IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 2, pp. 115–154. DOI:10.46586/tches.v2023.i2.115-154

Areion: Highly-Efficient Permutations and Its Applications to Hash Functions for Short Input

Takanori Isobe[1,2], Ryoma Ito[2], Fukang Liu[1], Kazuhiko Minematsu[3], Motoki Nakahashi[1], Kosei Sakamoto[1] and Rentaro Shiba[4]

[1] University of Hyogo, Kobe, Japan.
takanori.isobe@ai.u-hyogo.ac.jp,liufukangs@gmail.com,
motoki.n1998@gmail.com,k.sakamoto0728@gmail.com
[2] National Institute of Information and Communications Technology, Koganei, Japan.
itorym@nict.go.jp
[3] NEC Corporation, Kawasaki, Japan.
k-minematsu@nec.com
[4] Mitsubishi Electric Corporation, Kamakura, Japan.
shiba.rentaro@dc.mitsubishielectric.co.jp

Abstract. In the real-world applications, the overwhelming majority of cases require hashing with relatively short input, say up to 2K bytes. The length of almost all TCP/IP packets is between 40 to 1.5K bytes, and the maximum packet lengths of major protocols, *e.g.*, Zigbee, Bluetooth low energy, and Controller Area Network (CAN) are less than 128 bytes. However, existing schemes are not well optimized for short input. To bridge the gap between real-world needs (in future) and limited performances of state-of-the-art hash functions for short input, we design a family of wide-block permutations Areion that fully leverages the power of AES instructions, which are widely deployed in many devices. As its applications, we propose several hash functions. Areion significantly outperforms existing schemes for short input and even competitive to relatively long message. Indeed, our hash function is surprisingly fast, and its performance is less than 3 cycles/byte in the latest Intel architecture for any message size. Especially, it is about 10 times faster than existing state-of-the-art schemes for short message up to around 100 bytes, which are most widely-used input size in real-world applications, on both the latest CPU architectures (IceLake, Tiger Lake, and Alder Lake) and mobile platforms (Pixel 6 and iPhone 13).

Keywords: Short message · AES instruction · hash function · beyond 5G · IoT

1 Introduction

1.1 Background

In real-world communication environments, the overwhelming majority of cases require hashing with relatively short input, say up to 2K bytes. It is common knowledge that "real-world" TCP/IP packet length is biased towards short packets [MKZ+17], as implemented by the standard benchmark method (Internet Mix[1] and the variants) for Internet routers etc. Packet sizes on the Internet generally follow a bimodal distribution, where 44% of packets are between 40 and 100 bytes long, and 37% are between 1400 and 1500 bytes in size. Low-power wireless protocols employ short packets, *e.g.*, the maximum packet length of Zigbee is 127 bytes and 47 bytes for Bluetooth low energy. The next Controller Area Network (CAN) standard, CAN-FD, has a maximum packet size of 64

[1]https://en.wikipedia.org/wiki/Internet_Mix

Received: 2022-10-15 Accepted: 2022-12-15 Published: 2023-03-06

bytes. In the use of narrow-band IoT [NBI19], even the communication of 1-bit messages (*e.g.*, for device monitoring) is considered one of the target applications. For end-to-end encryption schemes in real-time video conference systems such as Zoom [Jos21] and Webex [Sys20, OUGM21], which rapidly became popular due to the COVID-19 pandemic, the frame size is about 1K bytes. In these applications, an efficient hash function for short messages is essential. Specifically, to maintain the authenticity of the message, particularly against potentially malicious servers, the hash value of each packet/frame should be signed by digital signatures [IIM21]. As such systems require real-time processing, the hash function should be as fast as possible for short messages.

Short inputs are also crucial for the future of mobile communications. So-called "beyond 5G" or 6G mobile communication technology will require the use of short packets to achieve ultra-low latency communication. In comparison, some applications of 6G are expected to require a peak speed of over 100 GBps [LaL19].

The importance of the short-input hash function has been widely recognized in the cryptographic research community. The NIST report on Lightweight Cryptography (LwC) [MBTM17, Sect 2.3.2] explicitly mentioned that lightweight applications typically need a hash function optimized for short messages, such as 256 bits. Some NIST LwC proposals advertise their performances for short inputs, such as the finalists Ascon [DEMS19] and Romulus [IKMP20], and a second-round candidates ForkAE [ALP+19] and Saturnin [CDL+20].

The ongoing NIST LwC project targets lightweight hash functions, but only a few proposals use AES because the project mainly focuses on devices with low computational resources. On the other hand, the percentage of CPUs that have (the components of) AES as a dedicated instruction is rapidly increasing in the mobile and desktop PC world, represented by Intel AES-NI and ARMv8 AES instructions. Steam Hardware Survey shows that the number of CPUs with AES instructions is as high as 96.05% of the clients as of June 2022[2]. Standardization of AES instructions is also being considered for the RISC-V architecture [MNP+21], which is expected to become popular in the future. This trend will also spread to low-end platforms such as IoT edge devices.

1.2 Related Work

Short-(Fixed)-Input-Length (SFIL) Hash Functions. Haraka v2 is a SFIL hashing for post-quantum applications such as hash-based signature schemes [KLMR16]. However, a recent study by Bao *et al.* [BDG+21] reveals that preimage attacks on Haraka-256 and Haraka-512 up to 9 out of 10 rounds and 11 out of 10 rounds are feasible, respectively. That is, Haraka-512 is completely broken by their cryptanalyses, and the security margin of Haraka-256 is only one round. Simpira v2 is a family of permutations [GM16], and a short-input hashing is one of its applications. Although the security flaw of this application has yet to be found, it needs to be better optimized for recent CPU architectures, especially for a single permutation call, which is required for this application. For example, one round of the 256-bit variant requires two times AES round function calls, and each AES call should be sequentially executed because the second execution requires the output of the first execution. Because Intel Ice Lake or later processors can pipeline up to 6 AES instructions, it does not take full advantage of the pipeline.

Variable-Input-Length (VIL) Hash Functions. As efficient VIL hash functions, there are KangarooTwelve [BDP+18], ParallelHash256 [KjCP16], and BLAKE3 [OANWO20], in which parallel/tree hash structures allow to leverage pipeline and parallel executions to enhance the performance in software. However, these are effective only for long input, *i.e.*, processing short messages (less than 2K bytes) is much less efficient. In addition,

[2]https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam

KangarooTwelve and BLAKE3 guarantee the 128-bit preimage security. Other standard hash functions SHA2-256 [oST15a] and SHA3-256 [oST15b] are also not efficient for short messages.

1.3 Motivation

Looking into real-world applications, an efficient hash function for short inputs up to 2K bytes is essential and will become increasingly important in the future. However, existing schemes need to be better optimized for short input. Especially, there are still no satisfactory VIL hash functions for short messages in terms of speed and security. To bridge the gap between real-world applications' needs and state-of-the-art hash functions' performances, we aim to design efficient and secure hash functions for short messages.

Specifically, our design goal for hash functions is to be highly efficient for short messages up to 2K bytes, and competitive even for long messages to software-efficient hash functions KangarooTwelve, ParallelHash256, and BLAKE3 on modern desktop and mobile platforms.

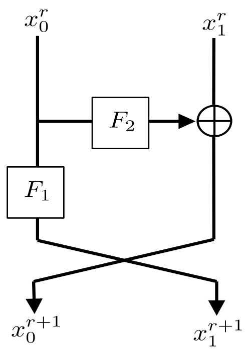
1.4 Our Contribution

To achieve our design goals, we first specify a family of efficient permutations Areion that is optimized for the latest CPU architectures, including Intel and ARM, by fully leveraging the power of AES instructions. As for its applications, we propose SFIL and VIL hashing. We then evaluate the security of underlying permutations and their applications and measure software performances in several architectures. Our contributions in this paper are summarized as follows.

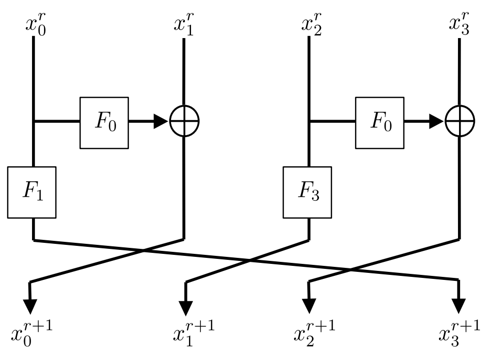
Software-Efficient AES-Based Permutations. For environments where AES instructions are available, we design a family of permutations, dubbed Areion, that can be implemented by only AES instructions such as aesenc and aesenclast in AES-NI or vaeseq and vaesmcq in ARMv8 NEON as AES instructions are most efficient cryptographic operation among SIMD operations. As an underlying structure, we propose pipeline-friendly Feistel-type schemes in which additional F functions are appended to Feistel-type schemes to take full advantage of the pipeline executions. We find optimal instantiations of F functions by thoroughly analyzing the security and performance of all possible candidates. As a result, the performance of Areion is significantly faster than existing permutations in the latest CPU architectures. Especially, Areion outperforms other permutations in the encrypt direction. It is an important characteristic of our target applications.

SFIL Hash Function. For an SFIL hashing, we apply Areion to the Davies-Meyer (DM) construction, which consists of a permutation with a feed-forward (applying the XOR operation) of the input as with Simpira v2 [GM16] and Haraka v2 [KLMR16]. Our schemes provide a 256-bit security level against preimage attacks. In addition, these are about 1.4 times faster than the schemes based on Simpira v2.

VIL Hash Function. For a VIL hashing, we design a compression function based on Areionand implant it to the general Merkle-Damgård (MD) construction [Mer89, Dam89]. Our scheme performs much faster than any other hash functions for the input size up to 1024 bytes and even competitive to other software-efficient hash functions for longer inputs in laptop and mobile environments while ensuring the 256-bit security level of preimage attacks. Its performance is less than three cycles/byte for any message size. It is about ten times faster than existing state-of-the-art schemes for short messages up to around 100 bytes. We remark that such message lengths are typical in real-world applications on the latest CPU architectures (IceLake, Tiger Lake, and Alder Lake) and mobile platforms (Pixel 6 and iPhone 13).



(a) Areion-256



(b) Areion-512

Figure 1: The round functions of Areion.

1.5 Paper Organization

In Sect. 2, we describe the specification of Areion. Sect. 3 explains details of our design rational of Areion and discusses the optimally of our design choices. In Sect. 4, we show several applications of Areion. In Sects. 5 and 6, we give the security and performance evaluations of Areion and its applications, respectively. Sect. 7 concludes the paper.

2 Specification of Permutations

We show the specification of Areion. Areion is based on Simpira v2 but has the structure that allows more AES instructions to be executed in parallel. We provide the following two variants of our permutation: Areion-256 and Areion-512. The former accepts a 256-bit block, and the latter accepts a 512-bit block as input.

To illustrate the specification of each permutation, we denote by $F_i \, (i \in \{0,1,2,3\})$ the function based on the operations in AES round function. Let SubBytes, ShiftRows, MixColumns, and AddRoundConstant in the AES round function be SB, SR, MC, and AC, respectively. AC is equivalent to AddRoundKey in ordinal AES, but the constant is added instead of the round key. F_i consists of a combination of SB, SR, MC and AC. For each value of i, F_i is defined as follows:

$$\begin{aligned}
F_0 &= MC \circ SR \circ SB \\
F_1 &= SR \circ SB \\
F_2 &= MC \circ SR \circ SB \circ AC \circ MC \circ SR \circ SB \\
F_3 &= MC \circ SR \circ SB \circ AC \circ SR \circ SB
\end{aligned}$$

A combination of AES instructions in AES-NI or NEON can implement these functions. Areion-256 consists of F_1 and F_2, and Areion-512 consists of F_0, F_1, and F_3. The round function of each variant is shown in Fig. 1.

We set the number of rounds of Areion-256 and Areion-512 are 10 and 15, respectively. These are derived from our security evaluation. Sect. 5 describes the details. The round constants are derived from the binary digits of a fraction part of $\pi = 3.1415926 \cdots$. Table 1 shows round constants in hexadecimal notation. In the r-th round of Areion, RC_r is added to the state.

Table 1: Round constants.

RC	Round constant
RC_0	`0x243f6a8885a308d313198a2e03707344`
RC_1	`0xa4093822299f31d0082efa98ec4e6c89`
RC_2	`0x452821e638d01377be5466cf34e90c6c`
RC_3	`0xc0ac29b7c97c50dd3f84d5b5b5470917`
RC_4	`0x9216d5d98979fb1bd1310ba698dfb5ac`
RC_5	`0x2ffd72dbd01adfb7b8e1afed6a267e96`
RC_6	`0xba7c9045f12c7f9924a19947b3916cf7`
RC_7	`0x801f2e2858efc16636920d871574e690`
RC_8	`0xa458fea3f4933d7e0d95748f728eb658`
RC_9	`0x718bcd5882154aee7b54a41dc25a59b5`
RC_{10}	`0x9c30d5392af26013c5d1b023286085f0`
RC_{11}	`0xca417918b8db38ef8e79dcb0603a180e`
RC_{12}	`0x6c9e0e8bb01e8a3ed71577c1bd314b27`
RC_{13}	`0x78af2fda55605c60e65525f3aa55ab94`
RC_{14}	`0x5748986263e8144055ca396a2aab10b6`

3 The Design

3.1 AES Instructions and SIMD

SIMD is an abbreviation for single instruction multiple data, and a type of parallel processing. Most modern processors support instructions set for SIMD. SIMD instructions perform operations vector-wise using data stored in dedicated registers, which allows arithmetic/bitwise operations in parallel and advanced operations like data shuffling to be performed with a single instruction.

An example of SIMD instructions that can perform complex operations is an instruction for executing AES, which is the dominant block cipher. This instruction belongs to AES-NI (AES New Instructions set) in the Intel/AMD processors. AES-NI includes aesenc to perform the round function of the encryption, aesenclast for the final round, instructions for decryption, and instructions to support the round key generation. On the other hand, in the ARMv8 processors, AES instructions are included in the NEON instructions set. AES instructions in NEON include vaeseq for AddRoundKey, SubBytes, and ShiftRows, and vaesmcq for MixColumns. NEON also supports the decryption instructions, while instructions to assist in the round key generation are not supported.

The performance of SIMD instructions can be measured by their latency, throughput, and port usage. Latency means the number of clock cycles that are required for the execution of an instruction. Throughput means the number of clock cycles required to wait before the responsible ports can accept the same instruction again. Dispatched instructions are decomposed into micro-operations and then processed by each execution port.

According to the website by Abel and Reineke *et al.* [RTL21], the latency and throughput of aesenc/aesenclast in Ice Lake are 3 and 0.5, respectively. A throughput of 0.5 means that two execution ports can accept the micro-operation from aesenc/aesenclast and each operation's throughput is 1 [RTL21]. Fig. 2 illustrates the pipelined execution of multiple aesenc on Ice Lake. We can see that up to 6 aesenc can be executed in 5 cycles on Ice Lake using two execution ports, port 0 and port 1.

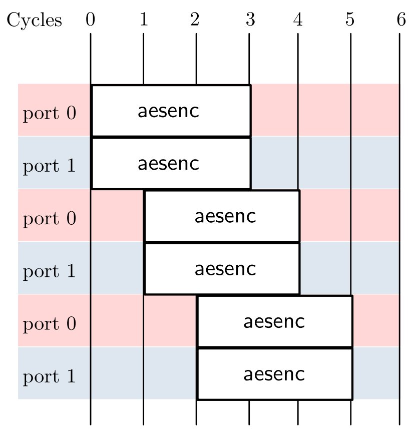


Figure 2: Execution of aesenc on Ice Lake processors.

Table 2: The latency and throughput of aesenc, referred by [RTL21].

Processor	aesenc	
	Latency	Throughput
Skylake	4	1
Kaby Lake		
Coffee Lake		
Cannon Lake		0.5
Ice Lake	3	
Tiger Lake		
Alder Lake		
Zen +	4	0.5
Zen2		

3.2 General Construction

3.2.1 Permutations Realized by only AES Instructions

To construct optimal permutations in environments where hardware instructions of AES are available, we focus on a class of *permutations that can be implemented solely by AES instructions* such as aesenc and aesenclast in AES-NI or vaeseq and vaesmcq in ARMv8 NEON for the following reasons.

- The latency of AES instructions in AES-NI becomes smaller as the processor's architecture is upgraded. Moreover, Intel 9th generation and later processors have an additional execution port that accepts micro-operations generated from AES instructions, which improves the throughput from 1 to 0.5. The latency and throughput of aesenc in Intel processors from 6 to 11 generation are shown in Table 2.

- Schemes based solely on AES instructions are beneficial in terms of performance and security. Since NIST selected AES as a standard block cipher in 2001, no attack has been published in spite of considerable cryptanalytic efforts over the past 20 years, and its security is deeply understood in the community of symmetric cryptography. Thus, it is easy to evaluate its security by existing tools convincingly and accumulated cryptanalysis knowledge.

- Haraka v2 [KLMR16] is a family of permutations. It is an SPN-type scheme based on AES instructions and word shuffle operations, such as unpack instructions. Shiba *et* *al.* show that the structure of Haraka v2 is optimal among SPN-type schemes based solely on AES instructions and shuffle operations [SSI21]. Thus, presenting a new SPN-type scheme with better performance than Haraka v2should be challenging.

- Word shuffle operations provide only simple linear transformations. In contrast, AES instructions include not only more complex linear operations (*i.e.*, MixColumns and ShiftRows) but also nonlinear operations (*i.e.*, 16 parallel executions of 8-bit S-box) by only a single instruction call. In addition, the latency of word shuffle operations requires one, even on the latest CPU architectures. Thus, arguably AES instructions are the most efficient and cryptographically-strong operations in all SIMD instructions.
- Haraka v2 does not provide a sufficient level of security as a hash function according to the recent study by Bao *et al.* [BDG$^+$21]. They present preimage attacks on Haraka-256 and Haraka-512 up to 9 out of 10 rounds and 11 out of 10 rounds, respectively. In addition, designers of Haraka v2 did not claim any security as a public permutation. According to these facts, Haraka v2 should require roughly 1.2 to 1.5 times of recommended rounds by the designers, *i.e.*, about 12 to 15 rounds to ensure the security as public permutations of Haraka v2 and hash functions. These additional rounds degrade the performance of Haraka v2 significantly. We remark that, due to the structure of Haraka v2, increasing the number of rounds requires not only more AES instructions but also more word shuffle operations. Thus, it significantly impacts the overall performance of the tweaked versions of Haraka v2 compared to the Feistel-type scheme such as Simpira v2, which is a class of *permutations that can be implemented solely by AES instructions.*

For the above reasons, we choose the Feistel-type scheme to design new 256- and 512-bit permutations from 128-bit AES instructions.

3.2.2 Feistel-type Scheme for Leveraging the Pipeline

Limitations of Simpira v2. For the 256- and 512-bit variants of Simpira v2 (hereafrer, we will refer to each variant as Simpira-256 and Simpira-512, respectively), there is still room for improvement in their design, considering the characteristic of AES instructions in modern processors, especially for applications that require sequential executions of underlying permutations, *e.g.*, SFIL and VIL hash functions.

Specifically, the one-block encryption of Simpira-256 requires two times of executions, and each AES call should be sequential because the second execution requires the output of the first execution. On the other hand, one-block encryption of Simpira-512 is capable of pipelining up to two 2-round AES executions. However, since Intel Ice Lake or later processors can pipeline up to 6 AES instructions, the structure of Simpira-256 and Simpira-512 does not take full advantage of the pipeline.

Pipeline-Friendly Feistel-type Schemes. To take advantage of the pipeline as possible, we design pipeline-friendly Feistel-type schemes in which F functions are added in left branch for 256-bit version and first and third branches for 512-bit version to Feistel-type scheme, respectively, as shown in Fig. 3. These allow for pipelined execution of two and four AES instructions, respectively.

As another possible scheme, we can add F functions in the right branch for the 256-bit version and the second and fourth branches for the 512-bit version to the above schemes before XOR operations, respectively. However, our initial evaluation confirmed that these additional instructions do not improve the performance because they cannot significantly reduce the required number of rounds to ensure the security of structural attacks on Feistel, such as impossible differential and integral attacks. Besides, the critical path in

Table 3: Instructions per cycle (IPC) of each permutation.

Algorithm	#Round	IPC
Areion-256	10	**0.66**
Simpira-256	15	0.46
Areion-512	15	**0.92**
Simpira-512	15	0.52

the decryption of this scheme becomes three times longer than that of the encryption. From these facts, we conclude that *the schemes in Fig. 1 are optimal for 2- and 4-line Feistel-type schemes for high performance.*

Comparison. In order to compare the degree of utilization of the pipeline, we checked instructions per cycle (IPC) of each variant of Areion and Simpira v2 by static code analysis using LLVM machine code analyzer (llvm-mca). Table 3 shows the results. For both variants, the results show the IPC of Areion is larger than that of Simpira v2. Based on this fact, the construction of Areion can utilizes the pipeline more effectively.

3.3 Finding Optimal Constructions

Possible Candidates of F Functions. Recall that our permutations are realized solely by AES instructions. As already discussed in [KLMR16, GM16, SSI21], F functions consisting of one or two AES round functions are optimal in Feistel- and SPN-type schemes. In this work, to find further efficient constructions, we also consider last-round instructions such as aesenclast in AES-NI or vaesmcq in ARMv8 NEON, respectively, as underlying instructions. Thus, F functions should be realized by one or two combinations of aesenc and aesenclast in AES-NI or vaeseq and vaesmcq in ARMv8 NEON, respectively.

There are six possible candidates of F_i ($i \in \{0, 1, 2, 3, 4, 5\}$), where F_0, F_1, F_2, F_3 are defined in Sect. 2 and F_4 and F_5 are as follows.

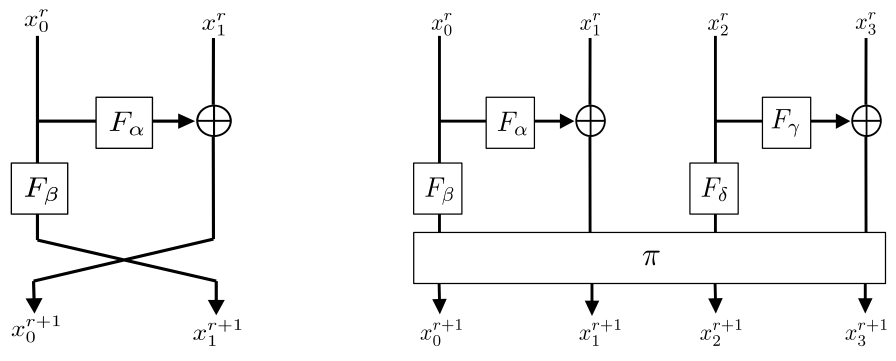
$$F_4 = SR \circ SB \circ AC \circ MC \circ SR \circ SB$$
$$F_5 = SR \circ SB \circ AC \circ SR \circ SB$$

For AES-NI, F_0, F_1, F_2, F_3, F_4 and F_5 are implemented by aesenc, aesenclast, aesenc $\rightarrow$ aesenc, aesenclast $\rightarrow$ aesenc, aesenc $\rightarrow$ aesenclast, and aesenclast $\rightarrow$ aesenclast, respectively. Note that XOR operations in the Feistel-type scheme are executed by the operation of AddRoundKey, which is the last operation of aesenc and aesenclast, respectively. This feature of AddRoundKey is the reason why AC is absent in the last of these equations.

For ARMv8 NEON, F_0, F_1, F_2, F_3, F_4 and F_5 are implemented by vaeseq $\rightarrow$ vaesmcq, vaeseq, vaeseq $\rightarrow$ vaesmcq $\rightarrow$ vaeseq $\rightarrow$ vaesmcq, vaeseq $\rightarrow$ vaeseq $\rightarrow$ vaesmcq, vaeseq $\rightarrow$ vaesmcq $\rightarrow$ vaeseq, vaeseq $\rightarrow$ vaeseq, respectively. As vaeseq performs AddRoundKey before SubBytes, the AddRoundKey operation of the first vaeseq in each function is used to realize the XOR operation of the previous round for Feistel-type schemes. This observation implies that our schemes can be implemented solely by vaeseq and vaesmcqin NEON, except for the XOR operation in the last round.

How to Find F functions. To find optimal combinations of functions F_i ($i \in \{0, 1, 2, 3, 4, 5\}$) in Fig. 3, we first evaluate the security against differential/linear, impossible differential, and integral attacks using Mixed-Integer Linear Programming (MILP) for all combinations. Let R_1, R_2 and R_3 be the number of rounds where the following three conditions are satisfied, respectively.

R_1: The number of rounds where the minimum number of active S-boxes is enough to ensure the security against differential/linear attacks.



(a) 256-bit Permutation **(b)** 512-bit Permutation

Figure 3: Target constructions of our permutations.

R_2: The number of rounds where there is no any byte-truncated impossible differential characteristic.

R_3: The number of rounds where there is no any byte-wise integral distinguisher.

Besides, we define $\max\{R_1, R_2, R_3\}$ as $R_{\max}$. After obtaining $R_{\max}$, we will look into characterises for the performance in $R_{\max}$ to find most efficient ones. The details are explained in the followings.

3.3.1 On 256-bit Permutations

Let a 256-bit permutation with F_α and F_β functions be (α, β)-perm, where $\alpha, \beta \in \{0, 1, 2, 3, 4, 5\}$, as illustrated in Fig. 3. As a 256-bit permutation has two functions in which there are six possible candidates, the total number of combinations is 36 $(= 6 \times 6)$. Among them, we look for combinations implemented by the lowest number of AES instructions in $R_{\max}$, *i.e.*, we choose the ones that can achieve the required security level with the lowest number of AES instructions.

Table 4 shows R_1, R_2, R_3, $R_{\max}$ and the number of AES instructions in $R_{\max}$ of all 36 candidates. According to this table, the lowest one is $(2, 1)$-perm for which, R_1, R_2 and R_3 are estimated as 5, 5, and 4, respectively, namely, $R_{\max} = 5$, and #AES instructions in 5 rounds is only 15. From this result, we select $(2, 1)$-perm as underlying one for Areion-256.

3.3.2 On 512-bit Permutations

Let a 512-bit permutation with F_α, F_β, F_γ and F_δ functions using π block shuffle layer be $(\alpha, \beta, \gamma, \delta, \pi)$-perm, where $\alpha, \beta, \gamma, \delta \in \{0, 1, 2, 3, 4, 5\}$, as illustrated in Fig. 3. As a 512-bit permutation has four F functions in which there are six possible candidates, and π block shuffle has 24 $(= 4!)$ patterns, the total number of combinations is estimated as 31104 $(= 6 \times 6 \times 6 \times 6 \times 4!)$.

To find the most efficient combination among them, we thoroughly analyze security and performance using the following procedures.

Step 1: Limiting the Number of AES Instructions in $R_{\max}$. As with the 256-bit case, we focus on combinations implemented by the lowest number of AES instructions in $R_{\max}$. As a result of our search, we find 30 candidates in which the lowest number of AES instructions in $R_{\max}(= 9)$ is 45, as shown in Table 5.

Step 2: Eliminating the Equivalent Candidates. 28 candidates out of the remaining 30 ones can be classified into 14 equivalent classes, *i.e.*, each two candidates of them is

Table 4: Search results on 256-bit permutations.

(α, β)	R_1	R_2	R_3	R_{max}	#AES instructions
$(0, 0)$	23	7	5	23	46
$(0, 1)$	8	9	5	9	18
$(0, 2)$	6	5	4	6	18
$(0, 3)$	16	7	4	16	48
$(0, 4)$	6	6	4	6	18
$(0, 5)$	6	8	5	8	24
$(1, 0)$	8	9	6	9	18
$(1, 1)$	33	-	-	-	-
$(1, 2)$	6	5	4	6	18
$(1, 3)$	6	9	6	9	27
$(1, 4)$	6	9	5	9	27
$(1, 5)$	23	-	-	-	-
$(2, 0)$	6	5	4	6	18
$(2, 1)$	**5**	**4**	**5**	**5**	**15**
$(2, 2)$	6	4	3	6	24
$(2, 3)$	6	5	4	6	24
$(2, 4)$	4	5	3	5	20
$(2, 5)$	5	5	4	5	20
$(3, 0)$	16	7	4	16	48
$(3, 1)$	7	9	5	9	27
$(3, 2)$	6	5	4	6	24
$(3, 3)$	12	-	4	-	-
$(3, 4)$	4	6	4	6	24
$(3, 5)$	7	-	5	-	-
$(4, 0)$	6	7	4	7	21
$(4, 1)$	6	9	5	9	27
$(4, 2)$	4	5	3	5	20
$(4, 3)$	4	6	4	6	24
$(4, 4)$	7	-	3	-	-
$(4, 5)$	6	-	5	-	-
$(5, 0)$	7	8	6	8	24
$(5, 1)$	23	-	-	-	-
$(5, 2)$	4	5	4	5	20
$(5, 3)$	7	-	6	-	-
$(5, 4)$	6	-	5	-	-
$(5, 5)$	17	-	-	-	-

mapped to one equivalent class. Based on this fact, we can eliminate 14 equivalent classes, and then we can reduce to 16 $(= 30 - 14)$ candidates.

Step 3: Considering Efficiency in NEON Instructions. The remaining 16 combinations can be classified into three different classes. Specifically, each different class has the following different π:

$$\pi_1 : x_0^r || x_1^r || x_2^r || x_3^r \mapsto x_1^{r+1} || x_2^{r+1} || x_3^{r+1} || x_0^{r+1}$$
$$\pi_2 : x_0^r || x_1^r || x_2^r || x_3^r \mapsto x_3^{r+1} || x_0^{r+1} || x_1^{r+1} || x_2^{r+1}$$
$$\pi_3 : x_0^r || x_1^r || x_2^r || x_3^r \mapsto x_1^{r+1} || x_3^{r+1} || x_0^{r+1} || x_2^{r+1}$$

The two constructions in π_3 are unsuitable for implementations using NEON instructions in ARMv8. This is due to the fact that the implementation of these

Table 5: Search results of 512-bit permutations.

$(\alpha, \beta, \gamma, \delta, \pi)$	R_1	R_2	R_3	R_{max}	#AES instructions
$(0, 0, 0, 4, \pi_1)$	8	9	5	9	45
$(0, 0, 1, 2, \pi_1)$	9	9	6	9	45
$(0, 0, 2, 1, \pi_1)$	9	9	6	9	45
$(0, 0, 4, 0, \pi_1)$	8	9	6	9	45
$(0, 0, 5, 0, \pi_1)$	9	9	7	9	45
$(0, 1, 0, 2, \pi_1)$	9	9	6	9	45
$(0, 1, 0, 3, \pi_1)$	9	9	6	9	45
$(0, 1, 0, 4, \pi_1)$	8	9	6	9	45
$(0, 1, 2, 0, \pi_1)$	9	9	6	9	45
$(0, 1, 2, 1, \pi_1)$	8	9	6	9	45
$(0, 2, 0, 1, \pi_1)$	9	9	6	9	45
$(0, 2, 1, 0, \pi_1)$	9	9	6	9	45
$(0, 3, 0, 1, \pi_1)$	9	9	6	9	45
$(0, 4, 0, 0, \pi_1)$	8	9	5	9	45
$(0, 4, 0, 1, \pi_1)$	8	9	6	9	45
$(1, 0, 0, 2, \pi_1)$	9	9	6	9	45
$(1, 0, 2, 0, \pi_1)$	9	9	7	9	45
$(1, 2, 0, 0, \pi_1)$	9	9	6	9	45
$(2, 0, 0, 1, \pi_1)$	9	9	6	9	45
$(2, 0, 1, 0, \pi_1)$	9	9	7	9	45
$(2, 1, 0, 0, \pi_1)$	9	9	6	9	45
$(2, 1, 0, 1, \pi_1)$	8	9	6	9	45
$(4, 0, 0, 0, \pi_1)$	8	9	6	9	45
$(5, 0, 0, 0, \pi_1)$	9	9	7	9	45
$(0, 1, 0, 2, \pi_2)$	8	9	5	9	45
$(0, 2, 0, 1, \pi_2)$	8	9	5	9	45
$(1, 0, 2, 0, \pi_2)$	9	9	5	9	45
$(2, 0, 1, 0, \pi_2)$	9	9	5	9	45
$(0, 0, 1, 2, \pi_3)$	9	9	6	9	45
$(0, 0, 1, 4, \pi_3)$	9	9	6	9	45

two constructions by NEON requires successive XORs, which hampers the implementation with only vaeseq and vaesmcq while maintaining the compatibility of the implementation on ARM and Intel. Based on this fact, we eliminate these two constructions using π_3 from the candidates. As a result, we obtain 14 constructions.

Step 4: Estimating Theoretical Number of Cycles. For the remaining 14 candidates, we use a performance analysis tool llvm-mca to estimate the theoretical number of cycles in Ice Lake or later architecture. Table 6 shows theoretical values of total cycles in 15-round encryption, calculated by llvm-mca. According to this result, we reduce to 6 candidates with the lowest number of cycles to perform the encryption.

Step 5: Performing Experimental Evaluations. We measure the performance of the remaining six candidates on several platforms. Table 6 shows the results on Ice Lake architecture (Intel(R) Core(TM) i7-1068NG7 CPU @ 2.30GHz). From these results, we selected $(0, 1, 0, 3, \pi_1)$-perm as the optimal combination for Areion-512.

Table 6: Theoretical and experimental value of total cycles at 15-round encryption.

$(\alpha, \beta, \gamma, \delta, \pi)$	total cycle (by llvm-mca)	cpb (by experiments)
$(0,0,0,4,\pi_1)$	**8899**	1.09016
$(0,0,1,2,\pi_1)$	**8899**	1.08927
$(0,0,2,1,\pi_1)$	11528	-
$(0,0,4,0,\pi_1)$	11528	-
$(0,0,5,0,\pi_1)$	11528	-
$(0,1,0,2,\pi_1)$	**8899**	1.08918
$(0,1,0,3,\pi_1)$	**8899**	**1.08882**
$(0,1,0,4,\pi_1)$	**8899**	1.08989
$(0,1,2,0,\pi_1)$	11528	-
$(0,1,2,1,\pi_1)$	11528	-
$(1,0,0,2,\pi_1)$	**8899**	1.09017
$(1,0,2,0,\pi_1)$	11528	-
$(0,1,0,2,\pi_2)$	9109	-
$(1,0,2,0,\pi_2)$	9919	-

4 Applications

4.1 SFIL Hash Function

For an SFIL hashing, we apply Areion to the Davies-Meyer (DM) construction, which consists of a permutation with a feed-forward (applying the XOR operation) of the input. The use of DM for SFIL hashing has already been discussed in [GM16, KLMR16]. In particular, Haraka v2 implemented two SFIL hash functions, Haraka256-DM : $\mathbb{F}_2^{256} \to \mathbb{F}_2^{256}$ and Haraka512-DM : $\mathbb{F}_2^{512} \to \mathbb{F}_2^{256}$, defined as follows:

$$\mathsf{Haraka256\text{-}DM}(x) = \pi_{256}(x) \oplus x, \tag{1}$$

$$\mathsf{Haraka512\text{-}DM}(x) = \mathsf{trunc}(\pi_{512}(x) \oplus x), \tag{2}$$

where π_{256} and π_{512} are the 256- and 512-bit permutations of Haraka v2, respectively; and trunc : $\mathbb{F}_2^{512} \to \mathbb{F}_2^{256}$ is a truncation function defined as follows:

$$\mathsf{trunc}(x_0 || \cdots || x_{15}) = x_2 || x_3 || x_6 || x_7 || x_8 || x_9 || x_{12} || x_{13}, \tag{3}$$

where $x = x_0 || \cdots || x_{15} \in \mathbb{F}_2^{512}$. Our SFIL hash functions, Areion256-DM and Areion512-DM, use Areion-256 and Areion-512 instead of Haraka v2's ones. The DM construction uses only the forward direction of the permutation, and the overhead beyond the permutation is negligible. Thus, the performances of Areion256-DM and Areion512-DM are effectively the same as those of the forward direction of underlying permutations.

The designers of Simpira v2 suggested its application to SFIL hash functions [GM16]. Then, for performance comparison, we define DM construction instantiations of Simpira v2 in the same way as above and refer to them as Simpira256-DM and Simpira512-DM.

4.2 VIL Hash Function

For a VIL hashing, we apply Areion-512 to the Merkle-Damgård (MD) construction, a classical method of building a cryptographic hash function from a compression function [Mer89, Dam89].

Our VIL hash function, Areion512-MD, is an MD construction instantiated with Areion512-DM. Other design details of Areion512-MD follow SHA2-256 [oST15a]. SHA2-256 has two phases, preprocessing and hash computation phases. The former is further divided into three steps: padding the message, parsing the message into message blocks and setting the initial hash value. For Areion512-MD, padding and message parsing are executed in the same procedure as SHA2-256. However, the length of the padded message should be adjusted to be a multiple of 256 bits instead of a multiple of 512 bits; and the size of the parsed message block is 256 bits (see [oST15a, Section 5] for more details). Areion512-MD uses the same initial hash value H of SHA2-256, and it consists of the following two 128-bit words:

$$H_0 = \texttt{0x6a09e667bb67ae853c6ef372a54ff53a}, \tag{4}$$

$$H_1 = \texttt{0x510e527f9b05688c1f83d9ab5be0cd19}. \tag{5}$$

Then, Areion512-DM is used for the hash computation phase. The parsed message block is inserted into x_0 and x_1 of the input word positions in Areion-512, and the initial hash value and chaining values (that is, the output value of each compression function) are set into x_2 and x_3 of the input word positions in Areion-512 (see Fig. 1b). Finally, the output value of the last DM compression function becomes a 256-bit message digest.

The designers of Simpira v2 and Haraka v2 did not mention its application to VIL hash functions [GM16, KLMR16]. However, for performance comparison, we define an MD construction instantiation of Simpira512-DM and Haraka512-DM in the same way as above and refer to them as Simpira512-MD and Haraka512-MD, respectively.

5 Security Evaluation

5.1 Security for Underlying Permutations

We evaluate the security of Areion-256 and Areion-512 as public permutations against differential, linear, impossible differential, and integral attacks.

Claimed Security for Underlying Permutations. We claim 128-bit security for both Areion-256 and Areion-512 as with Simpira v2, *i.e.*, we consider the attacks up to 2^{128} complexity. There is no rigorous definition of a distinguisher for a public permutation. In the literature, there is, however, a very related concept called the known-key distinguisher [KR07] or the correlation intractability [CGH04]. Note that once the key is known for a block cipher, the block cipher becomes a public permutation. Roughly speaking, a known-key distinguisher is that for a block cipher, there exists a relation such that given the key, it is easy to find plaintext-ciphertext pairs satisfying this relation. However, it is difficult to find them for a random permutation [KR07]. Moreover, if the relationship is simply the description of the block cipher itself, this should be meaningless for the following reasons. First, every block cipher will be vulnerable to this attack with only 1 query. Second, the relationship is not interesting at all from the designers' perspective [KR07]. In [Gil14], a more formal definition of the known-key distinguisher for a block cipher was given, which is a rigorous description of the above statement. In both the known-key distinguishers on AES [KR07, Gil14], they indeed are the extensions of the well-known integral attack on round-reduced AES, where the attackers start from a middle round and aim to find an input-output set such that the sum of some bytes in the input and output are all zero, respectively. We will rely on similar start-from-the-middle techniques to construct zero-sum distinguishers for our proposed public permutations. Moreover, our zero-sum distinguishers also resemble the known-key distinguishers on AES [KR07, Gil14] because we similarly find distinguishers based on the well-known integral attack on AES.

Differential/Linear Attacks. We estimate the security against differential/linear attacks by obtaining the lower bound for the number of differentially/linearly active S-boxes with

Table 7: The lower bound for the number of differentially/linearly active S-boxes for Areion-256 and Areion-512. Here, AS_D and AS_L denote the number of differentially and linearly active S-boxes, respectively.

Primitives \ Rounds		1	2	3	4	5	6	7	8	9	10	11	12
Areion-256	AS_D	0	6	12	38	46	53	60	86	92	99	108	135
Areion-512	AS_D	0	2	5	8	20	36	62	73	90	106	119	135
Areion-256	AS_L	0	1	9	24	42	48	65	79	91	102	114	128
Areion-512	AS_L	0	1	4	8	21	35	50	68	89	103	120	131

an MILP-based method proposed by Mouha *et al.* [MWGP11]. AS_D and AS_L denote the lower bound for the number of differentially and linearly active S-boxes, respectively.

Since the maximal differential and linear probability of the S-box of AES are both 2^{-6}, $AS_{D/L}$ of ≥ 22 ($2^{-6\times 22} < 2^{-128}$) is sufficient to ensure 128-bit security against differential/-linear attacks. Table 7 shows the lower bound for the number of differentially/linearly active S-boxes for Areion-256 and Areion-512. In our evaluation, Areion-256/Areion-512 achieves both AS_D and AS_L of ≥ 22 at 4/6 rounds, and both AS_D and AS_L at 12 rounds for both permutations outnumber well over 22. Therefore, we expect full rounds of Areion-256 and Areion-512 can resist differential and linear attacks.

Impossible Differential Attacks. The miss-in-the-middle approach is known as an efficient way to find the longest impossible differences, which can be implemented by an MILP with a small change from an MILP model for counting the number of differentially active S-boxes [ST17, CJF$^+$16]. In our evaluation, we search a class of impossible differential characteristics where input and output differences activate only one byte to find the longest impossible differences efficiently.

By this approach, we find the impossible differences at 4/8 rounds of Areion-256/Areion-512, both of which are the longest ones we can find. Since there is still enough margin to full rounds for both permutations, we expect that full rounds of Areion-256 and Areion-512 can resist impossible differential attacks.

Integral Attacks. To find the integral distinguisher, we evaluate the byte-wise division property with a MILP-based method proposed by Xiang *et al.* [XZBL16]. We search the input space where only one byte is constant, and the remaining bytes are active, *i.e.*, the data/time complexity of the integral distinguishers are 2^{248} and 2^{504} for Areion-256 and Areion-512, respectively.

As a result, we find the 3- and 5-round integral distinguisher on Areion-256 and Areion-512, respectively. It should be emphasized that the required data/time complexities for these distinguishers exceed our security claim. Hence, the longest integral distinguishers with up to 2^{128} data/time complexity, which are in our security claim, are expected to exist on fewer rounds than that of these distinguishers. Thus, we expect full rounds of Areion-256 and Areion-512 can resist integral attacks.

Zero-sum Distinguishers. The zero-sum distinguisher [AM09] is a popular attack on public permutations. The overall attack procedure is straightforward. Specifically, the attackers first choose a particular set of intermediate state values and then propagate this set of values backwards and forwards, respectively. If, in the corresponding set of inputs and outputs, the sum of some input bits and output bits are zero, respectively, a zero-sum distinguisher is found. We have evaluated the resistance against this attack based on the well-known 4-round integral distinguisher for AES. It is found that there are zero-sum distinguishers for 5-round Areion-256 and 10-round Areion-512, respectively. The data and time complexity of the 2 zero-sum distinguishers are the same, which are both 2^{32}. We give the details below.

The distinguisher for 5-round Areion-256. First, we explain the zero-sum distinguisher for 5-round Areion-256, as shown in Fig. 4.

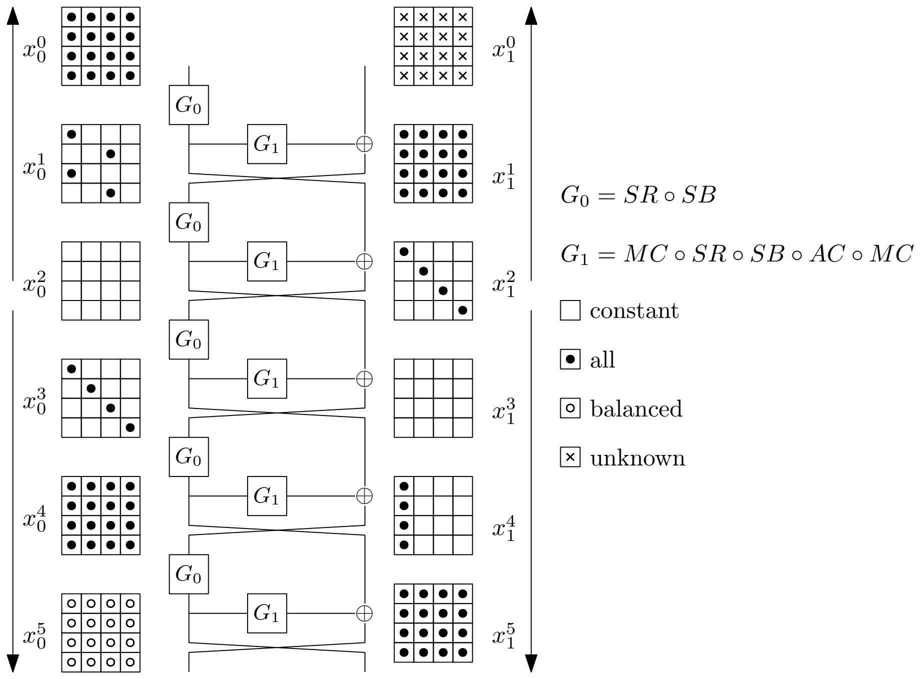


Figure 4: The zero-sum distinguisher for 5-round Areion-256.

Specifically, we choose 4 bytes of x_1^2 which traverses all the 2^{32} possible values. For x_0^2, it is assigned to a random constant value. According to the round function, we have

$$\begin{aligned}
x_0^4 &= G_1 \circ G_0(x_0^3) \oplus x_1^3 = G_1 \circ G_0(G_1 \circ G_0(x_0^2) \oplus x_1^2) \oplus G_0(x_0^2),\\
x_1^4 &= G_0(x_0^3) = G_0(G_1 \circ G_0(x_0^2) \oplus x_1^2),\\
x_0^5 &= G_1 \circ G_0(x_0^4) \oplus x_1^4,\\
x_1^5 &= G_0(x_0^4),
\end{aligned}$$

For the term $G_1 \circ G_0(x_0^4)$ in x_0^5, with our input form for (x_0^2, x_1^2), it is equivalent to that x_1^2 passes 4 AES rounds. For the term x_1^4 in x_0^5, it is equivalent to that x_1^2 passes 2 AES rounds. Hence, we need to use a data set of size 2^{32}, and all the bytes in x_0^5 will be balanced. For x_1^5, as x_0^4 can be viewed as applying 2 AES rounds to x_1^2, each byte of x_1^5 will also be balanced. The above observation also explains why the automatic method based on the division property could only detect a 3-round integral distinguisher in the forward direction, *i.e.*, we at least need to consider 4 AES rounds.

In the backward direction, we have

$$\begin{aligned}
x_0^1 &= G_0^{-1}(x_1^2),\\
x_1^1 &= G_0 \circ G_1(x_0^1) \oplus x_0^2,\\
x_0^0 &= G_0^{-1}(x_1^1).
\end{aligned}$$

Therefore, all the bytes in x_0^0 will be balanced. To better understand this, one can first consider the case when only one byte of x_1^2 traverses all the 2^8 possible values. For such a case, it can be easily checked that each byte in x_0^0 will also traverse all the 2^8 possible values. Hence, if one diagonal of x_1^2 takes all the possible 2^{32} values, all bytes in x_0^0 are also balanced.

The distinguisher for 10-round Areion-256. Next, we explain the zero-sum distinguisher for 10-round Areion-512, as shown in Fig. 5. We start from the state $(x_0^4, x_1^4, x_2^4, x_3^4)$ after 4 rounds of permutation. For the input form, we restrict that 4 bytes of x_0^4 will traverse all the 2^{32} possible values, as shown in Fig. 5. Then, we randomly choose a 128-bit constant $\mathcal{C}$ such that $F_0(x_0^4) \oplus x_1^4 = \mathcal{C}$ always holds. In other words, the value of x_1^4 is conditioned, and it is dynamically chosen according to x_0^4. For x_2^4, we assign a random constant value to it. For x_3^4, we also assign a random constant value $\mathcal{C}'$ to it but we require that the first column of $x_0^3 = F_1^{-1}(\mathcal{C}')$ is all 0. Note that (F_0, F_1, F_2, F_3) are defined in Sect. 2.

For such an input state, in the forward direction, we can trivially deduce that (x_0^6, x_1^6, x_3^6) are constants and one diagonal of x_2^6 will take all the 2^{32} possible values. Therefore, we can also deduce that (x_0^7, x_3^7, x_3^8) are all constants.

Since

$$\begin{aligned}
x_2^{10} &= F_0(x_2^9) \oplus x_3^9, \\
x_2^9 &= F_0(x_2^8) \oplus x_3^8, \\
x_2^8 &= F_0(x_2^7) \oplus x_3^7, \\
x_3^9 &= F_1(x_0^8) = F_1(F_0(x_0^7) \oplus x_1^7),
\end{aligned}$$

we can rewrite x_2^{10} as follows where $\mathcal{C}_i$ are 128-bit constants:

$$\begin{aligned}
x_2^{10} &= F_0(F_0(F_0(x_2^7) \oplus \mathcal{C}_0) \oplus \mathcal{C}_1) \oplus F_1(x_1^7 \oplus \mathcal{C}_2) \\
&= F_0(F_0(F_0(F_0(x_2^6) \oplus \mathcal{C}_3) \oplus \mathcal{C}_0) \oplus \mathcal{C}_1) \oplus F_1(F_3(x_2^6) \oplus \mathcal{C}_2).
\end{aligned}$$

Since $F_0 = MC \circ SR \circ SB$, $F_1 = SR \circ SB$ and $F_3 = MC \circ SR \circ SB \circ AC \circ SR \circ SB$, the term $F_0(F_0(F_0(F_0(x_2^6) \oplus \mathcal{C}_3) \oplus \mathcal{C}_0) \oplus \mathcal{C}_1)$ is equivalent to applying 4 AES rounds to x_2^6. The term $F_1(F_3(x_2^6) \oplus \mathcal{C}_2)$ is equivalent to applying 2.5 AES rounds to x_2^6. This implies that we need to use a data set of size 2^{32} to detect an integral property at x_2^{10}. Since one diagonal of x_2^6 takes all the 2^{32} possible values, each byte in x_2^{10} is balanced. For $(x_0^{10}, x_1^{10}, x_3^{10})$, we will lose the zero-sum property, and this can be deduced similarly. In other words, we can obtain a 6-round integral distinguisher with data complexity 2^{32} in the forward direction, which is one more round than the result obtained with the automatic method based on the division property. The main reason is that we dynamically choose values for x_1^4 such that $F_0(x_0^4) \oplus x_1^4$ is always a constant when x_0^4 varies.

In the backward direction, we consider a subset of $(x_0^4, x_1^4, x_2^4, x_3^4)$. Specifically, we consider the case when the first byte x_0^4 takes all the 2^8 possible values. In this case, the value of the first column of x_1^4 is dynamically chosen such that $F_0(x_0^4) \oplus x_1^4$ is a constant $\mathcal{C}$, as shown in Fig. 5. Then, we have 2^{24} such subsets in total.

Since

$$\begin{aligned}
x_1^4 &= F_3(x_2^3) = F_0 \circ AC \circ SR \circ SB(x_2^3), \\
\mathcal{C} &= F_0(x_0^4) \oplus x_1^4,
\end{aligned}$$

we have

$$AC \circ SR \circ SB(x_2^3) = F_0^{-1}(F_0(x_0^4) \oplus \mathcal{C}).$$

Since $F_0 = MC \circ SR \circ SB$, the above formula implies that the first byte of $AC \circ SR \circ SB(x_2^3)$ will traverse all the 2^8 possible values. Hence, only the first byte of x_2^3 will traverse all the 2^8 possible values. Therefore, we obtain the form of $(x_0^3, x_1^3, x_2^3, x_3^3)$ shown in Fig. 5.

Deducing $(x_0^2, x_1^2, x_2^2, x_3^2)$ from $(x_0^3, x_1^3, x_2^3, x_3^3)$ is trivial and we omit the details. Deducing (x_0^1, x_1^1) is also trivial based on $(x_0^2, x_1^2, x_2^2, x_3^2)$. Next, we mainly focus on (x_2^1, x_3^1). Similarly, we have

$$\begin{aligned} x_1^2 &= F_3(x_2^1) = F_0 \circ AC \circ SR \circ SB(x_2^1), \\ x_0^3 &= F_0(x_0^2) \oplus x_1^2, \\ AC \circ SR \circ SB(x_2^1) &= F_0^{-1}(F_0(x_0^2) \oplus x_0^3). \end{aligned}$$

As the first column of x_0^3 is zero, the first diagonal of x_2^1 will equal the first diagonal of $AC \circ SR \circ SB(x_2^1)$. Hence, we obtain the form of (x_2^1, x_3^1) as shown in Fig. 5. Based on $(x_0^1, x_1^1, x_2^1, x_3^1)$, deducing $(x_0^0, x_1^0, x_2^0, x_3^0)$ is trivial and we omit the details.

In a word, we can construct a zero-sum distinguisher for 10-round Areion-512.

5.2 Security for Hash Functions

Claimed Security for Hash Functions. We claim 256-bit security against the preimage attack for both Areion256-DM and Areion512-DM. However, as in Haraka v2 and the SFIL hash function built on Simpira v2, we do not claim their resistances against the collision attack since it is unnecessary for their applications.

For the MD-based hash function, we claim 256-bit security against the preimage attack and 128-bit security against the collision attack, the same as SHA2-256. Due to the generic second-preimage attack on the MD construction [KS05], our MD-based hash scheme could only provide about 193-bit security for second-preimage attacks. This limitation is because the maximal number of allowed message blocks is 2^{64} and $193 = 256 - 64 + 1$, which is the same security level as SHA2-256.

Meet-in-the-Middle Preimage Attack. For the DM-based SFIL hash functions by using Areion-256 and Areion-512 as the underlying permutations, respectively, it is necessary to take into account Sasaki's meet-in-the-middle (MITM) preimage attack [Sas11]. This attack is the most powerful preimage attack on such hash functions. Indeed, the designers of Haraka v2 have evaluated its resistance against this attack in a dedicated way. To better understand the security of our constructions, we also performed a careful analysis. We found preimage attacks on 5-round Areion256-DM and 10-round Areion512-DM, respectively. Therefore, there is still a sufficiently large security margin. We detail our analysis below.

To save space, we only describe the general procedure of Sasaki's meet-in-the-middle preimage attack, as shown below:

Step 1: Identify the bytes that are fixed to constants and assign proper values to them.

Step 2: Identify the bytes that are to be exhausted. Classify them into backward neutral bytes and forward neutral bytes.

Step 3: In the forward direction, we assume that the backward neutral bytes are unknown and compute the internal state values based on the constant bytes and the forward neutral bytes. In other words, we only compute the bytes that can be computed from the knowledge of the constant bytes and the forward neutral bytes. This step is repeated for all the possible values of the forward neutral bytes, and we store the corresponding computed information.

Step 4: In the backward direction, we assume that the forward neutral bytes and unknown, and we only compute the bytes that can be computed from the knowledge of the constant bytes and the backward neutral bytes. This step is repeated for all the

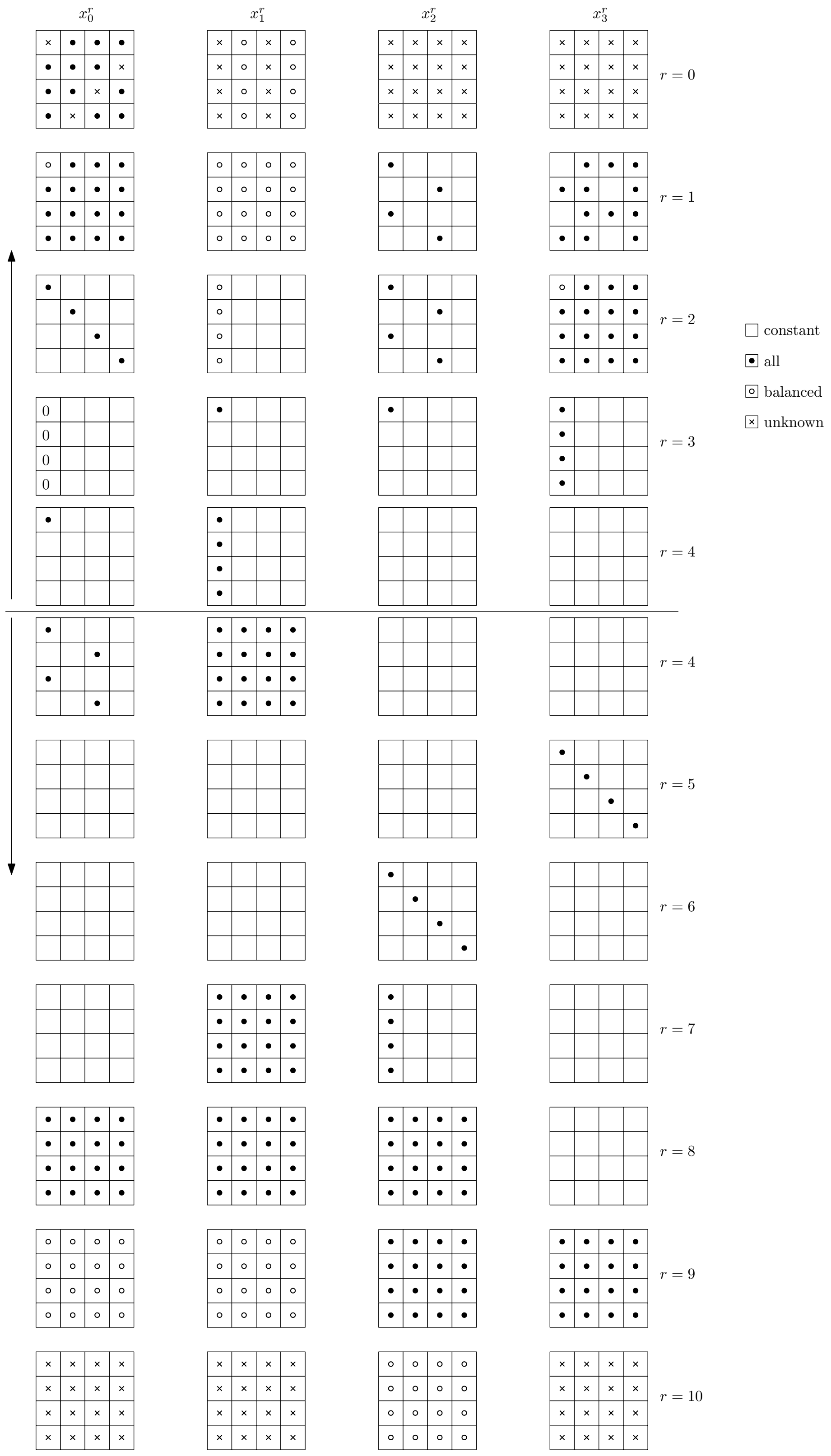


Figure 5: The zero-sum distinguisher for 10-round Areion-512.

possible values of the backward neutral bytes, and we store the corresponding computed information[3].

Step 5: Find matches between the store information obtained at Step 3 and Step 4. Suppose the matching probability is 2^{-p} and there are 2^{b_f} and 2^{b_b} possible values for the forward neutral bytes and the backward neutral bytes, respectively. Moreover, for each obtained state information at Step 3, if it is possible to identify the matched information obtained at Step 4 with time complexity 1, or vice versa, we can say that we find $2^{b_f+b_b-p}$ possible pairs among the $2^{b_f+b_b}$ pairs with time complexity $max(2^{b_f}, 2^{b_b})$ where usually $b_f + b_b - p \leq 0$. In other words, we exhaust $2^{b_f+b_b}$ possible candidates only with time complexity $max(2^{b_f}, 2^{b_b})$. Hence, the MITM preimage attack is $min(2^{b_b}, 2^{b_f})$ times faster than the brute force.

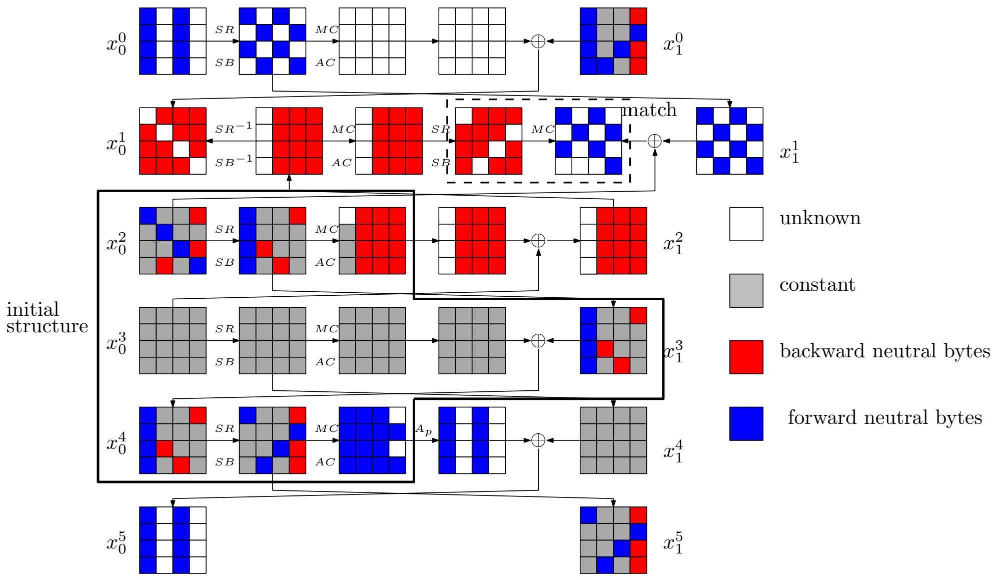


Figure 6: The preimage attack on 5-round Areion256-DM.

Hence, this attack aims to identify the forward and backward neutral bytes as well as an efficient matching method. For the two short-input hash schemes, we performed careful analysis and found preimage attacks on 5-round Areion256-DM and 10-round Areion512-DM, respectively. In the two attacks, $b_b = b_f = 8$ and the matching phase can be efficiently finished with time complexity 1. Hence, both the preimage attacks are 2^8 times faster than the brute force. The corresponding illustration of the two preimage attacks can be referred to Figs. 6 and 7, respectively.

Collision Attacks. The most powerful collision attack on AES-based hash functions is the rebound attack [MRST09], especially when it is built on the DM construction, as the attacker can fully control the whole internal state. However, as already mentioned in Haraka v2 and the SFIL hash function based on Simpira v2, the collision resistance of SFIL hash schemes is not necessary when they are used in the signature scheme, which is also the case of our SFIL hash functions.

[3]Note that in the actual implementations, we only need to store either the information obtained at Step 3 or Step 4. For simple explanations, we assume both are stored.

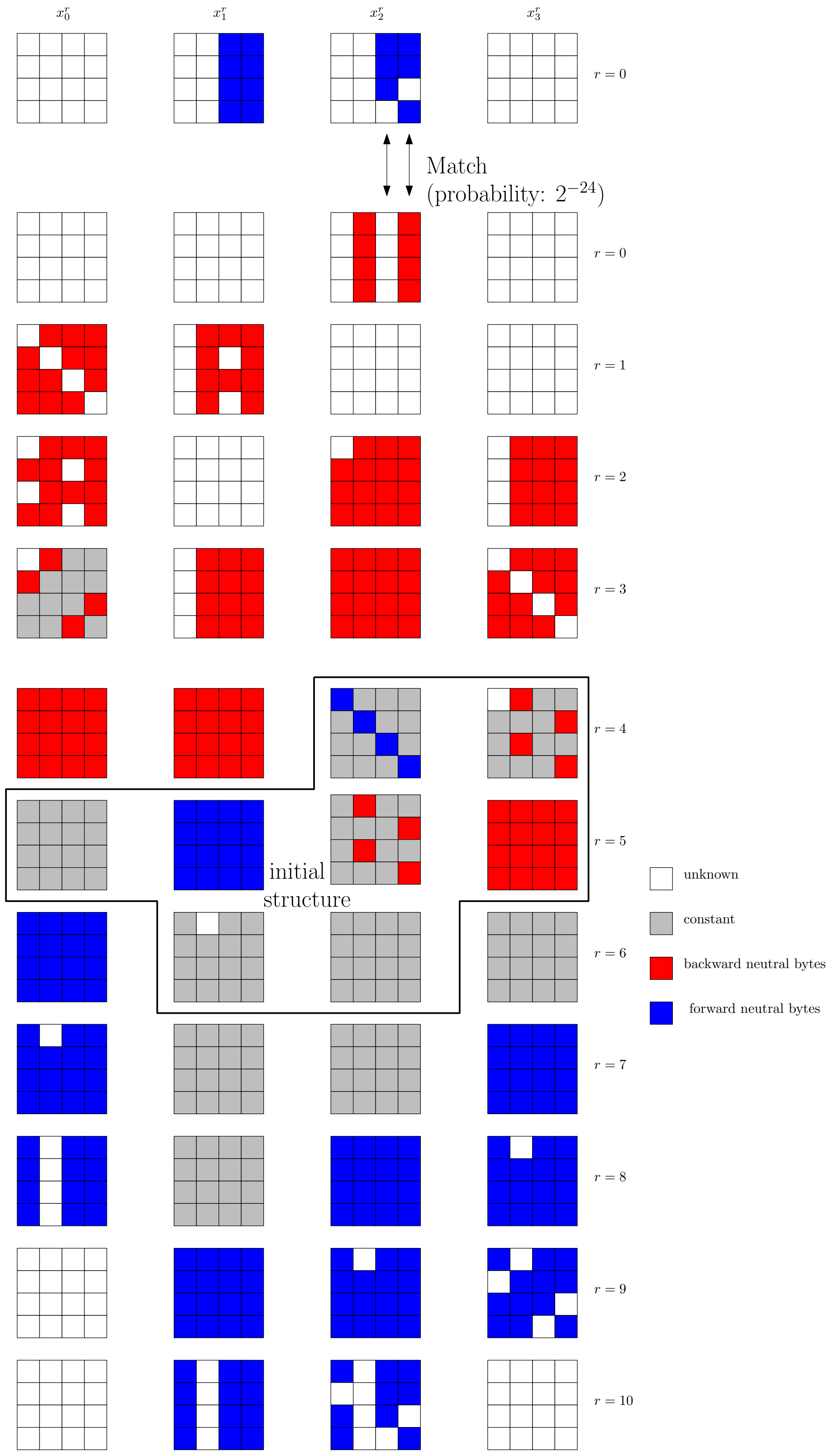


Figure 7: The preimage attack on 10-round Areion512-DM.

Security of MD Construction. For our hash scheme built on the MD construction, the attacker will soon lose the capability to fully control the internal state since each message block is only 256 bits, *i.e.*, half of the state size. However, by using $j > 1$ message blocks, Sasaki's MITM attack can still be applied in the same way as in the attack on the DM constructions. Specifically, although the 256-bit initial value set at (x_2, x_3) in the first input state is fixed, the attackers can view the 256-bit chaining variable (CV) in the last input state as a controllable part. Then, Sasaki's MITM attack is applied, and we aim to find 2^i solutions of the last input state to match the given hash value in less than 2^{256} time. This way, 2^i candidates of CV in the last input state can be obtained. Finally, we randomly pick values for the first $j-1$ message blocks to compute the corresponding CV for the last input state and expect that one such CV can match one of the 2^i candidates obtained by the MITM attack. Hence, we need to try 2^{256-i} different values for the first $j-1$ message blocks, and the time complexity is below 2^{256}.

For the collision resistance, we consider the rebound attack, the most efficient technique for AES-based hash functions. In particular, the most powerful rebound attack is always based on the Super-Sbox technique [GP10, LMR+09]. For such a technique, the attacker can control the difference transitions over two consecutive AES rounds with a pre-computation phase called the inbound phase, as shown in Fig. 8. Combined with the feature of the

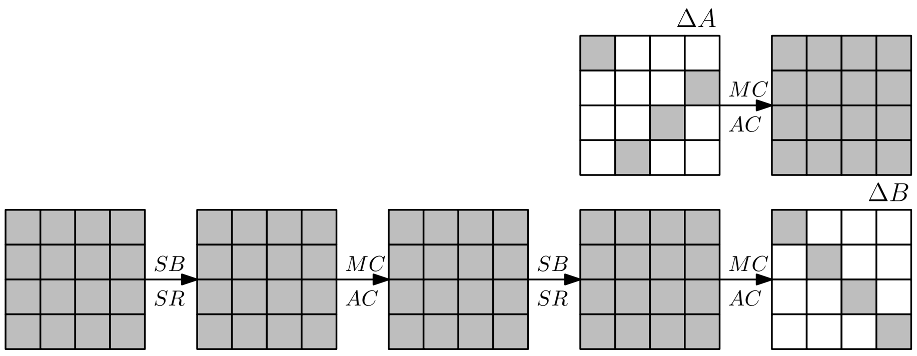


Figure 8: The inbound phase: precomputing the pairs $(A, \Delta A)$ such that $\Delta A \to \Delta B$ holds with probability 1.

rebound attack, this technique allows the attacker to ignore the influence of $4+16+16+4 = 38$ active S-boxes by using 128 free bits. Since the size of one message block in our VIL hash function is 256 bits, we expect that the attacker can ignore $38 \times 2 = 76$ active S-boxes with the Super-Sbox technique. However, we emphasize that it does not necessarily imply that the attacker can always ignore 76 active S-boxes in the actual attack because the rebound attack is also a start-from-the-middle-style attack, and one should be careful of the consistency in the CV.

According to Table 7, the minimal number of active S-boxes in 11-round Areion-512 is 119. By ignoring 76 active S-boxes, there are still $119 - 76 = 43$ active S-boxes left. In the outbound phase, we usually need to cancel the truncated differences. In the best case, we only need to consider half of the left active S-boxes, *i.e.*, we know the propagation of the truncated differences, and we only add conditions on the sum of the two truncated differences, as shown in Fig. 9. Even if we only consider $43/2 \approx 21$ active S-boxes, they still correspond to a very low uncontrolled probability of $2^{-21\times 8} = 2^{-168}$. Note that we have not yet taken into account the extra conditions on the truncated input and output differences to generate a collision. If they are considered, the truncated differential may be worse (*i.e.*, there are more active S-boxes), and the uncontrolled probability may further decrease. Hence, we believe the VIL hash function based on the 15-round Areion-512 is secure against the collision attack.

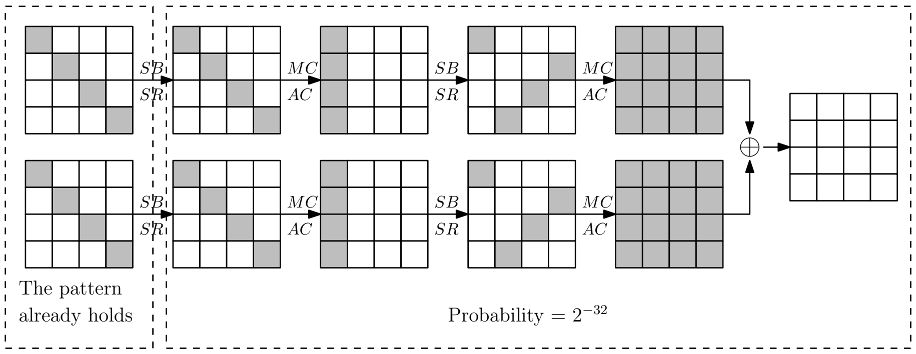


Figure 9: Cancel the truncated differences.

We also note that there is a variant method [JNP12] of the 2-round Super-Sbox technique that can cover three consecutive AES rounds, which can allow the attackers to ignore the influence of $4+16+16+16+4 = 54$ active S-boxes. However, this technique does not come for free. Specifically, different from the 2-round Super-Sbox technique to satisfy $4+16+16+4 = 38$ active S-boxes where lots of degrees of freedom are left after this phase, there is no degrees of freedom left after performing such a 3-round Super-Sbox technique and finding a solution to satisfy these 54 active S-boxes succeeds with probability 2^{-64}. In other words, it is like 2-round Super-Sbox technique with satisfying extra 16 active S-boxes with a probability of only 2^{-64}, which is a huge improvement over the 2-round Super-Sbox technique. We also note that it is almost equivalent to our conservative estimation that we only need to consider half of the remaining active S-boxes at the outbound phase when using the 2-round Super-Sbox technique for the inbound phase.

6 Performance Evaluation

In this section, we evaluate the performance of both Areion and its applications to the permutation-based hash functions described in Sect. 4. To this end, we used the available source code at GitHub[4] to evaluate the cycle counters, *i.e.*, cycles per byte (cpb), in the target primitive. All our evaluations were performed on the following widely deployed platforms: the Ice Lake, Tiger Lake, and Alder Lake platforms. More precisely, the Ice Lake platform has an Intel(R) Core(TM) i7-1068NG7 CPU @ 2.30GHz. The Tiger Lake platform has an Intel(R) Core(TM) i7-1165G7 CPU @ 2.80GHz. The Alder Lake platform has an Intel(R) Core(TM) i9-12900K CPU @ 3.20GHz on a performance-core (P-core) and 2.40GHz on an efficient-core (E-core). Turbo Boost technology has been switched off for all our evaluations. We note here that the P-core has been specified for our evaluations on the Alder Lake platform because there is almost no difference in the benchmarks between using either the P-core or E-core.

Besides, we also evaluate the performance of NEON implementations of permutation-based hash functions proposed in Sect. 4 in several mobile environments. To keep the page limit, the NEON implementations of Areion is shown in Appendix A.3.

6.1 Underlying Permutations

We first evaluate the performance of the underlying permutations, *i.e.*, Areion-256 and Areion-512. These implementations are given in Appendix A.1. For comparison, we used the underlying permutations of Simpira v2, Haraka v2, and the 512-bit permutation BLAKE2s.

[4]https://github.com/seb-m/cycles

Table 8: Benchmarks for single block encryption/decryption on the Ice Lake, Tiger Lake, and Alder Lake platforms. All values are given as cpb.

	Ice Lake		Tiger Lake		Alder Lake	
Primitive	Enc	Dec	Enc	Dec	Enc	Dec
Areion-256	**1.92**	**2.84**	**1.91**	**2.83**	**1.93**	**2.81**
Simpira-256	2.94	2.94	2.92	2.92	2.94	2.94
Haraka-256	1.58	4.08	1.58	4.08	1.55	4.00
Haraka-256 (x1.2)	1.90	5.28	1.90	5.28	1.86	4.80
Haraka-256 (x1.5)	2.37	6.12	2.37	6.12	2.32	6.00
Areion-512	**1.09**	**2.52**	**1.09**	**2.52**	**1.09**	**2.52**
Simpira-512	1.47	1.47	1.46	1.46	1.47	1.47
Haraka-512	1.06	2.58	1.06	2.58	1.09	2.58
Haraka-512 (x1.2)	1.27	3.10	1.27	3.10	1.31	3.10
Haraka-512 (x1.5)	1.59	3.87	1.59	3.87	1.63	3.87
BLAKE2s	3.05	1.83	3.04	1.83	3.04	1.80

Table 9: Benchmarks for parallel block encryption/decryption on the Ice Lake, Tiger Lake, and Alder Lake platforms. All values are given as cpb.

	Ice Lake		Tiger Lake		Alder Lake	
Primitive	Enc	Dec	Enc	Dec	Enc	Dec
Areion-256	**0.55**	**0.66**	**0.55**	**0.66**	**0.51**	**0.56**
Simpira-256	0.69	0.69	0.68	0.68	0.57	0.57
Haraka-256	0.53	1.75	0.54	1.74	0.44	1.52
Haraka-256 (x1.2)	0.64	2.10	0.65	2.09	0.53	1.83
Haraka-256 (x1.5)	0.79	2.62	0.81	2.61	0.66	2.28
Areion-512	**0.64**	**1.24**	**0.63**	**1.25**	**0.61**	**1.13**
Simpira-512	0.63	0.62	0.62	0.61	0.53	0.53
Haraka-512	0.67	2.06	0.66	2.04	0.64	1.83
Haraka-512 (x1.2)	0.81	2.48	0.80	2.45	0.77	2.20
Haraka-512 (x1.5)	1.00	3.09	0.99	3.06	0.96	2.74
BLAKE2s	2.51	1.64	2.51	1.64	2.29	1.59

We can find the source codes of Haraka v2 and BLAKE2s available at GitHub[5,6], but we could not find the available source code for Simpira v2. For this reason, we implemented it as described in Appendix A.2.

According to [GM16, KLMR16], Simpira v2 and Haraka v2 are supposed to operate on multiple message blocks, not just a single message block, to get the highest performance. Based on this concept, we also evaluate the performance when operating on eight message blocks in parallel as well as a single message block.

Tables 8 and 9 show benchmarks for single and parallel encryption/decryption on our platforms. From these tables, Haraka v2 appears to be the fastest encryption, but it cannot be regarded to have a security margin sufficiently, as discussed in Sect. 3.2.1. For this reason, we consider there is no problem even if Haraka v2 is excluded from our comparison. Instead of the original Haraka v2, we use the 12/15-round variants of Haraka v2, Haraka-256 (x1.2/x1.5) and Haraka-512 (x1.2/x1.5), for our comparison. This is because DM-based instantiations of the tweaked variants, Haraka256-DM (x1.2/x1.5) and Haraka512-DM (x1.2/x1.5), can be regarded to have a similar security level as Areion256-DM and Areion512-DM. Indeed, the security margins against MITM preimage attacks of Areion256-DM, Haraka256-DM (x1.2), and Haraka256-DM (x1.5) are 5, 3, and 6, respectively. Similarly, the security margins of Areion512-DM, Haraka512-DM (x1.2), and Haraka512-

[5] https://github.com/kste/haraka
[6] https://github.com/BLAKE2/BLAKE2

DM (x1.5) are 5, 1, and 4, respectively. We summarize the performance comparison for the underlying permutations as follows:

- Areion-256 realizes the fastest encryption among the target permutations, excluding Haraka-256 (x1.2) for single block encryption (although there are almost no differences in performance). Specifically, Areion-256 performs at least 1.52 and 1.20 times faster than Simpira-256 and Haraka-256 (x1.5) for single block encryption, respectively, and at least 1.12 and 1.03 times faster than Simpira-256 and Haraka-256 (x1.2) for parallel block encryption, respectively. On the other hand, for single and parallel block decryptions, Areion-256 performs faster than Haraka-256 (x1.2/x1.5), but there are almost no differences in performance between Areion-256 and Simpira-256.

- Areion-512 realizes the fastest encryption among the target permutations, excluding Simpira-512 for parallel block encryption (although there are almost no differences in performance). Specifically, Areion-512 performs at least 1.34 and 1.16 times faster than Simpira-512 and Haraka-512 (x1.2) for single block encryption, respectively, and at least 1.26 times faster than Haraka-512 (x1.2) for parallel block encryption. On the other hand, Areion-512 performs faster than Haraka-256 (x1.2/x1.5) and BLAKE2s, especially for parallel block decryption, but it performs at least 2.00 times slower than Simpira-512.

Given that the Areion-512 decryption function is not used for the proposed applications of Areion described in Sect. 4, we consider that there is no problem even if Areion-512 performs slower than Simpira-512 for decryption. Therefore, Areion has the strongest advantage of performing faster than any other target permutations, especially in terms of encryption direction.

Regarding the advantage of Areion-256 over Areion-512, Table 9 suggests that Areion-256 is consistently faster than Areion-512 for parallel processing and even the fastest among all the selected 256-/512-bit permutations in many cases. In addition, it has a balanced performance for encryption and decryption thanks to its Feistel-like structure, unlike Haraka-256, and faster than the Feistel-based Simpira-256. That is, it should work more efficiently with the existing parallelizable permutation-based authenticated encryption modes, *e.g.*, OPP [GJMN16] and a permutation-based counterpart of OTR [Min14] than other permutations. The latter would be similar to Prøst-OTR [KLL$^+$14] adopting the masking scheme of OPP for provable security and for avoiding the attack specific to (the masking scheme of) Prøst-OTR [DEM15]. Its applications to the parallel authenticated encryption modes are left as our future work. On the other hand, Areion-512 is the fastest among the selected 256-/512-bit permutations for single block encryption direction (Table 8). That is, it should work more efficiently with the existing permutation-based compression functions such as DM construction, and the existing sequential hash functions such as MD construction. These are the target applications for this study.

6.2 Permutation-based Hash Functions

Next, we evaluate the performance of the permutation-based hash functions, *i.e.*, the SFIL and VIL hash functions (DM and MD constructions). These instantiations of Areion are implemented based on the source codes of Areion-256 and Areion-512 described in Appendix A.1. For comparison regarding the SFIL hash functions, we used DM constructions instantiated with Simpira v2 and Haraka v2. On the other hand, for comparison regarding the VIL hash functions, we used AES-based VIL hash functions, such as Simpira512-MD, Haraka512-MD, and double-block-length hash functions proposed by Hirose at FSE 2006 [Hir06]. We refer to the Hirose's hash function as Hirose-DBL. These instantiations are also implemented similarly to those of Areion. In addition, we used SHA2-256, SHA3-256,

Table 10: Benchmarks for SFIL hash functions on the Ice Lake, Tiger Lake, and Alder Lake platforms. All values are given as cpb.

Primitive	Ice Lake	Tiger Lake	Alder Lake
Areion256-DM	**2.01**	**2.01**	**1.99**
Simpira256-DM	2.84	2.83	2.81
Haraka256-DM	1.64	1.63	1.61
Haraka256-DM (x1.2)	1.97	1.96	1.94
Haraka256-DM (x1.5)	2.46	2.44	2.41
Areion512-DM	**1.05**	**1.05**	**1.04**
Simpira512-DM	1.41	1.41	1.40
Haraka512-DM	1.12	1.10	1.13
Haraka512-DM (x1.2)	1.35	1.32	1.36
Haraka512-DM (x1.5)	1.68	1.65	1.69

ParallelHash256, KangarooTwelve, and BLAKE3. We can find these source codes available at GitHub[7,8,9]; then, we modified these source codes to use for our comparison.

Tables 10 and 11 show benchmarks for the SFIL and VIL hash functions on our platforms. From Table 10, Haraka512-DM appears to be the fastest SFIL hash function, but Haraka v2 cannot be regarded to have the security margin sufficiently; thus, we use Haraka256-DM (x1.2/x1.5) and Haraka512-DM (x1.2/x1.5) for our comparison regarding the SFIL hash functions, as discussed in Sect. 6.1. Similarly, we use Haraka512-MD (x1.2/x1.5) for our comparison regarding the VIL hash functions. We summarize the performance comparison for the SFIL hash functions as follows:

- Areion256-DM realizes the fastest SFIL hashing among the target DM constructions with the 256-bit permutation, excluding Haraka256-DM (x1.2) (although there are almost no differences in performance). Specifically, Areion256-DM performs at least 1.41 and 1.21 times faster than Simpira256-DM and Haraka256-DM (x1.5), respectively.

- Areion512-DM realizes the fastest SFIL hashing among the target DM constructions with the 512-bit permutation. Specifically, Areion512-DM performs at least 1.34 and 1.25 times faster than Simpira256-DM and Haraka256-DM (x1.2), respectively.

Consequently, It can be considered that Areion256-DM and Areion512-DM are the fastest SFIL hash functions. On the other hand, we summarize the performance comparison for the VIL hash functions as follows:

- Areion512-MD realizes the fastest VIL hashing among the target hash functions with a 256-bit security level for input sizes up to around 4K bytes. Specifically, its performance is less than 3 cpb for any message size. Moreover, it is about 10 times faster than existing state-of-the-art schemes (*e.g.*, SHA2-256, SHA3-256, and ParallelHash256) for short messages up to around 100 bytes, widely-used input size in real-world applications.

Considering the need for cryptographic primitives resistant to symmetric-key cryptanalysis based on quantum algorithms (*e.g.*, Grover's algorithm [Gro96]), hash functions with a 256-bit security level must be required for the future. For this reason, we consider that there is no problem even if Areion512-MD performs slower than KangarooTwelve when the input size is 2K bytes or more. In addition, according to the current study on packet sizes on the Internet [MKZ+17], it is known that around 44% of packets are between 40 and 100 bytes long and 37% are between 1400 and 1500 bytes in size. Given that most of

[7] https://github.com/wereHamster/sha256-sse
[8] https://github.com/XKCP/XKCP
[9] https://github.com/BLAKE3-team/BLAKE3

Table 11: Benchmarks for VIL hash functions on the Ice Lake, Tiger Lake, and Alder Lake platforms. All values are given as cpb.

		Security		Input sizes (bytes)						
Platform	Primitive	level[†]	Impl.	64	128	256	512	1024	2048	4096
Ice Lake	**Areion512-MD**	256	AES-NI	**1.99**	**2.35**	**2.54**	**2.60**	**2.63**	**2.65**	**2.66**
	Simpira512-MD	256	AES-NI	2.03	2.65	3.04	3.25	3.34	3.38	3.40
	Haraka512-MD	256	AES-NI	2.12	2.47	2.57	2.66	2.69	2.71	2.72
	Haraka512-MD (x1.2)	256	AES-NI	2.55	2.96	3.09	3.19	3.23	3.25	3.26
	Haraka512-MD (x1.5)	256	AES-NI	3.18	3.70	3.86	3.99	4.03	4.07	4.08
	Hirose-DBL	256	AES-NI	12.29	12.21	12.21	12.34	12.27	12.28	12.27
	SHA2-256	256	SSE	23.90	23.57	23.42	23.74	23.71	23.32	23.31
	SHA3-256	256	AVX2	18.69	9.48	9.33	9.13	9.05	9.03	8.70
	SHA3-256	256	AVX512	13.20	6.66	6.47	6.20	6.06	5.99	5.78
	ParallelHash256	256	AVX2	61.09	30.65	19.88	14.43	11.71	10.33	9.37
	ParallelHash256	256	AVX512	42.61	21.39	13.89	9.90	7.91	6.92	6.24
	BLAKE3	128	SSE	57.77	32.10	18.64	11.61	7.93	4.93	2.94
	KangarooTwelve	128	AVX2	11.59	5.96	5.27	4.86	4.15	3.79	3.61
	KangarooTwelve	128	AVX512	8.16	4.17	3.78	3.34	2.79	2.52	2.38
Tiger Lake	**Areion512-MD**	256	AES-NI	**1.89**	**2.31**	**2.50**	**2.57**	**2.61**	**2.63**	**2.64**
	Simpira512-MD	256	AES-NI	1.98	2.60	3.01	3.22	3.32	3.37	3.40
	Haraka512-MD	256	AES-NI	2.09	2.44	2.57	2.64	2.68	2.71	2.72
	Haraka512-MD (x1.2)	256	AES-NI	2.51	2.92	3.08	3.16	3.22	3.26	3.27
	Haraka512-MD (x1.5)	256	AES-NI	3.14	3.65	3.85	3.95	4.02	4.07	4.08
	Hirose-DBL	256	AES-NI	12.30	12.21	12.24	12.34	12.24	12.24	12.24
	SHA2-256	256	SSE	23.85	23.56	23.46	23.69	23.66	23.29	23.27
	SHA3-256	256	AVX2	18.67	9.46	9.30	9.14	9.05	8.99	8.70
	SHA3-256	256	AVX512	13.14	6.63	6.44	6.17	6.02	5.99	5.74
	ParallelHash256	256	AVX2	61.04	30.62	19.86	14.40	11.67	10.35	9.36
	ParallelHash256	256	AVX512	42.40	21.77	13.83	9.98	7.86	6.87	6.20
	BLAKE3	128	SSE	57.63	32.17	18.70	11.66	7.96	4.96	2.94
	KangarooTwelve	128	AVX2	11.60	5.93	5.30	4.87	4.16	3.78	3.61
	KangarooTwelve	128	AVX512	8.22	4.20	3.78	3.33	2.78	2.50	2.36
Alder Lake[‡]	**Areion512-MD**	256	AES-NI	**1.60**	**2.16**	**2.42**	**2.60**	**2.66**	**2.68**	**2.70**
	Simpira512-MD	256	AES-NI	1.65	2.30	2.87	3.19	3.32	3.39	3.42
	Haraka512-MD	256	AES-NI	1.68	2.15	2.41	2.55	2.62	2.65	2.67
	Haraka512-MD (x1.2)	256	AES-NI	2.02	2.58	2.90	3.05	3.14	3.18	3.21
	Haraka512-MD (x1.5)	256	AES-NI	2.52	3.23	3.62	3.82	3.92	3.97	4.01
	Hirose-DBL	256	AES-NI	12.67	12.61	12.58	12.59	12.61	12.61	12.61
	SHA2-256	256	SSE	21.46	21.72	21.58	21.52	21.49	21.48	21.47
	SHA3-256	256	AVX2	18.56	9.36	9.23	9.08	9.01	9.00	8.71
	SHA3-256	256	AVX512	–	–	–	–	–	–	–
	ParallelHash256	256	AVX2	59.38	29.47	19.48	14.25	11.62	10.30	9.36
	ParallelHash256	256	AVX512	–	–	–	–	–	–	–
	BLAKE3	128	SSE	55.24	30.76	17.99	11.25	7.72	4.50	2.79
	KangarooTwelve	128	AVX2	10.65	5.40	4.99	4.73	4.06	3.73	3.57
	KangarooTwelve	128	AVX512	–	–	–	–	–	–	–

[†] The security level is against preimage attacks.
[‡] Our Alder Lake platform does not support the AVX512 instruction set.

the packet sizes on the Internet are 1.5K bytes or less, Areion512-MD has the strongest advantage of performing faster than any other target VIL hash functions with a 256-bit security level.

Tables 12 shows benchmarks for the VIL hash functions using NEON implementations in Appendix A.3 on mobile environments. We compare with existing schemes of SHA2-256 and SHA3-256 which are available for optimized implementations in OpenSSL. Areion512-MD achieves outstanding performance for short messages, especially up to 128 bytes.

7 Conclusion

We proposed a family of wide-block permutations Areion that fully leverages the power of AES instructions and show its applications of hash functions. Our schemes significantly outperform existing schemes for short input and are competitive for relatively-long messages.

Table 12: Benchmarks for hash functions on the Pixel5, Pixel6, iPhone13, iPadPro. All values are given as Gbps.

		Input sizes (bytes)							
Platform	Primitive	32	64	128	256	512	1024	2048	4096
Pixel 5 (Snapdragon765G)	**Areion512-MD**	**3.99**	**4.38**	**4.82**	**5.07**	**5.22**	**5.29**	**5.33**	**5.33**
	SHA2-256	0.33	0.63	1.23	2.21	3.69	5.57	7.42	8.96
	SHA3-256	0.20	0.40	0.80	1.17	1.51	1.76	1.93	2.09
Pixel 6 (Google Tensor)	**Areion512-MD**	**6.07**	**7.28**	**7.20**	**7.19**	**7.18**	**7.18**	**7.18**	**7.18**
	SHA2-256	0.45	0.86	1.62	2.87	4.71	6.94	9.07	10.72
	SHA3-256	0.28	0.56	1.12	1.66	2.19	2.59	2.86	3.11
iPhone 13 (A15)	**Areion512-MD**	**8.39**	**14.71**	**13.84**	**11.15**	**10.19**	**9.75**	**9.56**	**9.46**
	SHA2-256	0.96	1.81	3.44	6.00	9.08	12.19	14.70	16.42
	SHA3-256	0.47	0.96	1.97	2.78	3.51	3.98	4.33	4.67
iPad Pro (Apple M1)	**Areion512-MD**	**8.39**	**14.98**	**14.38**	**11.74**	**10.75**	**10.31**	**10.11**	**10.00**
	SHA2-256	0.49	1.81	3.40	5.92	8.68	12.03	14.48	16.19
	SHA3-256	0.47	0.95	1.98	2.76	3.49	3.92	4.33	4.56

Our hash function is surprisingly fast. Its performance is less than 3 cycle/byte in the latest Intel architectures for any message size. It is about 10 times faster than existing schemes for short messages up to around 100 bytes, which are the most widely-used input size in real-world applications, on both of on latest CPU architectures (IceLake, Tiger Lake, and Alder Lake) and mobile environments (Pixel 6 and iPhone 13).

Acknowledgments

Takanori Isobe is supported by JST, PRESTO Grant Number JPMJPR2031. These research results were also obtained from the commissioned research (No.05801) by National Institute of Information and Communications Technology (NICT), Japan. Fukang Liu is supported by Grant-in-Aid for Research Activity Start-up (Grant No. 22K21282). Kosei Sakamoto is supported by Grant-in-Aid for JSPS Fellows (KAKENHI 20J23526) for Japan Society for the Promotion of Science.

References

[ALP+19] Elena Andreeva, Virginie Lallemand, Antoon Purnal, Reza Reyhanitabar, Arnab Roy, and Damian Vizár. Forkcipher: A New Primitive for Authenticated Encryption of Very Short Messages. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part II*, volume 11922 of *Lecture Notes in Computer Science*, pages 153–182. Springer, 2019. doi:10.1007/978-3-030-34621-8_6.

[AM09] Jean-Philippe Aumasson and Willi Meier. Zero-sum Distinguishers for Reduced Keccak-f and for the Core Functions of Luffa and Hamsi. 2009. https://131002.net/data/papers/AM09.pdf.

[BDG+21] Zhenzhen Bao, Xiaoyang Dong, Jian Guo, Zheng Li, Danping Shi, Siwei Sun, and Xiaoyun Wang. Automatic Search of Meet-in-the-Middle Preimage Attacks on AES-like Hashing. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 771–804. Springer, 2021. doi:10.1007/978-3-030-77870-5_27.

[BDP$^+$18] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. KangarooTwelve: Fast Hashing Based on Keccak-p. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 400–418. Springer, 2018. doi:10.1007/978-3-319-93387-0_21.

[CDL$^+$20] Anne Canteaut, Sébastien Duval, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Thomas Pornin, and André Schrottenloher. Saturnin: a suite of lightweight symmetric algorithms for post-quantum security. *IACR Trans. Symmetric Cryptol.*, 2020(S1):160–207, 2020. doi:10.13154/tosc.v2020.iS1.160-207.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[CJF$^+$16] Tingting Cui, Keting Jia, Kai Fu, Shiyao Chen, and Meiqin Wang. New Automatic Search Tool for Impossible Differentials and Zero-Correlation Linear Approximations. *IACR Cryptol. ePrint Arch.*, page 689, 2016. URL: http://eprint.iacr.org/2016/689.

[Dam89] Ivan Damgård. A Design Principle for Hash Functions. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1989. doi:10.1007/0-387-34805-0_39.

[DEM15] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Related-Key Forgeries for Prøst-OTR. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 282–296. Springer, 2015. doi:10.1007/978-3-662-48116-5_14.

[DEMS19] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to Round 1 of the NIST Lightweight Cryptography project, 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ascon-spec.pdf.

[Gil14] Henri Gilbert. A simplified representation of AES. In *ASIACRYPT (1)*, volume 8873 of *Lecture Notes in Computer Science*, pages 200–222. Springer, 2014.

[GJMN16] Robert Granger, Philipp Jovanovic, Bart Mennink, and Samuel Neves. Improved Masking for Tweakable Blockciphers with Applications to Authenticated Encryption. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 263–293. Springer, 2016. doi:10.1007/978-3-662-49890-3_11.

[GM16] Shay Gueron and Nicky Mouha. Simpira v2: A Family of Efficient Permutations Using the AES Round Function. In Jung Hee Cheon and Tsuyoshi

Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 95–125, 2016. doi:10.1007/978-3-662-53887-6_4.

[GP10] Henri Gilbert and Thomas Peyrin. Super-sbox cryptanalysis: Improved attacks for aes-like permutations. In *FSE*, volume 6147 of *Lecture Notes in Computer Science*, pages 365–383. Springer, 2010.

[Gro96] Lov K. Grover. A Fast Quantum Mechanical Algorithm for Database Search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996. doi:10.1145/237814.237866.

[Hir06] Shoichi Hirose. Some Plausible Constructions of Double-Block-Length Hash Functions. In Matthew J. B. Robshaw, editor, *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2006. doi:10.1007/11799313_14.

[IIM21] Takanori Isobe, Ryoma Ito, and Kazuhiko Minematsu. Security Analysis of SFrame. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *Computer Security - ESORICS 2021 - 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4-8, 2021, Proceedings, Part II*, volume 12973 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 2021. doi:10.1007/978-3-030-88428-4_7.

[IKMP20] Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. Duel of the Titans: The Romulus and Remus Families of Lightweight AEAD Algorithms. *IACR Trans. Symmetric Cryptol.*, 2020(1):43–120, 2020. doi:10.13154/tosc.v2020.i1.43-120.

[JNP12] Jérémy Jean, María Naya-Plasencia, and Thomas Peyrin. Improved rebound attack on the finalist grøstl. In Anne Canteaut, editor, *Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers*, volume 7549 of *Lecture Notes in Computer Science*, pages 110–126. Springer, 2012. doi:10.1007/978-3-642-34047-5_7.

[Jos21] Josh Blum and Simon Booth and Oded Gal and Maxwell Krohn and Julia Len and Karan Lyons and Antonio Marcedone and Mike Maxim and Merry Ember Mou and Jack O'Connor and Miles Steele and Matthew Green and Lea Kissner and Alex Stamos. E2E Encryption for Zoom Meetings – Version 3.2, October 2021. https://github.com/zoom/zoom-e2e-whitepaper.

[KjCP16] John Kelsey, Shu jen Chang, and Ray Perlner. SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash. *NIST special publication*, 800:185, 2016.

[KLL$^+$14] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe, and Tolga Yalçın. Prøst. CAESAR Proposal, 2014. http://proest.compute.dtu.dk.

[KLMR16] Stefan Kölbl, Martin M. Lauridsen, Florian Mendel, and Christian Rechberger. Haraka v2 - Efficient Short-Input Hashing for Post-Quantum Applications. *IACR Trans. Symmetric Cryptol.*, 2016(2):1–29, 2016. doi:10.13154/tosc.v2016.i2.1-29.

[KR07] Lars R. Knudsen and Vincent Rijmen. Known-key distinguishers for some block ciphers. In *ASIACRYPT*, volume 4833 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 2007.

[KS05] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[LaL19] Matti Latva-aho and Kari Leppänen. Key drivers and research challenges for 6g ubiquitous wireless intelligence. 6G Research Visions 1, 2019.

[LMR$^+$09] Mario Lamberger, Florian Mendel, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. Rebound distinguishers: Results on the full whirlpool compression function. In *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2009.

[MBTM17] Kerry A. McKay, Larry Bassham, Meltem Sönmez Turan, and Nicky Mouha. Report on Lightweight Cryptography, 2017. National Institute of Standards and Technology IR 8114. URL: https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf.

[Mer89] Ralph C. Merkle. One Way Hash Functions and DES. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1989. doi:10.1007/0-387-34805-0_40.

[Min14] Kazuhiko Minematsu. Parallelizable Rate-1 Authenticated Encryption from Pseudorandom Functions. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2014. doi:10.1007/978-3-642-55220-5_16.

[MKZ$^+$17] David Murray, Terry Koziniec, Sebastian Zander, Michael Dixon, and Polychronis Koutsakis. An Analysis of Changing Enterprise Network Traffic Characteristics. In *23rd Asia-Pacific Conference on Communications, APCC 2017, Perth, Australia, December 11-13, 2017*, pages 1–6. IEEE, 2017. doi:10.23919/APCC.2017.8303960.

[MNP$^+$21] Ben Marshall, G. Richard Newell, Dan Page, Markku-Juhani O. Saarinen, and Claire Wolf. The design of scalar AES Instruction Set Extensions for RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):109–136, 2021. doi:10.46586/tches.v2021.i1.109-136.

[MRST09] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The rebound attack: Cryptanalysis of reduced whirlpool and grøstl. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2009. doi:10.1007/978-3-642-03317-9_16.

[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. doi:10.1007/978-3-642-34704-7_5.

[NBI19] 3GPP TS 36.213: Evolved Universal Terrestrial Radio Access (E-UTRA); Physical layer procedures. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2427, 2019.

[OANWO20] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. BLAKE3: One Function, Fast Everywhere, 2020. https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf.

[oST15a] National Institute of Standards and Technology. Secure Hash Standard (SHS). *Federal Information Processing Standards Publication. FIPS PUB 180-4*, August 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[oST15b] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. *Federal Information Processing Standards Publication. FIPS PUB 202*, August 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[OUGM21] Emad Omara, Justin Uberti, Alexandre Gouaillard, and Sergio Garcia Murillo. Secure Frame (SFrame). https://tools.ietf.org/html/draft-omara-sframe-03, August 2021.

[RTL21] Real-Time and Embedded Sys Lab. uops.info. Official webpage, https://www.uops.info/, 2021.

[Sas11] Yu Sasaki. Meet-in-the-middle preimage attacks on AES hashing modes and an application to whirlpool. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 378–396. Springer, 2011. doi:10.1007/978-3-642-21702-9_22.

[SSI21] Rentaro Shiba, Kosei Sakamoto, and Takanori Isobe. Efficient constructions for large-state block ciphers based on AES New Instructions. *IET Information Security*, 2021. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/ise2.12053, doi:https://doi.org/10.1049/ise2.12053.

[ST17] Yu Sasaki and Yosuke Todo. New Impossible Differential Search Tool from Design and Cryptanalysis Aspects - Revealing Structural Properties of Several Ciphers. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 185–215, 2017. doi:10.1007/978-3-319-56617-7_7.

[Sys20] Cisco Systems. Zero-Trust Security for Cisco Webex, 2020. https://www.cisco.com/c/en/us/solutions/collateral/collaboration/white-paper-c11-744553.html.

[XZBL16] Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. Applying MILP Method to Searching Integral Distinguishers Based on Division Property for 6 Lightweight Block Ciphers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 648–678, 2016. doi:10.1007/978-3-662-53887-6_24.

A Reference Implementations

A.1 Areion-256 and Areion-512

```
#include <stdint.h>
#include <immintrin.h>

/* Round Constant */
const uint32_t RC[24*4] = {
   0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344,
   0xa4093822, 0x299f31d0, 0x082efa98, 0xec4e6c89,
   0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
   0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917,
   0x9216d5d9, 0x8979fb1b, 0xd1310ba6, 0x98dfb5ac,
   0x2ffd72db, 0xd01adfb7, 0xb8e1afed, 0x6a267e96,
   0xba7c9045, 0xf12c7f99, 0x24a19947, 0xb3916cf7,
   0x801f2e28, 0x58efc166, 0x36920d87, 0x1574e690,
   0xa458fea3, 0xf4933d7e, 0x0d95748f, 0x728eb658,
   0x718bcd58, 0x82154aee, 0x7b54a41d, 0xc25a59b5,
   0x9c30d539, 0x2af26013, 0xc5d1b023, 0x286085f0,
   0xca417918, 0xb8db38ef, 0x8e79dcb0, 0x603a180e,
   0x6c9e0e8b, 0xb01e8a3e, 0xd71577c1, 0xbd314b27,
   0x78af2fda, 0x55605c60, 0xe65525f3, 0xaa55ab94,
   0x57489862, 0x63e81440, 0x55ca396a, 0x2aab10b6,
   0xb4cc5c34, 0x1141e8ce, 0xa15486af, 0x7c72e993,
   0xb3ee1411, 0x636fbc2a, 0x2ba9c55d, 0x741831f6,
   0xce5c3e16, 0x9b87931e, 0xafd6ba33, 0x6c24cf5c,
   0x7a325381, 0x28958677, 0x3b8f4898, 0x6b4bb9af,
   0xc4bfe81b, 0x66282193, 0x61d809cc, 0xfb21a991,
   0x487cac60, 0x5dec8032, 0xef845d5d, 0xe98575b1,
   0xdc262302, 0xeb651b88, 0x23893e81, 0xd396acc5,
   0xf6d6ff38, 0x3f442392, 0xe0b4482a, 0x48420040,
   0x69c8f04a, 0x9e1f9b5e, 0x21c66842, 0xf6e96c9a
};

#define RC0(i) _mm_setr_epi32(RC[(i)*4+0], RC[(i)*4+1], RC[(i)*4+2], RC[(i)*4+3])
#define RC1(i) _mm_setr_epi32(0, 0, 0, 0)

/* Round Function for the 256-bit permutation */
#define Round_Function_256(x0, x1, i) do { \
   x1 = _mm_aesenc_si128(_mm_aesenc_si128(x0, RC0(i)), x1); \
   x0 = _mm_aesenclast_si128(x0, RC1(i)); \
} while(0)

/* 256-bit permutation */
#define perm256(x0, x1) do { \
   Round_Function_256(x0, x1, 0); \
   Round_Function_256(x1, x0, 1); \
   Round_Function_256(x0, x1, 2); \
   Round_Function_256(x1, x0, 3); \
   Round_Function_256(x0, x1, 4); \
   Round_Function_256(x1, x0, 5); \
   Round_Function_256(x0, x1, 6); \
   Round_Function_256(x1, x0, 7); \
   Round_Function_256(x0, x1, 8); \
   Round_Function_256(x1, x0, 9); \
} while(0)

/* Inversed Round Function for the 256-bit permutation */
#define Inv_Round_Function_256(x0, x1, i) do { \
   x0 = _mm_aesdeclast_si128(x0, RC1(i)); \
   x1 = _mm_aesenc_si128(_mm_aesenc_si128(x0, RC0(i)), x1); \
} while(0)

/* Inversed 256-bit permutation */
#define Inv_perm256(x0, x1) do { \
   Inv_Round_Function_256(x1, x0, 9); \
   Inv_Round_Function_256(x0, x1, 8); \
   Inv_Round_Function_256(x1, x0, 7); \
   Inv_Round_Function_256(x0, x1, 6); \
   Inv_Round_Function_256(x1, x0, 5); \
   Inv_Round_Function_256(x0, x1, 4); \
   Inv_Round_Function_256(x1, x0, 3); \
   Inv_Round_Function_256(x0, x1, 2); \
   Inv_Round_Function_256(x1, x0, 1); \
   Inv_Round_Function_256(x0, x1, 0); \
} while(0)

/* Round Function for the 512-bit permutation */
#define Round_Function_512(x0, x1, x2, x3, i) do { \
   x1 = _mm_aesenc_si128(x0, x1); \
   x3 = _mm_aesenc_si128(x2, x3); \
   x0 = _mm_aesenclast_si128(x0, RC1(i)); \
   x2 = _mm_aesenc_si128(_mm_aesenclast_si128(x2, RC0(i)), RC1(i)); \
} while (0)

/* 512-bit permutation */
#define perm512(x0, x1, x2, x3) do { \
   Round_Function_512(x0, x1, x2, x3, 0); \
   Round_Function_512(x1, x2, x3, x0, 1); \
   Round_Function_512(x2, x3, x0, x1, 2); \
   Round_Function_512(x3, x0, x1, x2, 3); \
   Round_Function_512(x0, x1, x2, x3, 4); \
   Round_Function_512(x1, x2, x3, x0, 5); \
   Round_Function_512(x2, x3, x0, x1, 6); \
   Round_Function_512(x3, x0, x1, x2, 7); \
   Round_Function_512(x0, x1, x2, x3, 8); \
   Round_Function_512(x1, x2, x3, x0, 9); \
   Round_Function_512(x2, x3, x0, x1, 10); \
   Round_Function_512(x3, x0, x1, x2, 11); \
   Round_Function_512(x0, x1, x2, x3, 12); \
   Round_Function_512(x1, x2, x3, x0, 13); \
   Round_Function_512(x2, x3, x0, x1, 14); \
} while(0)

/* Inversed Round Function for the 512-bit permutation */
#define Inv_Round_Function_512(x0, x1, x2, x3, i) do { \
   x0 = _mm_aesdeclast_si128(x0, RC1(i)); \
   x2 = _mm_aesdeclast_si128(_mm_aesimc_si128(x2), RC0(i)); \
   x2 = _mm_aesdeclast_si128(x2, RC1(i)); \
   x1 = _mm_aesenc_si128(x0, x1); \
   x3 = _mm_aesenc_si128(x2, x3); \
} while (0)

/* Inversed 512-bit permutation */
#define Inv_perm512(x0, x1, x2, x3) do { \
   Inv_Round_Function_512(x2, x3, x0, x1, 14); \
   Inv_Round_Function_512(x1, x2, x3, x0, 13); \
   Inv_Round_Function_512(x0, x1, x2, x3, 12); \
   Inv_Round_Function_512(x3, x0, x1, x2, 11); \
   Inv_Round_Function_512(x2, x3, x0, x1, 10); \
   Inv_Round_Function_512(x1, x2, x3, x0, 9); \
   Inv_Round_Function_512(x0, x1, x2, x3, 8); \
   Inv_Round_Function_512(x3, x0, x1, x2, 7); \
   Inv_Round_Function_512(x2, x3, x0, x1, 6); \
   Inv_Round_Function_512(x1, x2, x3, x0, 5); \
   Inv_Round_Function_512(x0, x1, x2, x3, 4); \
   Inv_Round_Function_512(x3, x0, x1, x2, 3); \
   Inv_Round_Function_512(x2, x3, x0, x1, 2); \
   Inv_Round_Function_512(x1, x2, x3, x0, 1); \
   Inv_Round_Function_512(x0, x1, x2, x3, 0); \
} while(0)
```

A.2 Simpira-256 and Simpira-512

```
#include <stdint.h>
#include <immintrin.h>

/* Round Constant */
#define RC0(i) _mm_setr_epi32(0x00^(i)^(2), 0x10^(i)^(2), 0x20^(i)^(2), 0x30^(i)^(2))
#define RC1(i) _mm_setr_epi32(0x00^(2*(i)+1)^(4), 0x10^(2*(i)+1)^(4), 0x20^(2*(i)+1)^(4), 0
     x30^(2*(i)+1)^(4))
#define RC2(i) _mm_setr_epi32(0x00^(2*(i)+2)^(4), 0x10^(2*(i)+2)^(4), 0x20^(2*(i)+2)^(4), 0
     x30^(2*(i)+2)^(4))

/* Round Function for the 256-bit permutation */
#define Round_Function_256(x0, x1, i) do { \
   x1 = _mm_aesenc_si128(_mm_aesenc_si128(x0, RC0(i)), x1); \
} while(0)

/* 256-bit permutation */
#define perm256(x0, x1) do { \
   Round_Function_256(x0, x1, 0); \
   Round_Function_256(x1, x0, 1); \
   Round_Function_256(x0, x1, 2); \
   Round_Function_256(x1, x0, 3); \
   Round_Function_256(x0, x1, 4); \
   Round_Function_256(x1, x0, 5); \
   Round_Function_256(x0, x1, 6); \
   Round_Function_256(x1, x0, 7); \
   Round_Function_256(x0, x1, 8); \
   Round_Function_256(x1, x0, 9); \
   Round_Function_256(x0, x1, 10); \
   Round_Function_256(x1, x0, 11); \
   Round_Function_256(x0, x1, 12); \
   Round_Function_256(x1, x0, 13); \
   Round_Function_256(x0, x1, 14); \
} while(0)

/* Inversed 256-bit permutation */
#define Inv_perm256(x0, x1) do { \
   Round_Function_256(x0, x1, 14); \
   Round_Function_256(x1, x0, 13); \
   Round_Function_256(x0, x1, 12); \
   Round_Function_256(x1, x0, 11); \
   Round_Function_256(x0, x1, 10); \
   Round_Function_256(x1, x0, 9); \
   Round_Function_256(x0, x1, 8); \
   Round_Function_256(x1, x0, 7); \
   Round_Function_256(x0, x1, 6); \
   Round_Function_256(x1, x0, 5); \
   Round_Function_256(x0, x1, 4); \
   Round_Function_256(x1, x0, 3); \
   Round_Function_256(x0, x1, 2); \
   Round_Function_256(x1, x0, 1); \
   Round_Function_256(x0, x1, 0); \
} while(0)

/* Round Function for the 512-bit permutation */
#define Round_Function_512(x0, x1, x2, x3, i) do { \
   x1 = _mm_aesenc_si128(_mm_aesenc_si128(x0, RC1(i)), x1); \
   x3 = _mm_aesenc_si128(_mm_aesenc_si128(x2, RC2(i)), x3); \
} while (0)

/* 512-bit permutation */
#define perm512(x0, x1, x2, x3) do { \
   Round_Function_512(x0, x1, x2, x3, 0); \
   Round_Function_512(x1, x2, x3, x0, 1); \
   Round_Function_512(x2, x3, x0, x1, 2); \
   Round_Function_512(x3, x0, x1, x2, 3); \
   Round_Function_512(x0, x1, x2, x3, 4); \
   Round_Function_512(x1, x2, x3, x0, 5); \
   Round_Function_512(x2, x3, x0, x1, 6); \
   Round_Function_512(x3, x0, x1, x2, 7); \
   Round_Function_512(x0, x1, x2, x3, 8); \
   Round_Function_512(x1, x2, x3, x0, 9); \
   Round_Function_512(x2, x3, x0, x1, 10); \
   Round_Function_512(x3, x0, x1, x2, 11); \
   Round_Function_512(x0, x1, x2, x3, 12); \
   Round_Function_512(x1, x2, x3, x0, 13); \
   Round_Function_512(x2, x3, x0, x1, 14); \
} while(0)

/* Inversed 512-bit permutation */
#define Inv_perm512(x0, x1, x2, x3) do { \
   Round_Function_512(x2, x3, x0, x1, 14); \
   Round_Function_512(x1, x2, x3, x0, 13); \
   Round_Function_512(x0, x1, x2, x3, 12); \
   Round_Function_512(x3, x0, x1, x2, 11); \
   Round_Function_512(x2, x3, x0, x1, 10); \
   Round_Function_512(x1, x2, x3, x0, 9); \
   Round_Function_512(x0, x1, x2, x3, 8); \
   Round_Function_512(x3, x0, x1, x2, 7); \
   Round_Function_512(x2, x3, x0, x1, 6); \
   Round_Function_512(x1, x2, x3, x0, 5); \
   Round_Function_512(x0, x1, x2, x3, 4); \
   Round_Function_512(x3, x0, x1, x2, 3); \
   Round_Function_512(x2, x3, x0, x1, 2); \
   Round_Function_512(x1, x2, x3, x0, 1); \
   Round_Function_512(x0, x1, x2, x3, 0); \
} while(0)
```

A.3 NEON Implementations of Areion-256 and Areion-512

```
#include<stdint.h>
#include<arm_neon.h>

/* Round Constant aligned for little endian */
const uint32_t RC[][4] = {
    {0x03707344, 0x13198a2e, 0x85a308d3, 0x243f6a88},
    {0xec4e6c89, 0x082efa98, 0x299f31d0, 0xa4093822},
    {0x34e90c6c, 0xbe5466cf, 0x38d01377, 0x452821e6},
    {0xb5470917, 0x3f84d5b5, 0xc97c50dd, 0xc0ac29b7},
    {0x98dfb5ac, 0xd1310ba6, 0x8979fb1b, 0x9216d5d9},
    {0x6a267e96, 0xb8e1afed, 0xd01adfb7, 0x2ffd72db},
    {0xb3916cf7, 0x24a19947, 0xf12c7f99, 0xba7c9045},
    {0x1574e690, 0x36920d87, 0x58efc166, 0x801f2e28},
    {0x728eb658, 0x0d95748f, 0xf4933d7e, 0xa458fea3},
    {0xc25a59b5, 0x7b54a41d, 0x82154aee, 0x718bcd58},
    {0x286085f0, 0xc5d1b023, 0x2af26013, 0x9c30d539},
    {0x603a180e, 0x8e79dcb0, 0xb8db38ef, 0xca417918},
    {0xbd314b27, 0xd71577c1, 0xb01e8a3e, 0x6c9e0e8b},
    {0xaa55ab94, 0xe65525f3, 0x55605c60, 0x78af2fda},
    {0x2aab10b6, 0x55ca396a, 0x63e81440, 0x57489862},
    {0x7c72e993, 0xa15486af, 0x1141e8ce, 0xb4cc5c34},
    {0x741831f6, 0x2ba9c55d, 0x636fbc2a, 0xb3ee1411},
    {0x6c24cf5c, 0xafd6ba33, 0x9b87931e, 0xce5c3e16},
    {0x6b4bb9af, 0x3b8f4898, 0x28958677, 0x7a325381},
    {0xfb21a991, 0x61d809cc, 0x66282193, 0xc4bfe81b},
    {0xe98575b1, 0xef845d5d, 0x5dec8032, 0x487cac60},
    {0xd396acc5, 0x23893e81, 0xeb651b88, 0xdc262302},
    {0x48420040, 0xe0b4482a, 0x3f442392, 0xf6d6ff38},
    {0xf6e96c9a, 0x21c66842, 0x9e1f9b5e, 0x69c8f04a}
};
```

```
#define RC0 vmovq_n_u8(0)
#define RC1(i) vreinterpretq_u8_u32(vld1q_u32(RC[i]))

/* Operations for the round function */
#define A1(X, K) vaesmcq_u8((vaeseq_u8(X, K)))
#define A2(X, K) vaeseq_u8(X, K)
#define A3(X) vaesmcq_u8(X)
#define A4(X, K) vaesdq_u8(X, K)
#define XOR(X, Y) veorq_u8(X, Y)

/* Round Function for the 256-bit permutation */
#define R_FIRST(x0, x1, i) \
    do { \
        x1 = A2(A1(A1(x0, RC0), RC1(i)), x1); \
        x0 = A2(x0, RC0); \
    } while (0)

#define R_MIDDLE(x0, x1, i) \
    do { \
        x1 = A2(A1(A1(x0, RC0), RC1(i)), x1); \
    } while (0)

#define R_LAST(x0, x1, i) \
    do { \
        x1 = XOR(A1(A1(x0, RC0), RC1(i)), x1); \
        x0 = A2(x0, RC0); \
    } while (0)

/* 256-bit permutation */
#define perm256(x0, x1) \
    do { \
        R_FIRST(x0, x1, 0); \
        R_MIDDLE(x1, x0, 1); \
        R_MIDDLE(x0, x1, 2); \
        R_MIDDLE(x1, x0, 3); \
        R_MIDDLE(x0, x1, 4); \
        R_MIDDLE(x1, x0, 5); \
        R_MIDDLE(x0, x1, 6); \
        R_MIDDLE(x1, x0, 7); \
        R_MIDDLE(x0, x1, 8); \
        R_LAST(x1, x0, 9); \
    } while (0)


 /* Inversed Round Function for the 256-bit permutation */
#define Inv_R_FIRST(x0, x1, i) \
    do { \
        x0 = A4(x0, RC0); \
        x1 = A4(A1(A1(x0, RC0), RC1(i)), x1); \
    } while (0)

#define Inv_R_MIDDLE(x0, x1, i) \
    do { \
        x1 = A4(A1(A1(x0, RC0), RC1(i)), x1); \
    } while (0)

#define Inv_R_LAST(x0, x1, i) \
    do { \
        x1 = XOR(A1(A1(x0, RC0), RC1(i)), x1); \
    } while (0)

/* Inversed 256-bit permutation */
#define Inv_perm256(x0, x1) \
    do { \
        Inv_R_FIRST(x1, x0, 9); \
        Inv_R_MIDDLE(x0, x1, 8); \
        Inv_R_MIDDLE(x1, x0, 7); \
        Inv_R_MIDDLE(x0, x1, 6); \
        Inv_R_MIDDLE(x1, x0, 5); \
        Inv_R_MIDDLE(x0, x1, 4); \
        Inv_R_MIDDLE(x1, x0, 3); \
        Inv_R_MIDDLE(x0, x1, 2); \
        Inv_R_MIDDLE(x1, x0, 1); \
        Inv_R_LAST(x0, x1, 0); \
    } while (0)

 /* Round Function for the 512-bit permutation */
#define R_FIRST(x0, x1, x2, x3, i) \
    do { \
        x1 = A2(A1(x0, RC0), x1); \
        x3 = A2(A1(x2, RC0), x3); \
        x0 = A2(x0, RC0); \
        x2 = A1(A2(x2, RC0), RC1(i)); \
    } while (0)

#define R_MIDDLE(x0, x1, x2, x3, i) \
    do { \
        x1 = A2(A1((x0), x1); \
        x3 = A2(A3(x2), x3); \
        x2 = A1(x2, RC1(i)); \
    } while (0)

#define R_LAST(x0, x1, x2, x3, i) \
    do { \
        x1 = XOR(A3(x0), x1); \
        x3 = XOR(A3(x2), x3); \
        x2 = A1(x2, RC1(i)); \
    } while (0)

/* 512-bit permutation */
#define perm512(x0, x1, x2, x3) \
    do { \
        R_FIRST(x0, x1, x2, x3, 0); \
        R_MIDDLE(x3, x0, x1, x2, 1); \
        R_MIDDLE(x2, x3, x0, x1, 2); \
        R_MIDDLE(x1, x2, x3, x0, 3); \
        R_MIDDLE(x0, x1, x2, x3, 4); \
        R_MIDDLE(x3, x0, x1, x2, 5); \
        R_MIDDLE(x2, x3, x0, x1, 6); \
        R_MIDDLE(x1, x2, x3, x0, 7); \
        R_MIDDLE(x0, x1, x2, x3, 8); \
        R_MIDDLE(x3, x0, x1, x2, 9); \
        R_MIDDLE(x2, x3, x0, x1, 10); \
        R_MIDDLE(x1, x2, x3, x0, 11); \
        R_MIDDLE(x0, x1, x2, x3, 12); \
        R_MIDDLE(x3, x0, x1, x2, 13); \
        R_LAST(x2, x3, x0, x1, 14); \
    } while (0)
```

B Test Vectors

B.1 Areion-256

```
=== test vector #1 ===
Input =
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Output =
e5 a7 66 63 82 50 14 24 68 dc 9d 76 65 dd 36 9f
8f 79 99 8b 7a a0 92 90 6f e5 1b fd eb fa c9 c1

=== test vector #2 ===
Input =
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f

Output =
73 53 ec 51 d4 9f ad 89 ee cb 5b ef 1e a0 e4 76
ed 6c dc dd af 34 62 0d 01 3d cc f2 a2 26 f4 57
```

B.2 Areion-512

```
=== test vector #1 ===
Input =
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Output =
5f ee f7 7c bb e8 4c 79 58 08 94 59 f4 54 e9 6f
bf 21 fa b8 35 65 cc af 91 6b cf 9c fb 63 d2 5b
a0 26 42 fc c1 75 12 36 40 d6 a2 18 3b a6 82 b2
0b 72 3a fc 66 68 ff f3 de c4 7c 17 61 27 b9 84

=== test vector #2 ===
Input =
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f

Output =
a6 09 5f e0 57 d2 83 80 ba d2 5c 28 12 b2 30 f6
6f 07 b0 09 a3 04 98 5a f4 37 bb 60 8a 4c b8 31
39 2a 6f 2f 48 e4 25 ef 24 11 96 21 67 2e 37 c4
f1 9b 94 e0 e4 ea ed af b9 f4 eb 12 6a 6d 8a bb
```

B.3 Areion256-DM

```
=== test vector #1 ===
Input =
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Output =
e5 a7 66 63 82 50 14 24 68 dc 9d 76 65 dd 36 9f
8f 79 99 8b 7a a0 92 90 6f e5 1b fd eb fa c9 c1

=== test vector #2 ===
Input =
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
```

```
Output =
73 52 ee 52 d0 9a ab 8e e6 c2 51 e4 12 ad ea 79
fd 7d ce ce bb 21 74 1a 19 24 d6 e9 be 3b ea 48
```

B.4 Areion512-DM

```
=== test vector #1 ===
Input =
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Output =
58 08 94 59 f4 54 e9 6f 91 6b cf 9c fb 63 d2 5b
a0 26 42 fc c1 75 12 36 0b 72 3a fc 66 68 ff f3

=== test vector #2 ===
Input =
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f

Output =
b2 db 56 23 1e bf 3e f9 ec 2e a1 7b 96 51 a6 2e
19 0b 4d 0c 6c c1 03 c8 c1 aa a6 d3 d0 df db 98
```

B.5 Areion512-MD

```
=== test vector #1 ===
Input =
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Output =
47 dd 7f 2c 11 f3 05 e6 97 40 95 e3 c8 61 2f 6e
8d 09 bb ea 63 ef be 8d 84 55 8f cb f5 28 81 37

=== test vector #2 ===
Input =
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f

Output =
61 17 b5 9f 30 25 cd 4e 66 8b dc b3 66 bd 89 b9
06 0e 8d cf 67 0c bf 43 08 a8 96 86 8e bc c6 fc
```