

Article

Chaining Optimization Methodology: A New SHA-3 Implementation on Low-End Microcontrollers

Young Beom Kim ¹, Taek-Young Youn ^{2,*} and Seog Chung Seo ^{1,*}

¹ Department of Financial Information Security, Kookmin University, Seoul 02707, Korea; darania@kookmin.ac.kr

² Department of Industrial Security, Dankook University, Giheung-gu, Yongin-si (16891) 655, Korea

* Correspondence: taekyoung@dankook.ac.kr (T.-Y.Y.); scseo@kookmin.ac.kr (S.C.S.); Tel.: +82-02-910-4742 (T.-Y.Y. & S.C.S.)

Abstract: Since the Keccak algorithm was selected by the US National Institute of Standards and Technology (NIST) as the standard SHA-3 hash algorithm for replacing the currently used SHA-2 algorithm in 2015, various optimization methods have been studied in parallel and hardware environments. However, in a software environment, the SHA-3 algorithm is much slower than the existing SHA-2 family; therefore, the use of the SHA-3 algorithm is low in a limited environment using embedded devices such as a Wireless Sensor Networks (WSN) environment. In this article, we propose a software optimization method that can be used generally to break through the speed limit of SHA-3. We combine the θ , π , and ρ processes into one, reducing memory access to the internal state more efficiently than conventional software methods. In addition, we present a new SHA-3 implementation for the proposed method in the most constrained environment, the 8-bit AVR microcontroller. This new implementation method, which we call the chaining optimization methodology, implicitly performs the π process of the f -function while minimizing memory access to the internal state of SHA-3. Through this, it achieves up to 26.1% performance improvement compared to the previous implementation in an AVR microcontroller and reduces the performance gap with the SHA-2 family to the maximum. Finally, we apply our SHA-3 implementation in Hash_Deterministic Random Bit Generator (Hash_DRBG), one of the upper algorithms of a hash function, to prove the applicability of our chaining optimization methodology on 8-bit AVR MCUs.

Keywords: SHA-3; Keccak algorithm; 8-bit AVR MCUs; embedded; microcontroller; WSN



Citation: Kim, Y.B.; Youn, T.-Y.; Seo, S.C. Chaining Optimization Methodology: A New SHA-3 Implementation on Low-End Microcontrollers. *Sustainability* **2021**, *13*, 4324. <https://doi.org/10.3390/su13084324>

Academic Editor: Il-sun You

Received: 28 February 2021

Accepted: 30 March 2021

Published: 13 April 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Due to the increase in the use of IoT devices, many applications have been designed to use the devices so that many algorithms are implemented on low-powered devices [1]. However, due to the weaknesses of communication in Wireless Sensor Networks (WSNs), the security of communicating messages is easily damaged. Since the cryptographic hash function can guarantee the integrity of a communicating message, we can use it for the security of the communication in WSNs. Moreover, the function can be used as a core technique in security applications such as PBKDF2, HMAC, DRBG, and digital signature algorithms (including DSA, ECDSA, and RSA-PSS). Recently, some weaknesses of standard hash functions (e.g., SHA-1 or SHA2) have been discovered. Therefore, the US National Institute of Standards and Technology (NIST) recommends not using SHA-1 [2–6]. Generally applicable attacks concerning the SHA-2 family were introduced before 2008. Two functions SHA-1 and SHA-2 have different structures, but they share the same algorithm (i.e., SHA) [7,8]. Therefore, they share the same vulnerability. In some scenarios, the security of SHA-2 is the same as that of SHA-1, except SHA-2 uses larger inputs and outputs [7,8]. A number of preimage attacks on SHA-2 have been found [9–11]. Fortunately, before the weakness of SHA-2 against preimage attacks was found, NIST held the SHA-3 competition. The Keccak algorithm was selected as SHA-3, the new standard hash function [12].

Until now, researchers had examined the security of SHA-3 (Keccak). While SHA-3 is proven to be secure against known attacks, its usage is limited due to its inefficiency [13–20]. It is known that SHA-3 is three times slower than SHA-256 in the 8-bit AVR platform [12,15–17] and two times slower than SHA-512 in a CPU environment [2,18–20]. For sustainable IT services, security is the most important requirement, and we need usable tools that can be applied to IT services with sufficient performance. Since the hash function is one of fundamental tools for the security of IoT-based IT services, research on the software optimization for SHA-3 is essential for designing sustainable IoT-based IT services.

In this paper, we present a new optimization method for fast SHA-3 implantation. We examine the efficiency of the proposed technique for 8-bit AVR microcontrollers. Note that they are mainly used for sensor devices in WSNs. The proposed techniques come from the fact that 25 lanes of the internal state can be operated independently. Differently from existing implementations, we combine the θ and ρ processes to reduce memory accesses. Based on this fact, π process can be executed implicitly. The efficiency gain can be obtained without spending additional operations or lookup tables. This feature is very useful for WSN applications that use limited embedded devices. It is used in both PKE/KEM and digital signature group algorithms and has been applied in various ways such as polynomial generation and signature verification. Therefore, in the future, an optimization method for SHA-3 is absolutely necessary.

1.1. The Contribution of This Paper is as Follows:

- **Proposing an efficient reduced memory access method for Fast SHA-3 implementation**

In this article, we analyze the SHA-3 algorithm in detail from an implementation point of view. We propose a new method to reduce memory access to the internal state of SHA-3 by calculating the number of memory accesses and analyzing the characteristics of each process while the f -function is operating. Our technique of combining the three processes performs the f -function efficiently without breaking the security of SHA-3. Moreover, our method is a generic method that can be applied to low and high-end processors such as 8-bit AVR, 32-bit ARM, CPU, and GPGPU.

- **Presenting chaining optimization methodology on 8-bit AVR MCUs**

We present a new SHA-3 implementation methodology called the chaining optimization methodology. Through the chaining optimization methodology, the θ , π , and ρ processes of f -function are combined into one, and, by using this, the memory access to the internal state of SHA-3 is reduced to the maximum. Based on the chaining optimization methodology, our SHA-3 software implements the π process implicitly using registers efficiently. This shows a performance improvement of up to 26.1% over the previous best implementation. In addition, our software achieves the fastest performance in an 8-bit AVR microcontroller as far as we know.

- **Presenting optimized Hash_DRBG on 8-bit AVR environment**

We prove the applicability of our SHA-3 software by using Hash_DRBG in an 8-bit AVR MCUs. In addition, we propose an optimized implementation of Hash_DRBG to reduce the performance gap with the existing SHA-2 implementation in the AVR environment. three operations are omitted from the first loop of the f -function through the input data used repeatedly in the derivation function of Hash_DRBG. The proposed lookup table is only 160 bytes and has the advantage that it can be created during SHA-3 operation. By using this, the proposed Hash_DRBG implementation achieves the fastest performance among Hash_DRBG using SHA-3.

1.2. Hash Function for Service Sustainability in Embedded Systems

As the development of 5G communication is accelerating and 6G communication is being developed, the communication-based Information Technology (IT) industry is in charge of the development of fundamental technologies in a wide range of fields such as economic growth, social integration, and environmental preservation. In order to design sustainable development based on close exchanges in each field, the IT engineering-based industry must be solid. Security technology is the basis of the IT industry, and so far it has supported the security of the IT industry based on encryption technology in various network communications. When communicating with each other, integrity in the verification of data is essential in terms of data storage and transmission. The hash function is an algorithm that verifies the integrity, and it is used in all cryptographic protocols. As the Internet of Things (IoT) environment develops, the embedded-based communication industry is becoming more active. Therefore, the hash function must be used for service-sustainability in embedded systems. For integrity verification, so far mainly SHA-1 and SHA-2 have been used. However, as existing SHA-1-based vulnerabilities have been discovered, National Institute of Standards and Technology (NIST) has recommended using SHA-3. Up to now, the migration of SHA-3 in crypto-engineering and industry is inadequate due to the limitations of the software performance. The SHA-3 software optimization method proposed in this article can be applied for service sustainability in various fields in the future. In addition, with the development of the quantum computing environment, NIST has held a competition for standardization of Post Quantum Cryptography (PQC). Since most of the algorithms currently submitted in Round 3 use the SHA-3 algorithm, our research can be future-oriented and contribute to the crypto-based communication industry in the future [21–24].

1.3. Extended Version of ICISC'20

In this paper, we extended our previous work published in ICISC'20 [25]. In ICISC'20, it was difficult to describe the proposed method and implementation methodology in detail, due to page limitations. However, in this article, we have modified our software and re-established the implementation methodology through detailed explanations. Compared to ICISC'20, the algorithm of the implementation methodology has been completely modified, and there is a performance improvement over ICISC'20 based on the modified code. We also added Hash_DRBG to prove that our implementation methodology can be applied to Hash_DRBG.

The remainder of this paper is organized as follows. In Section 2, we explain the background of SHA-3 and Hash_DRBG. Section 3 contains an analysis of related work concerning SHA-3's effects on the AVR environment. Section 4 proposes a new optimization method for SHA-3 and Hash_DRBG. Section 6 evaluates our software. Finally, Section 7 concludes the article.

2. Background

2.1. Overview of SHA-3

In 1993, NIST published Secure Hash Algorithm 0 (SHA-0). Subsequently, two standard hash functions, SHA-1 and SHA-2, were proposed. However, Stevens et al. broke the security of SHA-1 by finding a collision [2]. Therefore, NIST has proposed a new hash function standard and redefined the Keccak algorithm of Bertoni et al. as SHA-3, a new standard function [12]. Differently from SHA-2, the Keccak algorithm has a new type of structure, the so-called sponge structure. As a result, the Keccak algorithm is not vulnerable to known attacks that are applied to SHA-2 [12].

2.1.1. Sponge Structure

In Figure 1, we describe the structure of SHA-3. The process of SHA-3 is a sponge structure composed of two processes, the absorbing process and the squeezing process. The first process compresses the input value using the permutation function f . For each

iteration, an exclusive-OR (XOR) operation is applied to some part of the output of f and the padding value. The digest is computed in the squeezing process. In SHA-3, b -bit permutation means the size of the state and is fixed with $b \in \{25, 50, 100, 200, 400, 800, 1600\}$. Here, b is composed with the bitrate r and the capacity c , and satisfies $b = r + c$. If the digest length is longer than r , then f is applied to change the internal state. In this paper, we use the following parameters: $b = 1600$, $r = 1088$, and $c = 512$, and the digest length is 256 bits. Note that the values are widely used [12].

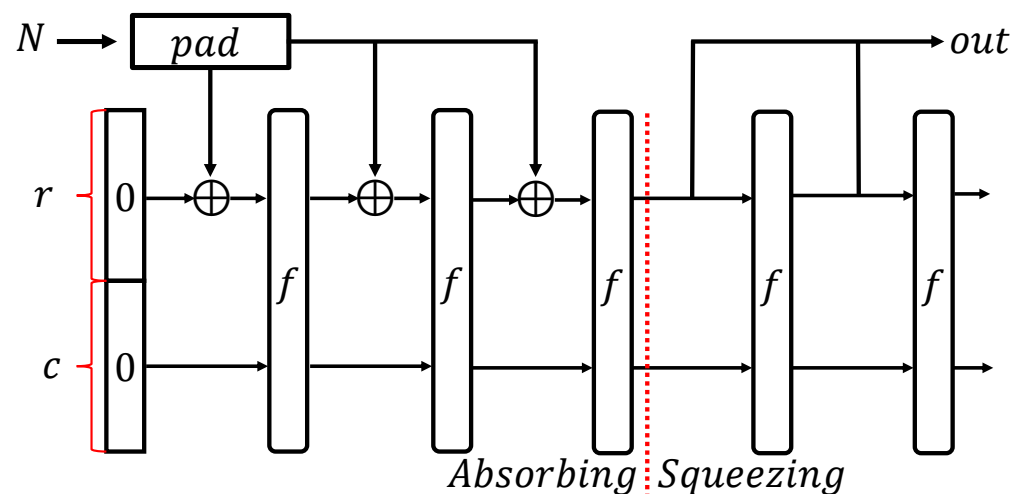


Figure 1. Overview of Sponge Structure [12].

2.1.2. State of SHA-3

The state is used as the input of the function f . Recall that f is the core component of SHA-3. Therefore, it is important to make out the structure of the state to understand SHA-3. In Figure 2, the structure of the state is described.

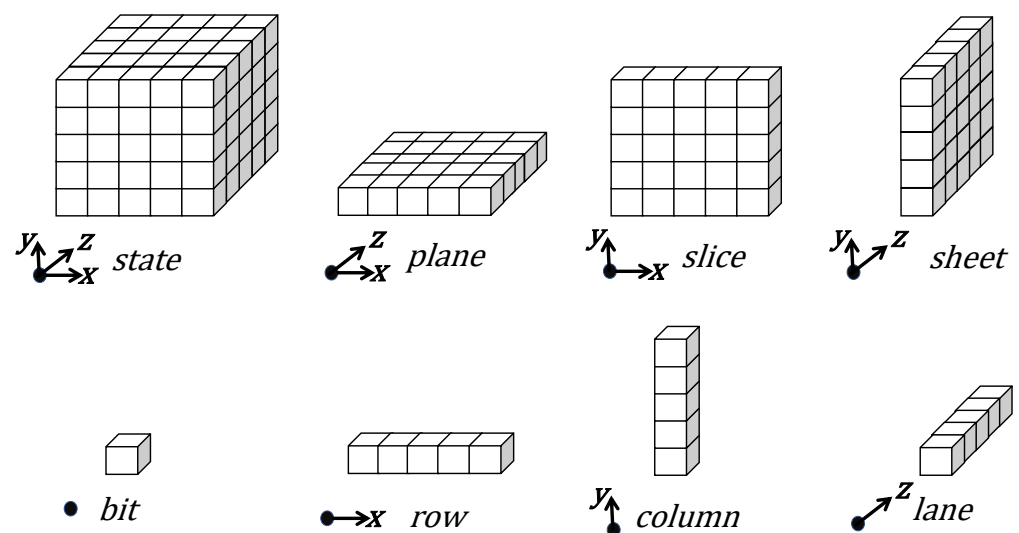


Figure 2. State of SHA-3 [12].

The state is a three-dimensional matrix such that $x \times y \times z$ and $|x| + |y| = 5$. The state consists of 25 lanes. The length of each lane is determined by b , w , and l . According to b , the state is composed of $5 \times 5 \times w$ -bit. Some parameter settings are shown in Table 1. As seen in the table, $l = \log_2(b/25)$ and $w = 2^l$. In SHA-3, the function f is repeated n_r times where $n_r = 12 + 2 \times l$.

Table 1. The values of w and l for each b [12].

b (bit)	25	50	100	200	400	800	1600
w (bit)	1	2	4	8	16	32	64
l	0	1	2	3	4	5	6

2.1.3. f -Function

The f -function is a b -bit permutation which consists of five processes: θ , π , ρ , χ and ι . Based on the processes, the state is updated by repeating the f -function n_r times.

In the θ process, each bit in the state is XORed to two columns in the array. Let us consider the case where the θ process is performed for the bit (x_0, y_0, z_0) . In this case, two columns, of which the x -coordinates are $(x_0 + 1) \bmod 5$ and $(x_0 - 1) \bmod 5$, are used. In this process, $(z_0 - 1) \bmod w$ is assigned to the z -coordinate of the column of which the x -coordinate is $(x_0 - 1) \bmod 5$. We describe the procedure in Algorithm 1. In Line 2, $D[x, z]$ is computed. Note that the procedure is called as the *initial* θ .

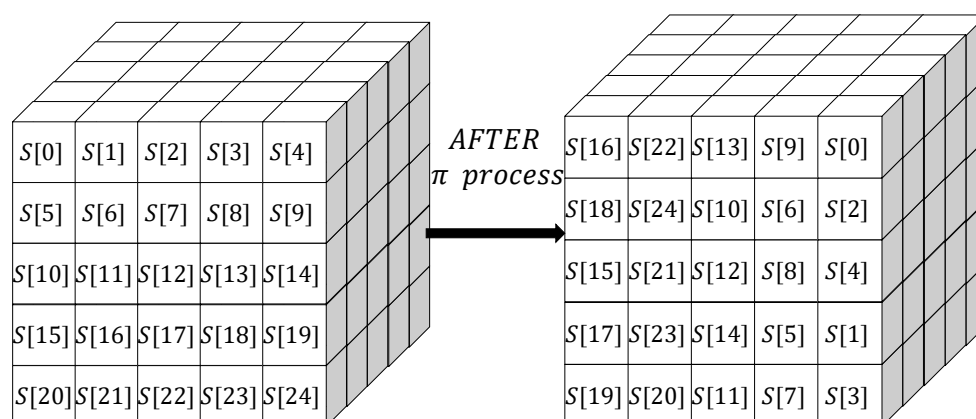
Algorithm 1 θ Process [12]

Require: state A

Ensure: state A'

- 1: For all pairs (x, z) such that $0 \leq x < 5$ and $0 \leq z < w$
 $C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z];$
- 2: For all pairs (x, z) such that $0 \leq x < 5$ and $0 \leq z < w$
 // This step is initial θ
 $D[x, z] = C[(x - 1) \bmod 5, z] \oplus C[(x + 1) \bmod 5, (z - 1) \bmod w];$
- 3: For all triples (x, y, z) such that $0 \leq x, y < 5$ and $0 \leq z < w$
 $A'[x, y, z] = A[x, y, z] \oplus D[x, z];$
- 4: **return** A'

In Figure 3, the π process is described. In the process, the values of lanes are rearranged. Here, $S[i] (i \in [0, 24])$ is the i -th lane of the state. Note that $S[12]$ is a lane of $(x = 0, y = 0)$ in state [12].

**Figure 3.** Overview of the π process [12].

In the ρ process, each lane is right-rotated. The size of rotation is called the offset. The offset is determined by the x and y coordinates as seen in Table 2 [12]. The z -coordinate is modified by adding the offset where the lane size is used as the modulo.

Table 2. Offsets of the ρ Process [12].

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

The effect of the χ process is to XOR each bit with a nonlinear function of two other bits in its row [12]. Note that the difference between the χ process and the other processes (i.e., the θ , π , and ρ processes) is that the χ process should be operated in row form and implemented accordingly.

The ι process executes an XOR operation for the lane of $(x = 0, y = 0)$ of the state and constants RC [12]. Since the ι process operates only one lane, in most implementations, the χ and ι processes are combined into a single process.

2.2. Overview of Hash_DRBG

The security of the crypto used in the cryptographic protocol is proved on the premise of the use of a perfect random number generator. However, since it is practically impossible to implement an ideal random number generator in an embedded device such as 8-bit AVR microcontrollers and a 32-bit ARM Cortex series, a pseudo-random number is generated by using a Deterministic Random Bit Generator (DRBG). Among DRBGs, there are two types of DRBGs using a hash function (Hash_DRBG, and HMAC_DRBG). In the case of HMAC_DRBG, Hash-based Message Authentication Code (HMAC), the application algorithm of the hash function is used as the core algorithm; therefore, it is basically slower than Hash_DRBG and uses more memory [26]. Therefore, for the cryptographic protocol, it is recommended to use Hash_DRBG in the constrained 8-bit AVR environment [27].

Figure 4 shows the overall overview of Hash_DRBG, and Table 3 shows the specification of parameters used in Hash_DRBG. Hash_DRBG basically extracts a random bit while updating operational status consisting of V and C. The length of V and C is the seedlen for each hash function. The initial setting of operational status is completed by using the derivation function in the instantiate function. Afterward, random bit is extracted using operational status in the generate function. After extracting random bit, the value of V is updated through the hash function. Figure 5 shows the structure of the derivation function. The derivation function is a function to extract V and C for the initial instance setting. It updates V and C by operating the hash function as much as len_seed for input data. For instance, in the case of Hash_DRBG using SHA3-512, len_seed is 3, so the hash function is operated three times for input data. The reseed function is a function to update operational status when the generate function is called multiple times and has the same structure as the derived function. The extraction function is the process of extracting the actual random bits. It receives V as an input and extracts as many random bits as the user wants using the hash function. At this time, when the hash function is used multiple times, the value of V increases by 1. Finally, when random bit extraction is finished, operational status is updated.

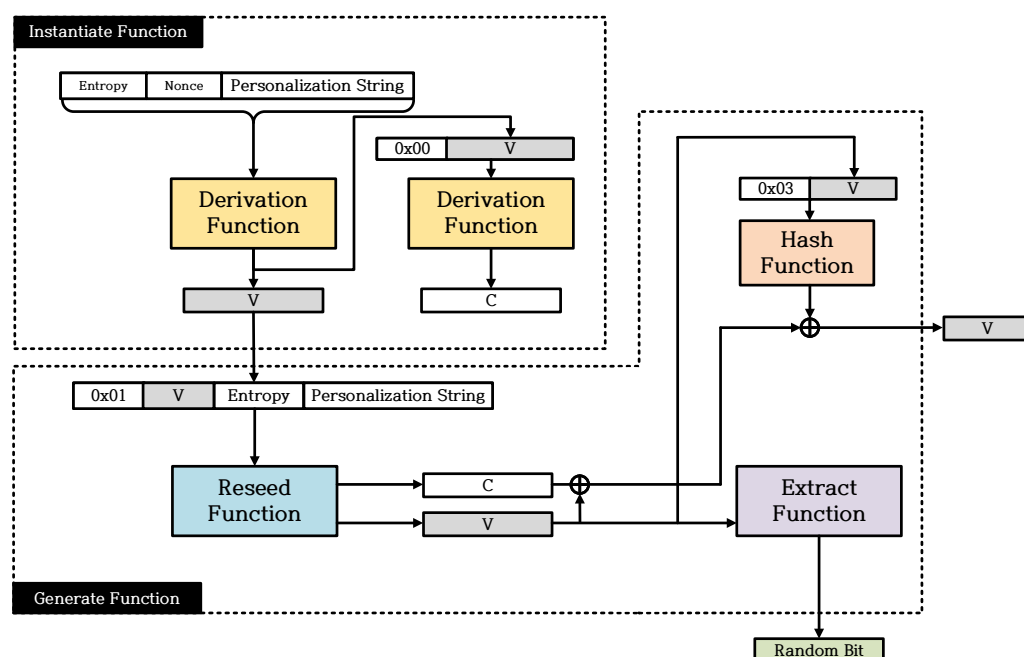


Figure 4. Overview of Hash_DRBG [26].

Table 3. Notations of Hash_DRBG [26].

	SHA3-256	SHA3-384	SHA3-512
highest supported security strength	2^{256}	2^{384}	2^{512}
digest (out length of hash function)	256-bit	384-bit	512-bit
len_seed	2	3	3
seedlen	440-bit	888-bit	

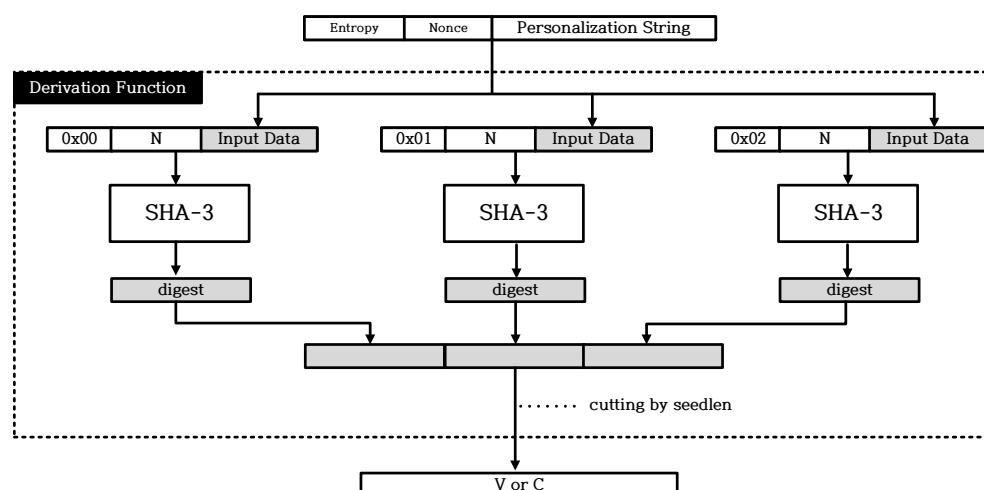


Figure 5. Structure of Derivation Function [26].

2.3. Overview of 8-Bit AVR MCUs

The 8-bit AVR microcontroller is an embedded device made into a single integrated circuit by adding memory and I/P to the microprocessor. Recently, the microcontroller has been widely used in many applications for WSNs [1]. The 8-bit AVR microcontroller's commands consists of operation codes and an operand. In Table 4, we summarize some commands and related information for the commands that are used in this paper. We

focus on ATmega128 since it is widely used in for sensor nodes [28]. The target device has the following resources: 128 KB of flash memory, 4 KB SRAM, and 4 KB EEPROM. The device supports throughput of 16 MIPS at 16 MHz and operates between 4.5 and 5.5 volts [29]. The AVR-MCU has 32 8-bit general-purpose registers, which are used for various purposes, e.g., basic private operations and bit operations. Specifically, the R26-R31 registers can be combined and used as three 16-bit registers, i.e., X, Y, and Z registers. These registers (X, Y, and Z) are used as pointers to indirectly specify a 16-bit address for data memory. The status register (SREG) shows the status and result after Arithmetic Logic Unit (ALU) calculations.

Table 4. The 8-bit AVR Assembly Instructions [30,31]; cc means clock cycles.

Asm	Operands	Description	Operation	cc
ADD	Rd, Rr	Add without Carry	$Rd \leftarrow Rd + Rr$	1
ADC	Rd, Rr	Add with Carry	$Rd \leftarrow Rd + Rr + C$	1
MOV	Rd, Rr	Copy Register	$Rd \leftarrow Rr$	1
EOR	Rd, Rr	Exclusive OR	$Rd \leftarrow Rd \oplus Rr$	1
LSL	Rd	Logical Shift Left	$C Rd \leftarrow Rd \ll 1$	1
LSR	Rd	Logical Shift Right	$Rd C \leftarrow 1 \gg Rd$	1
ROL	Rd	Rotate Left Through Carry	$C Rd \leftarrow Rd \ll 1 C$	1
ROR	Rd	Rotate Right Through Carry	$Rd C \leftarrow C 1 \gg Rd$	1
BST	Rd, b	Bit store from Bit in Reg to T Flag	$T \leftarrow Rd(b)$	1
BLD	Rd, b	Bit load from T Flag to a Bit in Reg	$Rd(b) \leftarrow T$	1
LDI	Rd, K	Load Immediate	$Rd \leftarrow K$	1
LD	Rd, X	Load Indirect	$Rd \leftarrow (X)$	2
ST	Z, Rr	Store Indirect	$(Z) \leftarrow Rr$	2

3. Analysis of Previous Hash Softwares on 8-Bit AVR Microcontrollers

3.1. Related Works of SHA-3 Implementation on 8-Bit AVR Environment

Keccak algorithm has been widely implemented in various environments including embedded processors since SHA-3 standard selection in 2012. It is widely known that the hardware implementation of SHA-3 has the advantage of faster execution than that of SHA-1 and SHA-2 [32]. However, with respect to software implementation, SHA-3 is much slower than existing hash algorithm including SHA-2 [13–17]. Currently, existing SHA-3 software on a variety of IoT devices, including 8-bit AVR MCUs, are implemented according to the pseudo-codes of the NIST SHA-3 standard [13–16]. The existing implementations following the pseudo-codes of the standard compute π and ρ in the combined way as $\pi \sim \rho$ rather than computing them separately, because rotate-operation (ρ) can be embedded in the process of π computation. Note that software implementations following the standard execute f -function in the following order: θ process $\rightarrow \pi \sim \rho$ process $\rightarrow \chi \sim \iota$ process.

There exists two representative SHA-3 implementations: Otte et al.'s one and Balasch et al.'s one [15,16] on 8-bit AVR MCUs. In the case of Otte et al.'s, the SHA-3 implementation is contained in the AVR-Crypto-Lib and its codes are written in pure C-language [15]. Otte et al.'s SHA-3 implementation (the version resulting in 256-bit digest) in AVR-Crypto-Lib consumes 2,570,828 clock cycles when hashing a message of 500 bytes, which means a hash rate of 5142 (CPB).

This execution time of SHA-3 is almost seven times that of SHA-2 software implemented by Otte et al. [15]. Thus, we focused on analyzing Balasch et al.'s SHA-3 implementation written with a combination of C and assembly codes because it is the fastest SHA-3 implementation on 8-bit AVR MCUs [16]. Balasch et al.'s SHA-3 software (the version resulting in 256-bit digest) takes 716,483 clock cycles when computing a digest of a message composed of 500 bytes, which denotes a hash rate of 1432 (CPB).

Balasch et al. introduced a new shift-rotation strategy for a faster $\pi \sim \rho$ process. Note that we need shift-rotations by 1 bit to the left (ROL) and to the right (ROR). In SHA-3, a single lane is 64 bits in length when $b = 1600$ and $w = 64$. In the ρ process, we need eight registers to rotate a 64-bit. Let LSL be a 1-bit logical left-shift, ADC be an addition with carry, BST be a 1-bit store to T in SREG, and BLD be a 1-bit load from T in SREG. Generally, in an 8-bit AVR MCU, a 1-bit left-rotation is implemented using an LSL followed by seven ROLs and an ADC [33]. Besides, a 1-bit right-rotation is implemented using a BST followed by eight RORs and a BLD. For $1 < n < 8$, n bits rotation for 64-bit data, the procedure can be performed by repeating 1 bit rotation of 64-bit n times. However, the cost of the dedicated operation for n -bit rotation is not equal to the cost of n -bit rotation that can be implemented by repeating 1-bit rotation. Therefore, we need a dedicated method for n -bit rotation or $(64-n)$ -bit rotation for efficient implementation [33]. Note that, for $n > 8$, the execution time of n -bit shift-rotations can be reduced to 40 clock cycles or fewer if $x = (x \ggg n)$ operations are replaced by $x = (x \ggg n\%8)$. In this process, operations of $x = (x \ggg n/8)$ directly allocate and store in memory. When storing to memory (the ρ process), the implementation of Balasch et al. combines π and ρ processes into a single process ($\pi \sim \rho$). Note that Balasch et al. implemented SHA-3 based on the order of standard implementation [12].

While Balasch et al.'s technique gives better performance for the 8-bit AVR MCUs, SHA-2 is still much faster than SHA-3. This is caused by the fact that the cost of accessing memory is more expensive than arithmetic operations in low-end processors. Moreover, the state in SHA-3 (for $b = 1600$) requires at least 200 bytes. Note that the memory requirement is very heavy compared with ordinary symmetric ciphers, where only 128 bits are required for the state. Therefore, it is important to minimize the amount of memory access to the state.

In Table 5, we summarize all memory accesses to state when SHA-3 is implemented as recommended by NIST. Here, $\pi \sim \rho$ means that π and ρ are combined, and $\theta \sim \rho$ means that θ and ρ are combined. χ and ι processes are also combined with the same logic. The goal of *initial* θ is to create $D[x, z]$ in Algorithm 1. Note that, *initial* θ is a part of the θ process. In Table 5, we can see that the state is accessed 3, 2, and 2 times during the θ , $\pi \sim \rho$, and $\chi \sim \iota$ processes, respectively. Hence, well-known SHA-3 implementation needs seven memory accesses. For $b = 1600$, $w = 64$, and $l = 16$, the length of the state is 200 bytes ($= 25 \times 64/8$). In addition, all processes are repeated 24 times ($12 + 6 \times 2$) in the f -function. Hence, each execution of the f -function requires 168 ($= 24 \times 7$) memory accesses.

Table 5. Number of times memory accesses the state of previous implementation, e.g., Balasch et al. and Otte et al. [13–17].

Standard Method	Initial θ	θ Process	$\pi \sim \rho$ Process	$\chi \sim \iota$ Process	Total Access
Load	O	O	O	O	7
Store	X	O	O	O	

In low-powered embedded devices, frequent memory accesses can cause low performance. Hence, in the following section, we introduce a new strategy to reduce the number of memory accesses without increasing additional computation or lookup tables.

3.2. Related Works of DRBG Implementation on 8-Bit AVR Environment

Research on Hash_DRBG and HMAC_DRBG, which is based on hash functions in 8-bit AVR environments, has not been studied as far as we know. However, research on block-cipher based CounTeR_DRBG (CTR_DRBG) has been actively conducted recently [27,34]. Based on the fact that the nonce, which is the input data of CTR-mode, is used repeatedly when a session is generated, a method was proposed to optimize by generating a lookup-table of common parts of the initial few rounds of the block-cipher. In addition, the Instantiate Function of DRBG is efficiently compressed by using the characteristic of CTR_DRBG that the initial operational status is zero [27].

4. Proposed Technique for Efficient SHA-3 Implementations in 8-Bit AVR MCUs

In this section, we propose an efficient implementation technique for optimized SHA-3 execution on 8-bit AVR MCUs. As mentioned in Section 3, memory accesses to state takes longer clock cycles compared with the arithmetic operations on 8-bit AVR MCUs. Thus, it is important to efficiently arrange the use of available general-purpose registers for the optimized state accesses and memory accesses during the computation of SHA-3.

4.1. Main Idea

The main idea is to make the π process implicitly executed at a minimum cost and integrate the θ and ρ processes into a single process ($\theta \sim \rho$ process) so as to reduce the number of memory accesses to the state. Figure 6 depicts the overview of the proposed SHA-3 implementation technique. From the figure, it is noticeable that, since each lane of the state is processed individually in the θ process, the ρ process can be applied to each lane. Namely, after computing $D[x, z]$ through the initial θ , we can compute the remaining θ and ρ processes (note that the remaining θ after the initial θ is computed with ρ as $\theta \sim \rho$). When computing the $\theta \sim \rho$ process, two memory accesses (load and store) are required to update the state. The π process can be executed implicitly while executing the $\theta \sim \rho$ process because its operation is just changing the position of the lane in the state without updating any values. Putting the above explanation together, the main idea of our implementation implicitly computes the π process when storing the updated state in the memory.

Table 6 compares the number of memory accesses to the state between the proposed method and the previous implementation following the standard method when computing the f -function. The proposed method reduces the number of memory accesses from 7 to 5, which is a saving of almost 28.57%. Since the state is 200-byte, the f function requires 120 (24×5) times memory accesses; our proposed technique can save 48 times the memory accesses in total compared with the previous implementation methods ($168 = 24 \times 7$).

Implementations of SHA-3 on the 8-bit AVR MCUs depend on the value of b . The value b of the SHA-3 configuration determines the type of implementation on 8-bit AVR MCUs. For example, if b is less than 200-bit (namely, w is less than seven), the registers on the MCU can hold all lanes of the state, unless, in the case of 400-bit, 800-bit, or 1600-bit (the size of a single lane is 2-byte, 4-bytes, and 8-byte, respectively), it is hard to hold whole lanes of the state on the 8-bit AVR's general-purpose registers. Therefore, how much memory access to the state is optimized with the available registers determines the overall performance of SHA-3 implementation when the lane is larger than 8-bit. Our implementation focusses on optimizing the performance of SHA-3 implementation when $b = 1600$ because it is the most common configuration value (in the Korean Cryptographic Module Validation Program (KCMVP) [35], 1600-bit is used for b).

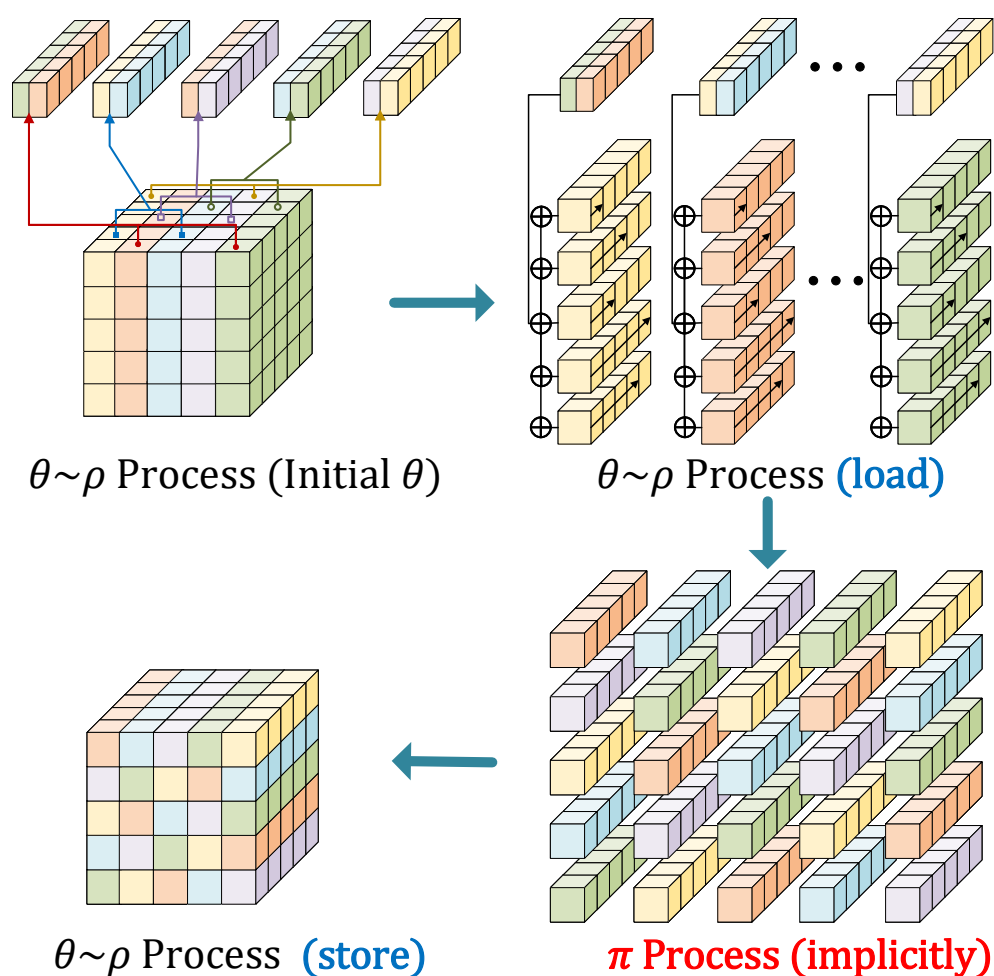


Figure 6. Proposed main idea.

Table 6. Number of memory accesses to the state (o/x are the signs for indicating there is a memory access or not, respectively).

Standard Method	Initial θ	θ Process	$\pi \sim \rho$ Process	$\chi \sim \iota$ Process	Total Access
Load	O	O	O	O	7
Store	X	O	O	O	
Proposed Method	Initial θ	$\theta \sim \rho$ Process	π Process	$\chi \sim \iota$ Process	Total Access
Load	O	O	X(Implied)	O	5
Store	X	O	X(Implied)	O	

From a more crypto-engineering point of view, the advantages of our method are out of the typical trade-off relationship. Until now, research has been conducted to optimize and implement various ciphers in various environments. They were accelerated by generating a lookup-table for skipping specific rounds and repetitive parts. However, since this increases the usage of Flash Memory or SRAM, it does not escape from the trade-off relationship; therefore, it is necessary to consider whether it can be used from the perspective of the actual crypto-industry. Our SHA-3 optimization technique is an implementation that increases performance without using additional computational tables. This fundamental method of reducing memory access does not affect the security of SHA-3 from a theoretical safety point of view, as it does not change the input/output of the hash function and implements

the SHA-3 mechanism of the standard document. In addition, differently, the existing and standard implementations do not use tables to run π and ρ processes. In other words, our implementation no longer uses the look-up table that the existing implementations inevitably used and additional tables, while raising the performance to the top. In terms of code size, there is a difference of about 1 KB, but since it occupies only about 4% of our target platform, ATmega128, it hardly affects the actual overall performance.

4.2. Proposed Implementation Technique on 8-Bit AVR MCUs

Figure 7 depicts the proposed register scheduling strategy for our SHA-3 implementation on 8-bit AVR MCUs. From the figure, eight registers are used to compute a single lane (64-bit) of the state. Namely, registers R8–R15 are used to execute operations on the single lane. Note that registers R16–R23 are used for temp register and they can also store a single lane like registers R8–R15. Registers R2–R5 are used to manage the address value of state $\theta \sim \rho$ process. Registers R26–R31 are address registers. For example, registers R26:R27 hold the state's address value in the f function. Registers R28:R29 maintain the address value of the $D[x, z]$ stored initial θ . Finally, the address value of the constant data used in the ι process is maintained in registers R30:R31.

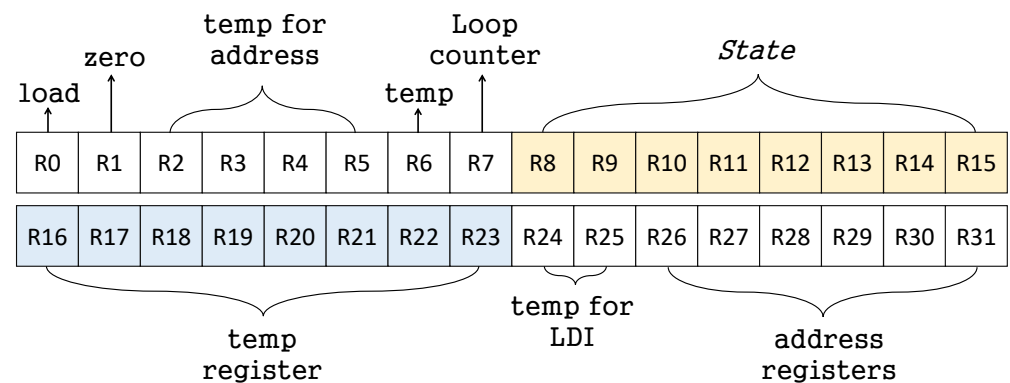


Figure 7. Proposed register scheduling strategy for our SHA-3 implementation on 8-bit AVR MCUs.

4.3. Proposed Assembly Code on 8-Bit AVR MCUs

In this section, we present algorithms and codes which implement our proposed method on an 8-bit AVR environment. There are some things to consider in order to integrate the $\theta \sim \rho$ process into one for the implicit execution of the π process in the AVR environment. As mentioned in Section 4.1, in order to minimize memory accesses, once a lane is loaded into registers, the $\theta \sim \rho$ process needs to be completed before being stored into memory. In other words, we apply the $\theta \sim \rho$ process to 25 lanes and execute the π process with cheap operations rather than an actual operation when stored in memory. For example, one lane is loaded into the registers to execute the $\theta \sim \rho$ process, and the result is stored at the memory location where the π process is applied. However, if the lane to which the $\theta \sim \rho(\pi)$ process has been executed is immediately stored in memory, the $\theta \sim \rho$ process cannot be applied to the lane at the original location. Thus, the corresponding lane needs to be loaded into registers before storing. For efficiency, our implementation arranges the order of the lanes to apply the $\theta \sim \rho$ process to the position index of the π process. That is, when storing the lane to which the $\theta \sim \rho$ process is executed along the index of the π process, the next lane is first loaded into the registers. Therefore, it is possible to implicitly perform the π process while performing the theta process for the lanes in turn. We call this implementation technique “chaining optimization methodology”. This new SHA-3 implementation can show that our proposed optimization method can be effectively applied in 8-bit AVR microcontrollers. The chaining optimization methodology efficiently alternates between two lanes and executes the π process without an additional lookup-table and operations while the previous π process implementation made use of the table containing position information and additional operations.

Algorithm 2 shows the execution of the $\theta \sim \rho$ process for the first five lanes of the index order of the π process among 25 lanes. Note that the π process is implicitly executed. We make use of some macro functions in order to efficiently integrate the θ and ρ processes. Algorithm 3 is a macro function LD_state in Algorithm 2 and it executes the θ process. Namely, LD_state loads a lane into R8–R15 and does XOR the loaded values with $D[i, z]$, ($i \in [0, 4]$, $z \in [0, 7]$) generated in the initial θ process. The lane and $D[i, z]$ are selected according to the offset, such as lines 11 and 12 in Algorithm 2. Line 1 in Algorithm 3 adds offset value to the lower 8-bit part of the X register, maintaining the address of the internal state. With this approach, the index of the operation sequence of the π process can be properly selected. Note that the memory address of f -function's 200-byte internal state can be aligned through compiler's directive (`_attribute_(aligned(256))`); carry does not occur when adding offset value to the 8-bit lower part of the X register. Through lines 2–9 in Algorithm 3, load one lane to R8–R15 registers and return the address value of the X register through lines 10 and 11. Similar to line 1, line 12 adds offset value to the X register to determine the i value of $D[i, z]$ and then lines 13–27 do XOR $D[i, 0]$, $D[i, 1]$, $\dots$, $D[i, 7]$ with the lane loaded in R8–R15. Like lines 10 and 11, line 29 restores the address value of $D[i, z]$ stored in the Y register. When the LD_state terminates, the θ process is completed for one lane maintained in R8–R15. LD_temp is a macro function that performs the θ process using R16–R23 registers, and executes the same as LD_state.

Algorithm 2 AVR Assembly Codes for proposed combined $\theta \sim \rho$ process for implicitly executing π process with initial five lane. $D[i]$ ($i \in [0, 4]$) : $D[i, z]$ of initial θ , ($z \in [0, 7]$). $S[j]$: 64-bit data of one lane of state. $S'[j]$: 64-bit data with θ process calculated, $\bar{S}[j]$: 64-bit data with θ and ρ process calculated, $j \in [0, 24]$.

load S[0]	$S[17] \leftarrow \bar{S}[14]$ computation
1: EOR R24	17: LDI R24, 136 // S[17]
2: EOR R25	18: LDI R25, 16 // D[2]
3: LD_state // R8–R15 : $S'[0] \leftarrow (S[0] \oplus D[0])$	19: LD_temp // R16–R23 : $S'[17] \leftarrow (S[17] \oplus D[2])$
4: ROR_3bit_state // $S'[0] \leftarrow S'[0] \ggg 3$	20: LDI R24, 136 // S[17]
	21: ST_ROR2bytes_state // $S[17] \leftarrow \bar{S}[14]$
$S[4] \leftarrow \bar{S}[0]$ computation	
5: LDI R24, 32 // S[4]	$S[15] \leftarrow \bar{S}[17]$ computation
6: LDI R25, 32 // D[4]	22: LDI R24, 120 // S[15]
7: LD_temp // R16–R23 : $S'[4] \leftarrow (S[4] \oplus D[4])$	23: LD_state // R8–R15 : $S'[15] \leftarrow (S[15] \oplus D[0])$
8: ROL_2bit_temp // $S'[4] \leftarrow S'[4] \lll 2$	24: ROR_4bit_state // $S'[15] \leftarrow S'[15] \ggg 4$
9: LDI R24, 32 // S[4]	25: LDI R24, 120 // S[15]
10: ST_ROR5bytes_state // $S[4] \leftarrow \bar{S}[0]$	26: ST_ROR7bytes_temp // $S[15] \leftarrow \bar{S}[17]$
$S[14] \leftarrow \bar{S}[4]$ computation	$S[10] \leftarrow \bar{S}[15]$ computation
11: LDI R24, 112 // S[14]	27: LDI R24, 80 // S[10]
12: LDI R25, 32 // D[4]	28: LD_temp // R16–R23 : $S'[10] \leftarrow (S[10] \oplus D[0])$
13: LD_state // R8–R15 : $S'[14] \leftarrow (S[14] \oplus D[14])$	29: ROR_4bit_temp // $S'[10] \leftarrow S'[10] \ggg 4$
14: ROR_2bit_state // $S'[4] \leftarrow S'[4] \ggg 2$	30: LDI R24, 80 // S[10]
15: LDI R24, 112 // S[14]	31: ST_ROR3bytes_state // $S[10] \leftarrow \bar{S}[15]$
16: ST_ROR0bytes_temp // $S[14] \leftarrow \bar{S}[4]$	

Algorithm 3 AVR Assembly macro codes of LD_state

```

        .macro LD_state
1:  ADD R26, R24
2:  LD  R8, X+
3:  LD  R9, X+
4:  LD  R10, X+
5:  LD  R11, X+
6:  LD  R12, X+
7:  LD  R13, X+
8:  LD  R14, X+
9:  LD  R15, X+
10: ADD R24, R7
11: SUB R26, R24
12: ADD R28, R25
13: LDD R0, Y + 0
14: EOR R8, R0
15: LDD R0, Y + 1
16: EOR R9, R0
17: LDD R0, Y + 2
18: EOR R10, R0
19: LDD R0, Y + 3
20: EOR R11, R0
21: LDD R0, Y + 4
22: EOR R12, R0
23: LDD R0, Y + 5
24: EOR R13, R0
25: LDD R0, Y + 6
26: EOR R14, R0
27: LDD R0, Y + 7
28: EOR R15, R0
29: SUB R28, R25
        .endm

```

Table 7 is the macro codes that performs the ρ process for one lane stored in registers. The ρ process performs a right rotate-shift operation according to the offset in Table 2. When $b = 1600$, the size of a lane is 64-bit. Thus, the offset of rotation-shift is less than 64. Since AVR's rotation-instruction only supports a single register, a combination of instructions is required to conduct the 1-bit rotation operation for one 64-bit lane maintained in eight registers. Rotating 64-bit data to 1-bit right (resp. left) incurs 10 (resp. 9) clock cycles. Therefore, when rotating right with n -bit ($0 < n \leq 3$) offset, it is effective to use ROR_1bit_state n times, and m -bit ($4 \leq m < 8$) offset, it is effective to use ROL_1bit_state m times. In 8-bit AVR MCUs, the 8-bit rotate-shift operation can be conducted with no cost through directly allocating the position of the register rather than actual shift arithmetics. Therefore, when updating the value of the lane to the memory, by changing the register order, rotate-shift operations for offsets of a multiple of eight can be efficiently computed. The macro codes storing the result in the memory are ST_ROR k bytes_state, ($k \in [0, 7]$), and like Algorithm 3, codes changing the address value are added.

Table 7. Macro code for chaining optimization methodology on 8-bit AVR MCUs [16,27].

fROR_1bit_state	ROL_1bit_state	ST_ROR1bytes_state	ST_ROR3bytes_state
BST R15, 0		ADD R26, R24	ADD R26, R24
ROR R8	LSL R15	ST X+, R15	ST X+, R13
ROR R9	ROL R14	ST X+, R8	ST X+, R14
ROR R10	ROL R13	ST X+, R9	ST X+, R15
ROR R11	ROL R12	ST X+, R10	ST X+, R8
ROR R12	ROL R11	ST X+, R11	ST X+, R9
ROR R13	ROL R10	ST X+, R12	ST X+, R10
ROR R14	ROL R9	ST X+, R13	ST X+, R11
ROR R15	ROL R8	ST X+, R14	ST X+, R12
BLD R8, 7	ADC R15, R1	ADD R24, R7	ADD R24, R7
		SUB R26, R24	SUB R26, R24
10 cycles	9 cycles	19 cycles	19 cycles

Algorithm 2 is assembly codes that implement chaining optimization methodology in the AVR environment by efficiently using the previously presented macro functions and codes. The X registers (R24:R25) are used as the address offset for the state and $D[i, z]$, the desired data are accessed during load and store operations. The θ process is applied to the lane while loading one lane from memory to the register, and the ρ process is applied when stored. At this time, the next lane is loaded into the register in the π process before storing. As shown in Figure 7, each lane is alternatively stored in either R8–R15 or R16–R23. For example, line 3 of Algorithm 2 executes the θ process while loading the 64-bit data of $S[0]$ to R8 – R15. At this time, since X and Y registers store the start addresses of the state (start address of $S[0]$) and $D[i, z]$, the offset for $S[0]$ is omitted. In line 4, the ρ process is performed on $S[0]$ where the θ process is completed. In the π process, the $S[0]$ lane moves to the $S[4]$ position, 171-bit of rotate-right operation is required (refer to Table 2). Actually, due to $171 = (64 \times 2) + 43$ in $b = 1600$, 43-bit of rotate-right is required for one lane. A 43-bit rotate-right can be computed with actual 3-bit rotate-right because 40-bit right-rotate can be implicitly conducted by directly adjusting the storage location. This is why the right rotate-shift operation is performed as much as 3-bit in line 4. The remaining 40-bit shift operation is implicitly executed when storing. Since the lane of $S[0]$ moves to $S[4]$ by the π process, line 7 loads the lane of $S[4]$ into R16–R23 before storing $S[0]$ in the $S[4]$ position. Since the offset is determined through lines 5 and 6, $S[4]$ can be accurately loaded into the register. $S[4]$ needs to be moved to the position of $S[14]$ through the π process, which requires 62-bit of rotate-left ($(192 = 64 \times 2 + 62)$), according to Table 2. This can be conducted with the 2-bit rotate-left operation via line 8. Since the $S[4]$ lane is kept in the registers, $S[0]$ is moved to the $S[4]$ position through line 10. At this time, the rotate operations of 8-bit multiple can be conducted implicitly. The θ , π , and ρ processes are completed for the internal state of SHA-3 when the operation for each lane of Algorithm 2 is repeated 25 times.

The proposed chaining optimization methodology contributes to the much reduced number of memory accesses to the internal state. Since the memory access cost for 8-bit data is at least two clock cycles in the AVR microcontroller, a reduction in memory access of two times per round for 200 bytes of the internal state of SHA-3 has a speedy performance improvement. Thus, compared to previous implementations, we can save 50 memory accesses to the internal state when calling an f -function of SHA-3. our chaining optimization methodology is an optimization method that can always be expected to improve performance, regardless of the SHA-3 parameter and is not restricted by the platform.

5. Proposed Technique for Efficient Hash_DRBG Implementations Using SHA-3

In this section, we propose an efficient implementation technique for optimized Hash_DRBG execution on 8-bit AVR MCUs. First, we analyzed the various functions that makeup Hash_DRBG. Initially, V and C must be updated with the instantiate function; thus, the optimized method using the fixed initial operational status of CTR_DRBG is difficult to apply in Hash_DRBG [27]. In addition, in the case of the generates function in Hash_DRBG, V and C values are initially impossible to infer; therefore, we analyzed the SHA-3 optimization factor in the derivation function of Hash_DRBG. Since the input data of the instantiate function are the same as the input data of the derivation function, duplicate parts exist between the input data of the hash functions in the derivation function. Therefore, we chose a strategy to infer the state of each SHA-3 for the same part.

As mentioned in Section 2.2, when generating an initial instance, the derivation function operates SHA-3 as much as len_seed according to the security level of SHA-3. Input data of the derivation function consists of entropy, nonce, and personalization string. Entropy and nonce require at least half of the security-bit of SHA-3, and personalization string cannot exceed 35-bit at most [26]. In the case of $b = 1600$, which our target parameters, the input data for the SHA-3 hash function of the derivation function is less than 136 bytes, corresponding to the r of the sponge structure. Therefore, the initial state of the hash function operates repeatedly as long as len_seed in the derivation function has only 1 byte

difference for each SHA-3. While the first SHA-3 of the derivation function is operating, a lookup-table can be generated for the duplicated part; thus, part of the one round operation of the f -function for the next SHA-3 can be omitted. Figure 8 shows the optimization strategy of Hash_DRBG, applying chaining optimization methodology. When executing the second SHA-3 of the derivation function, the spread of 1 byte in the first round of the f -function occurs independently only in the lane before the $\chi \sim \iota$ process. Therefore, after applying chaining optimization methodology to one round of the f -function in 8-bit AVR MCUs, 20 lanes have duplicated values with the previous SHA-3 state. By using this, if the first SHA-3 of the derivation function generates a lookup-table for 20 lanes of the state while performing one round of the f -function, 20 lanes of the second SHA-3 in the derivation function can omit the $\theta \sim \rho$ process in one round.

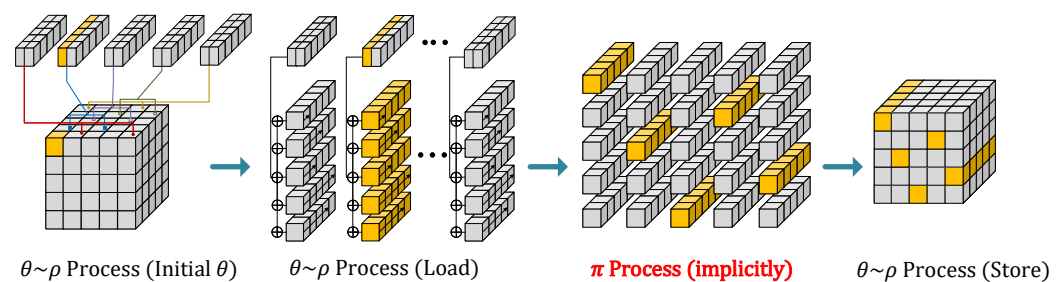


Figure 8. Proposed f -function of Hash_DRBG implementation.

6. Performance Analysis

6.1. Performance Analysis of SHA-3 Implementation

In this section, we compare the proposed implementation of the chaining optimization methodology with the existing SHA-3 implementation. We use SHA-3 with general parameters of $b = 1600$, $r = 1088$, and $c = 512$ applicable to actual fields; therefore, the internal state of SHA-3 has 200 bytes, and up to 136 bytes per f -function can be hashed by using the characteristics of the sponge structure. In our software, the padding function of the sponge structure uses the same method as the standard implementation. As the target environment, we chose ATmega 128, one of the most used pieces of equipment in the WSN environment. For accurate implementation and comparison, ATmega 128 was simulated in Atmel studio7 and the $-O2$ option was applied when compiling code. The performance of each hash function is measured in clock cycles per byte (CPB) in order that the relationship between hashing bytes and performance can be fairly expressed.

Table 8 compares our new SHA-3 implementation in an 8-bit AVR MCU with several hash functions implemented in an AVR environment. Our SHA-3 software applying chaining optimization methodology implicitly implements the π process and minimizes memory access to the internal state; thus, high performance can be expected compared to existing SHA-3 implementation. Our software achieved 2646, 1326, and 1066 CPB when hashing 50, 100, and 500 bytes, respectively. The proposed implementation not only shows up to 26.1% performance improvement over the existing best performance, Balasch et al.'s SHA-3 implementation in the AVR environment, but also achieves the highest performance regardless of the hash rate compared to existing SHA-3 software. In addition, compared to the previous SHA-3 implementation, the difference in performance with the SHA-2 family has been reduced almost two times in the 8-bit AVR microcontroller [15,16].

Table 8. Performance of proposed SHA-3 Implementations by hash rate when hashing a byte of various messages in an 8-bit AVR microcontroller, hash rates represent cyc/byte (CPB) [15,16].

Reference	Algorithm	Language	Length of Message Byte		
			50 Byte	100 Byte	500 Byte
This Paper	SHA-3(256-bit)	Asm	2646 (+25.6%)	1326 (+26.1%)	1066 (+25.5%)
Otte et al. [15]	SHA-3 (256-bit)	C	12854	6427	5142
Balasch et al. [16]	SHA-3 (256-bit)	Asm	3560 (–)	1795 (–)	1432 (–)
Balasch et al. [16]	SHA-256	Asm	672	668	532
Balasch et al. [16]	Blake (256-bit)	Asm	714	708	562
Balasch et al. [16]	Grøstl (256-bit)	Asm	1220	1012	686
Balasch et al. [16]	Photon (256-bit)	Asm	9723	7982	4788

6.2. Performance Analysis of Hash_DRBG Based on SHA-3 Implementation

Table 9 shows the performance of Hash_DRBG based on clock cycles by extracted random bytes. Since there are no Hash_DRBG software results for Balasch et al. and Otte et al. [15,16], the overall part of Hash_DRBG, except the core hash function, is applied to the software implemented directly to compare the performance improvement. Therefore, the rest of the functions of Hash_DRBG, except the core hash function, are configured identically; therefore, the performance improvement of chaining optimization methodology can be measured fairly. Since the nonce requires at least half the size of the security-bit of the hash function and the personalization string cannot exceed 35-bit, we set the input data of the derivation function to 64 bytes, which is an appropriate size in the AVR environment [26]. Our Hash_DRBG software, chaining optimization methodology and optimized technique for derivation function, shows a performance improvement of 26.1%–26.5% compared to the Hash_DRBG with Balasch et al. and Otte et al., which are the existing SHA-3 implementations in 8-bit AVR MCUs. The reason the performance improvement of Hash_DRBG slightly decreases as the number of extracted random bytes increases is that the f -function's first-round optimization factor of the proposed derivation function is fixed, but the hash function calls increase as the extracted bytes increase. Therefore, our Hash_DRBG software achieves the best performance when generating random numbers less than 100 bytes. In addition, our software reduces the gap with Hash_DRBG, using the existing SHA-2 family up to four times or less in the AVR environment.

Table 9. Performance of proposed Hash_DRBG Implementations by extracting random byte in the 8-bit AVR microcontroller; performance measured by clock cycles [15,16].

Reference	DRBG	Algorithm	Language	Extracted Random Byte		
				50 Byte	100 Byte	500 Byte
This Paper	Hash	SHA-3(256-bit)	Asm	917,600 (+26.5%)	1,182,200 (+26.3%)	1,579,100 (+26.1%)
Balasch et al. [16]	Hash	SHA-3 (256-bit)	Asm	1,248,100 (–)	1,604,100 (–)	2,138,100 (–)
Balasch et al. [16]	Hash	SHA-256	Asm	247,100	317,100	422,100
Otte et al. [15]	Hash	SHA-3 (256-bit)	C	4,501,000	5,786,400	7,714,500

7. Concluding Remarks

In this article, we presented a new SHA-3 implementation, which we call chaining optimization methodology, in the 8-bit AVR microcontroller. We proposed a new optimization method for SHA-3, which had a speed limitation in the software environment compared to the SHA-2 family. The memory access to the internal state of SHA-3 is reduced as much as possible by combining processes that can be calculated independently for each lane of the internal state among the f -function. Through efficient register scheduling and chaining operation for the internal state, the performance load of the π process is reduced to a minimum in the 8-bit AVR microcontroller. Our chaining optimization methodology, which is not dependent on a specific platform, such as a parallel environment, essentially reduces memory access to the state. Finally, our SHA-3 optimization method can be applied in a variety of crypto-fields. Unlike technologies that can only be applied to specific situations or devices, our methodology is applicable to all platforms and application algorithms without compromising the security of SHA-3. This means our software can completely replace the previous SHA-3 implementation on an 8-bit AVR environment. From a memory access point of view, we have proven that we have proposed a method applicable to all platforms. It also replaced the application algorithms used in the actual crypto industry with SHA-3-based DRBG. By using SHA-3 in Hash_DRBG, the most widely used algorithm has proven the effectiveness of the proposed technique. Therefore, the new SHA-3 implementation can be widely used and is particularly effective in limited environments such as embedded devices. In addition, in NIST Post-Quantum Cryptography Competition (PQC), as most of the candidates for the competition use the SHA-3 algorithm, we believe that our proposed method can be applied to PQC.

Author Contributions: Writing—original draft, Y.B.K.; Writing—review and editing, T.-Y.Y. and S.C.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research of YoungBeom Kim and Seog Chung Seo was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1F1A1058494). This research of Taek-Young Youn was supported by Korea Institute for Advancement of Technology(KIAT) grant funded by the Korea Government(MOTIE) (P0008703, The Competency Development Program for Industry Specialist).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Park, S.E.; Hwang, C.G.; Park, D.C. Internet of Things(IoT) ON system implementation with minimal Arduino based appliances standby power using a smartphone alarm in the environment. *JKIECS* **2015**, *10*, 1175–1182.
2. Stevens, M.; Bursztein, E.; Karpman, P.; Albertini, A.; Markov, Y. The First Collision for Full SHA-1. In *Advances in Cryptology—CRYPTO 2017, Proceedings of the 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, 20–24 August 2017*; Springer: Berlin, Germany, 2017; Volume 10401, pp. 570–596.
3. Wang, X.; Yin, Y.L.; Yu, H. Finding Collisions in the Full SHA-1. In *Advances in Cryptology—CRYPTO 2005, Proceedings of the 25th Annual International Cryptology Conference, Santa Barbara, CA, USA, 14–18 August 2005*; Springer: Berlin, Germany, 2005; Volume 3621, pp. 17–36.
4. Rijmen, V.; Oswald, E. Update on SHA-1. *IACR Cryptol. ePrint Arch.* **2005**, *2005*, 10.
5. Cannière, C.D.; Rechberger, C. Finding SHA-1 Characteristics: General Results and Applications. In *Advances in Cryptology—ASIACRYPT 2006, Proceedings of the 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, 3–7 December 2006*; Springer: Berlin, Germany, 2006; Volume 4284, pp. 1–20.
6. Manuel, S. Classification and generation of disturbance vectors for collision attacks against SHA-1. *Des. Codes Cryptogr.* **2011**, *59*, 247–263.
7. Khovratovich, D.; Rechberger, C.; Savelieva, A. Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family. In *Fast Software Encryption, Proceedings of the 19th International Workshop, FSE 2012, Washington, DC, USA, 19–21 March 2012*; Canteaut, A., Ed.; Springer: Berlin, Germany, 2012; Volume 7549, pp. 244–263.

8. Lamberger, M.; Mendel, F. Higher-Order Differential Attack on Reduced SHA-256. *IACR Cryptol. ePrint Arch.* **2011**, 2011, 37.
9. Mendel, F.; Nad, T.; Schl  ffer, M. Improving Local Collisions: New Attacks on Reduced SHA-256. In *Advances in Cryptology—EUROCRYPT 2013, Proceedings of the 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, 26–30 May 2013*; Springer: Berlin, Germany, 2013; Volume 7881, pp. 262–278.
10. Dobraunig, C.; Eichlseder, M.; Mendel, F. Analysis of SHA-512/224 and SHA-512/256. *IACR Cryptol. ePrint Arch.* **2016**, 2016, 374.
11. Sasaki, Y.; Wang, L.; Aoki, K. Preimage Attacks on 41-Step SHA-256 and 46-Step SHA-512. *IACR Cryptol. ePrint Arch.* **2009**, 2009, 479.
12. Morris, J.D. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. 2015. Available online: <https://doi.org/10.6028/NIST.FIPS.202> (accessed on 2 February 2021).
13. Lee, H.; Hong, D.; Kim, H.; Seo, C.; Park, K. An Implementation of an SHA-3 Hash Function Validation Program and Hash Algorithm on 16bit-UICC. *J. Korea Inst. Inf. Sec. Cryptol.* **2014**, 41, 885–891.
14. Kang, M.; Lee, H.w.; Hong, D.; Seo, C. Implementation of SHA-3 Algorithm Based On ARM-11 Processors. *J. Korea Inst. Inf. Sec. Cryptol.* **2015**, 25, 749–757.
15. AVR-Crypto-Lib. 2015. Available online: <https://wiki.das-labor.org/w/-AVR-Crypto-Lib/en> (accessed on 2 February 2021).
16. Balasch, J.; Ege, B.; Eisenbarth, T.; G  rard, B.; Gong, Z.; G  neysu, T.; Heyse, S.; Kerckhof, S.; Koeune, F.; Plos, T.; et al. Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices. In *Smart Card Research and Advanced Applications, Proceedings of the 11th International Conference, CARDIS 2012, Graz, Austria, 28–30 November 2012*; Springer: Berlin, Germany, 2012; Volume 7771, pp. 158–172.
17. Team, K. Extended Keccak Code Package. 2018. Available online: <https://keccak.team/index.html> (accessed on 2 February 2021).
18. KISA. SHA-3 Source Code Manual. 2020. Available online: <https://seed.kisa.or.kr//kisa/kcmvp/EgovVerification.do> (accessed on 2 February 2021).
19. Team, K. The eXtended Keccak Code Package (Open-Source Implementations of the Cryptographic Schemes Defined by the Keccak Team). Available online: <https://github.com/XKCP/XKCP> (accessed on 2 February 2021).
20. Korea Internet & Security Agency Open Cryptography Algorithms. Available online: <https://seed.kisa.or.kr/kisa/reference/EgovSource.do> (accessed on 2 February 2021).
21. Basso, A.; Mera, J.M.B.; D’Anvers, J.P.; Karmakar, A.; Roy, S.S.; Beirendonck, M.V.; Vercauteren, F. SABER: Mod-LWR Based KEM (Round 3 Submission). 2020. Available online: <https://www.esat.kuleuven.be/cosic/pqcrypto/saber/index.html> (accessed on 2 February 2021).
22. Avanzi, R.; Bos, J.; Ducas, L.; Kiltz, E.; Lepoint, T.; Lyubashevsky, V.; Schanck, J.M.; Schwabe, P.; Seiler, G.; Stehl  , D. CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation. 2020. Available online: <https://pq-crystals.org/> (accessed on 2 February 2021).
23. Bai, S.; Ducas, L.; Kiltz, E.; Lepoint, T.; Schwabe, P.; Seiler, G.; Stehle, D. CRYSTALS-Dilithium Algorithm Specifications And Supporting Documentation. 2020. Available online: <https://pq-crystals.org/> (accessed on 2 February 2021).
24. Fouque, P.A.; Hoffstein, J.; Kirchner, P.; Lyubashevsky, V.; Thomas, T.P.; Ricosset, P.T.; Seiler, G.; Whyte, W.; Zhang, Z. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. 2020. Available online: <https://falcon-sign.info> (accessed on 2 February 2021).
25. Kim, Y.; Choi, H.; Seo, S.C. Efficient Implementation of SHA-3 Hash Function on 8-bit AVR-based Sensor Nodes. In *The 23rd Annual International Conference on Information Security and Cryptology*; Springer: Berlin, Germany, 2020.
26. Barker, E.; Kelsey, J. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. 2015. Available online: <https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final> (accessed on 2 February 2021).
27. Kim, Y.; Kwon, H.; An, S.; Seo, H.; Seo, S.C. Efficient Implementation of ARX-Based Block Ciphers on 8-Bit AVR Microcontrollers. *Mathematics* **2020**, 8, 1837.
28. Kim, Y.; Seo, S.C. An Efficient Implementation of AES on 8-bit AVR-based Sensor Nodes. In *The 21th World Conference on Information Security Applications*; Springer: Berlin, Germany, 2020.
29. Liu, Z.; Seo, H.; Gro  sch  dl, J.; Kim, H. Efficient Implementation of NIST-Compliant Elliptic Curve Cryptography for 8-bit AVR-Based Sensor Nodes. *IEEE Trans. Inf. Forensics Secur.* **2016**, 11, 1385–1397.
30. Atmel. AVR Instruction Set Manual. 2012. Available online: <http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf> (accessed on 2 February 2021).
31. Kwon, H.; Kim, H.; Choi, S.J.; Jang, K.; Park, J.; Kim, H.; Seo, H. Compact Implementation of CHAM Block Cipher on Low-End Microcontrollers. In *The 21th World Conference on Information Security Applications*; Springer: Berlin, Germany, 2020.
32. Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations. Available online: <https://www.semanticscholar.org/paper/Fair-and-Comprehensive-Performance-Evaluation-of-14-Guo/0a1eeac2c74ef77127bbd926b87a13805eb61b6b> (accessed on 2 February 2021).
33. Cheng, H.; Dinu, D.; Gro  sch  dl, J. Efficient Implementation of the SHA-512 Hash Function for 8-Bit AVR Microcontrollers. In *Innovative Security Solutions for Information Technology and Communications, Proceedings of the 11th International Conference, SecITC 2018, Bucharest, Romania, 8–9 November 2018*; Springer: Berlin, Germany, 2018; Volume 11359, pp. 273–287.

-
34. KIM, Y.; SEO, S.C. Efficient Implementation of AES and CTR_DRBG on 8-bit AVR-based Sensor Nodes. *IEEE Access* **2021**, *1*, doi:10.1109/ACCESS.2021.3059623.
 35. KISA. KCMVP Manual for Cryptography. 2020. Available Online: <https://seed.kisa.or.kr/kisa/Board/79/detailView.do> (accessed on 2 February 2021).