweight block cipher," *Int. J. Circuit Theory Appl.*, vol. 48, no. 8, pp. 1289–1303, Aug. 2020.
[22] B. Rashidi, "Flexible structures of lightweight block ciphers PRESENT, SIMON and LED," *IET Circuits, Devices Syst.*, vol. 14, no. 3, pp. 314–324, Mar. 2020.
[23] B. Rashidi, "Flexible and high-throughput structures of camellia block cipher for security of the Internet of Things," *IET Comput. Digital Techn.*, vol. 15, no. 3, pp. 156–166, May 2021.
[24] S. Eum, H. Kim, H. Kwon, M. Sim, G. Song, and H. Seo, "Parallel implementations of ARIA on ARM processors and graphics processing unit," *Appl. Sci.*, vol. 12, no. 23, p. 12246, Nov. 2022.
[25] H. Seo, H. Kwon, H. Kim, and J. Park, "ACE: ARIA-CTR encryption for low-end embedded processors," *Sensors*, vol. 20, no. 13, p. 3788, Jul. 2020.
[26] A. Biesheuvel. (2017). *Accelerated AES for Arm64 Linux Kernel*. Accessed: Mar. 5, 2023. [Online]. Available: https://www.linaro.org/blog/accelerated-aes-for-the-arm64-linux-kernel/
[27] C. Rebeiro, D. Selvakumar, and A. S. L. Devi, "Bitslice implementation of AES," in *Proc. 5th Int. Conf.*, 2006, Suzhou, China: Springer, Dec. 2006, pp. 203–212.
[28] T. Müller, A. Dewald, and F. C. Freiling, "AESSE: A cold-boot resistant implementation of AES," in *Proc. 3rd Eur. Workshop Syst. Secur.*, Apr. 2010, pp. 1–10.
[29] R. Xu, Z. Xiang, D. Lin, S. Zhang, D. He, and X. Zeng, "High-throughput block cipher implementations with SIMD," *J. Inf. Secur. Appl.*, vol. 70, Nov. 2022, Art. no. 103333.
[30] J. Zhang, M. Ma, and P. Wang, "Fast implementation for SM4 cipher algorithm based on bit-slice technology," in *Proc. Smart Comput. Commun., 3rd Int. Conf. SmartCom*, Tokyo, Japan: Springer, 2018, pp. 1–15.
[31] M. -J. O. Saarinen. (2019). *SM4NI*. [Online]. Available: https://github.com/mjosaarinen/sm4ni Accessed: Jul. 8, 2022.
[32] J. Kivilinna. (2020). *Sm4-Aesni-Avx2-Amd64.S*. Accessed: Mar. 5, 2023. [Online]. Available: https://git.gnupg.org/cgi-bin/gitweb.cgi?p=libgcrypt.git;a=commit;h=35a78eb248d6bacd2a58477a122a0020d796ce63
[33] H. Kwon, H. Kim, S. Eum, M. Sim, H. Kim, W.-K. Lee, Z. Hu, and H. Seo, "Optimized implementation of SM4 on AVR microcontrollers, RISC-V processors, and ARM processors," *IEEE Access*, vol. 10, pp. 80225–80233, 2022.
[34] J. Kivilinna, "Block ciphers: Fast implementations on X86–64 architecture," M.S. thesis, Faculty Sci., Dept. Inf. Process. Sci., Univ. Oulu, Oulu, Finland, 2013.
[35] M. Alrowaithy and N. Thomas, "Investigating the performance of C and C++ cryptographic libraries," in *Proc. 12th EAI Int. Conf. Perform. Eval. Methodologies Tools*, Mar. 2019, pp. 1–15.
[36] R. Benadjila, "The Intel AES instructions set and the SHA-3 candidates," in *Proc. Int. Conf. Theory Appl. Cryptol. Info. Secur.* Berlin, Germany: Springer, 2009, pp. 162–178.
[37] A. Biryukov, "Security and performance analysis of ARIA," KU Leuven, Leuven, Belgium, Tech. Rep. ESAT/SCD-COSIC 3, 2004, p. 4.
[38] *OpenSSL Code*. Accessed: Jul. 26, 2023. [Online]. Available: https://github.com/openssl/openssl/blob/master/crypto/aria/aria.c
[39] S. Gueron, "Intel advanced encryption standard (AES) new instructions set," Intel Corp., Santa Clara, CA, USA, Tech. Rep. 18, 2010.
[40] *Generic Version of ARIA Algorithm*. Accessed: Feb. 9, 2023. [Online]. Available: https://github.com/torvalds/linux/blob/master/crypto/aria_generic.c
[41] *AVX Related Version of ARIA Algorithm*. Accessed: Feb. 9, 2023. [Online]. Available: https://github.com/torvalds/linux/blob/master/arch/x86/crypto/aria-aesni-avx-asm_64.S
[42] *AVX512 Version of ARIA Algorithm*. Accessed: Feb. 9, 2023. [Online]. Available: https://github.com/torvalds/linux/blob/master/arch/x86/crypto/aria-gfni-avx512-asm_64.S
[43] *Tcrypt (The Linux Kernel Cryptography Testing Module)*. Accessed: Feb. 9, 2023. [Online]. Available: https://github.com/torvalds/linux/blob/master/crypto/tcrypt.c

TAE-HEE YOO received the B.S. degree in computer engineering from Daegu University, Gyeongsan, South Korea, in 2015. He started his career as a Firewall Software Engineer with Future Systems. Since 2020, he has been a Cloud Networking Software Engineer with Kakao Enterprise Corporation. He has also been contributing to Netfilter, Virtual Interface, TC, BPF, XDP, and many other areas in the Linux kernel networking stack opensource project.

JUSSI KIVILINNA received the B.Sc. degree in chemistry and the M.Sc. degree in information processing science from the University of Oulu, Finland, in 2011 and 2013, respectively. He is an experienced embedded software architect with a passion for open-source software. He has contributed to several open-source projects, including the Linux kernel, libgcrypt, and GnuPG. He also has extensive experience with IoT technologies from his time as a Software Architect with Haltian, until 2020. He is currently an Embedded Software Architect with Oura Health, where he develops innovative solutions to improve people's health and well-being.

CHOONG-HEE CHO (Member, IEEE) received the B.S. degree in computer science from Sahmyook University, Seoul, South Korea, in 2010, the M.S. and Ph.D. degrees in information and communication network technology from the Korea University of Science and Technology (UST), Daejeon, in 2019, and the Ph.D. degree in telecommunication networks and optimization. He was a Postdoctoral Researcher with the Korea Advanced Institute of Science and Technology (KAIST), until 2020. From 2020 to 2022, he was with the Cloud Development Department, Kakao Enterprise Corporation. He is currently an Assistant Professor with the Department of Computer Science, Sahmyook University, conducting research on cloud networks and data center architecture.