cryptography

MDPI

Article

Hardware Performance Evaluation of Authenticated Encryption SAEAES with Threshold Implementation

Takeshi Sugawara

Department of Informatics, The University of Electro-Communications, Tokyo 182-8585, Japan; sugawara@uec.ac.jp

Received: 30 June 2020; Accepted: 5 August 2020; Published: 9 August 2020

Abstract: SAEAES is the authenticated encryption algorithm instantiated by combining the SAEB mode of operation with AES, and a candidate of the NIST's lightweight cryptography competition. Using AES gives the advantage of backward compatibility with the existing accelerators and coprocessors that the industry has invested in so far. Still, the newer lightweight block cipher (e.g., GIFT) outperforms AES in compact implementation, especially with the side-channel attack countermeasure such as threshold implementation. This paper aims to implement the first threshold implementation of SAEAES and evaluate the cost we are trading with the backward compatibility. We design a new circuit architecture using the column-oriented serialization based on the recent 3-share and uniform threshold implementation (TI) of the AES S-box based on the generalized changing of the guards. Our design uses 18,288 GE with AES's occupation reaching 97% of the total area. Meanwhile, the circuit area is roughly three times the conventional SAEB-GIFT implementation (6229 GE) because of a large memory size needed for the AES's non-linear key schedule and the extended states for satisfying uniformity in TI.

Keywords: threshold implementation; SAEAES; authenticated encryption, side-channel attack; changing of the guards; lightweight cryptography; implementation

1. Introduction

There is an increasing demand for secure data communication between embedded devices in many areas, including automotive, industrial, and smart-home applications. To enable cryptography in resource-constrained devices, researchers have studied lightweight cryptography that has a good performance in implementation by design. Lightweight cryptography emerged from block cipher design [1], which now covers a larger area in cryptography, including authenticated encryption (AE). In particular, NIST is running a standardization process for lightweight AE algorithms (NIST LWC) [2].

Side-channel attack (SCA) [3,4] is a considerable security risk in lightweight cryptography's main targets: embedded devices under a hostile environment in which a device owner attacks the device with physical possession. Consequently, NIST LWC considers the grey-box security model with side-channel leakage [5]. In addition to security, the cost of implementing SCA countermeasures in resource-constrained devices is a big issue because SCA countermeasures multiply the cost.

Threshold implementation (TI) [6] is an SCA countermeasure based on multi-party computation (MPC) [7]. TI is popular for hardware implementations because it can provide the security in the presence of glitches, i.e., transient signal propagation through a combinatorial circuit, which is inevitable in common hardware design. Consequently, there are an increasing number of papers reporting authenticated encryptions with TI [8–10]. Researchers are even optimizing the algorithms for TI: the TI-friendly S-boxes [11,12] and the TI-friendly modes of operation [13,14].

SAEAES is an instantiation of the SAEB mode of operation [15] with the standard block cipher AES [16], and is a NIST LWC candidate. Choosing AES is a practical decision for providing backward

compatibility with the numerous AES accelerators and coprocessors that the industry has invested so far. However, not so many NIST LWC candidates chose AES (COMET [17], mixFeed [18], and SAEAES [19] our of the 32 candidates) because newer lightweight primitives outperform AES in lightweight implementations. The impact of using AES is even larger with TI. Many lightweight algorithms, such as GIFT [20] and SKINNY [21], use an S-box with which an efficient, i.e., 3-share and uniform TI is available [21]. In contrast, this is not the case for AES [22], which was standardized before TI become popular. The early AES TI compensated for this disadvantage by refreshing the output share by adding fresh randomness [23–25], but this raised another implementation challenge of generating fresh randomness at a high rate. Daemen's *changing of the guards* [26] in 2017 opened the door for enabling a uniform TI for a larger class of functions, and its generalization enabled the first 3-share TI for AES without fresh randomness in 2019 [27].

1.1. Purpose and Approach

The question that naturally arises is the cost of the backward compatibility: *how many more gates do we need by choosing AES instead of other lightweight algorithms with TI?* The question has been unanswered because of the gap between the conventional works on lightweight AE and efficient TI implementation: the conventional SAEAES implementations are all without TI [15,19,28]. The purpose of this paper is to implement the first threshold implementation of SAEAES and to evaluate the cost we are trading with the backward compatibility.

Our approach is to extend the recent AES implementation with the 3-share and uniform TI using the generalized changing of the guards [27], but we redesign the AES circuit architecture to satisfy the additional requirements by the mode of operation. Then, we evaluate our design's performance and compared it with the previous implementation of SAEB-GIFT [13]: the same mode of operation instantiated with the state-of-the-art lightweight block cipher GIFT [20].

1.2. Contributions

Here we summarize our key contributions.

(I) Identification of design challenges in extending AES implementation to SAEAES (Section 4) Our design is based on the 3-share and uniform TI of AES using the generalized changing of the guards [27]. We identify that the mode of operation enforces the byte order, making the conventional row-oriented serialization inefficient [23]. Also, the mode of operation should preserve the secret key that the on-the-fly key schedule overwrites.

(II) Column-oriented AES implementation (Section 5.2) We propose a new AES circuit architecture that uses the column-oriented data serialization to address the aforementioned incompatibility with the row-oriented serialization.

(III) The first SAEAES implementation with threshold implementation (Section 5) We show the first TI of SAEAES that uses the column-oriented serialization and the 3-share and uniform AES S-box. The design has an independent key store for preserving the secret key until the next AES call.

(IV) Improved TI of key array (Section 5.5) We show the concrete realization of the key array for TI that reduces the register size by 216 bits or 32% from the original design [27].

(V) Performance evaluation and comparison (Section 6) We synthesize our design using a standard cell library to evaluate its circuit area in GE (gate equivalents). We show that our design uses 18,288 GE with TI composed of AES (14,256 GE, 78%), the key store (3422 GE, 19%), and the mode of operation (610 GE, 3%). Compared with the conventional SAEB-GIFT implementation that uses 6229 GE [13], the SAEAES implementation is roughly three times larger. We identify that the non-linear key schedule and the extended states for satisfying uniformity as the major factors for this difference.

1.3. Organization

This paper is organized as follows. We begin by reviewing the algorithm of SAEAES in Section 2, and the previous TI of AES in Section 3. Then, we state the design challenges we address in the paper in Section 4. We describe our proposed design Section 5 followed by the performance evaluation in Section 6. Section 7 is the conclusion.

2. SAEAES

2.1. Authenticated Encryption

Authenticated encryption (AE) is a cryptographic algorithm that provides confidentiality and integrity using a symmetric key. An AE encryption algorithm converts a plaintext and an associated data into a ciphertext and authentication tag. The corresponding AE decryption algorithm converts the ciphertext and the unencrypted associated data, back into the original plaintext. By using the tag during the decryption, the algorithm checks the integrity, i.e., detects changes in the original ciphertext or the associated data, to prevent forgery attacks. A common AE construction is to combine a block cipher with a mode of operation, and AES [16] with the Galois/counter mode (AES-GCM) is by far the most popular AE approved by NIST SP800-38D [29] and RFC5288 [30], and being used in major systems including SSL/TLS.

Lightweight cryptography is a branch of cryptography on designing cryptographic algorithms that achieve efficient performance in resource-constrained devices. The demand for such lightweight cryptography is higher than ever before for the recent technology trend of adding connectivity to embedded devices. Moreover, NIST is running a standardization process called NIST LWC [2] since 2017, which makes lightweight AE an even more active research area.

2.2. SAEAES and its Algorithm

SAEAES [19] is an instantiation of the lightweight mode of operation SAEB [15] using AES, which is a candidate of NIST LWC [2]. We focus on `SAEAES_128_64_128` with the 128-bit key, 64-bit associated data block, and 128-bit tag among the ten variants.

SAEAES is composed of HASH, Encryption, and Decryption algorithms that process the associated data (AD), plaintext, ciphertext blocks, respectively. SAEAES is based on the sponge construction, in which the data blocks are absorbed into the internal state in between iterated AES calls, as shown in Figure 1.

HASH consumes AD blocks $A_1, \ldots, A_n$ and a 120-bit nonce $\{N_U, N_L\}$, to generate an initial value for the subsequent Encryption or Decryption denoted by $\{IV_U, IV_L\}$. Encryption consumes the message blocks in the same way: for previous AES output $\{t_U, t_L\}$ and the message block M_i, the next AES input is $\{M_i \oplus t_U, t_L\}$ and the corresponding ciphertext block is $C_i = M_i \oplus t_U$. In Decryption, on the other hand, it recovers the message block $M_i = C_i \oplus t_U$, and feeding $\{C_i, t_L\}$ as the next AES input. The tag is the final AES output $\{T_U, T_L\}$, as shown in Figure 1.

The SAEAES (and SAEB) satisfies the following four properties that contribute to a lightweight implementation:

- **(1)** **Minimum state size** No extra memory in addition to AES, which reduces the register size in hardware implementation.
- **(2)** **Inverse free** No need for AES decryption. This reduces the cost of implementing inverse AES operations and overhead for selectors or conditional branches.
- **(3)** **XOR only** The extra operation in addition to AES is XOR only, which is more efficient than other options, such as GF(2^{128}) multiplication in AES-GCM.
- **(4)** **Online** The message and ciphertext blocks are scanned only once. There is no need for a buffer storing the blocks until the second scan.
- **(5)** **Efficient handling of repeated associated data** SAEAES can skip shortcut some operations for several Encryption/Decryption with the same (i.e., repeated) associated data.

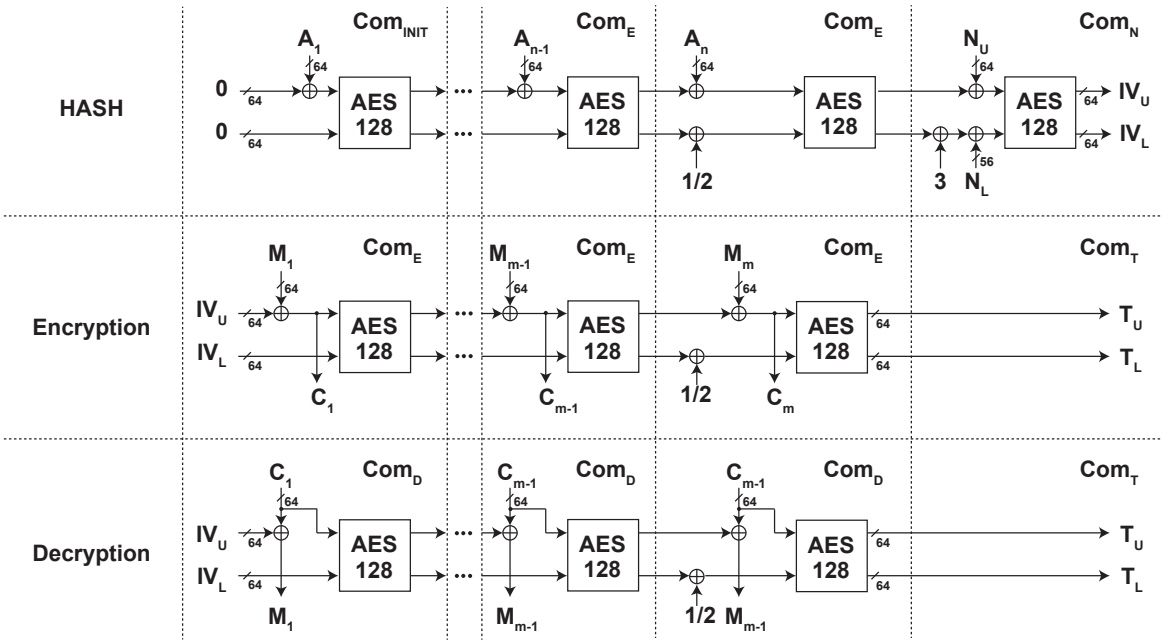

Figure 1. Diagram of SAEAES_128_64_128. $\mathsf{Com}_{\mathsf{INIT}}$, $\mathsf{Com}_{\mathsf{E}}$, $\mathsf{Com}_{\mathsf{N}}$, $\mathsf{Com}_{\mathsf{D}}$, and $\mathsf{Com}_{\mathsf{T}}$ correspond to the commands supported in our implementation (see Section 5.4).

2.3. Hardware Implementations of SAEAES

The original SAEB paper reported a compact hardware implementation of SAEB instantiated with AES [15] using 3502 GE. The design uses the byte-serial architecture [23] that we discuss later in more detail. Balli et al. further reduced the circuit area to 2067 GE using the bit-serial technique, and compared it with with many other AEs using SKINNY and GIFT [28].

There is no TI of SAEAES as far as the authors are aware. Meanwhile, there is a TI of SAEB instantiated with the GIFT lightweight block cipher [13,20]. Caforio et al. evaluated a number of NIST LWC candidates with TI [10].

3. Threshold Implementation of AES

3.1. Side-Channel Attack and Countermeasure

Many embedded devices, such as a smartcard, store a service provider's key and should withstand attacks by the legitimate device owners. Under such a hostile environment, the attacker with physical access uses side-channel attack that exploits information leakage in power consumption and/or electromagnetic radiation [4]. Designing cryptographic modules secure against such attacks is challenging, and researchers have studied new attacks and countermeasures for more than two decades.

Masking based on MPC is by far the most well-studied countermeasure against a power side-channel attack [4,7]. In (additive) masking, we encode a sensitive variable x into a set of variables called a share $\overline{x} = [x_a, x_b, x_c]$ satisfying $x = x_a \oplus x_b \oplus x_c$ thereby randomizing and decoupling the sensitive value from the data representation. An attacker with a limited access to only a proper subset of the share cannot reconstruct the original value x. Masking provides a way to realize a target cryptographic algorithm (e.g., AES) while preserving the shared representation, thereby providing the protection against side-channel attack.

3.2. Threshold Implementation

Threshold implementation (TI) proposed by Nikova et al. [6] is an MPC-based countermeasure suitable for hardware implementation because it can be secure even in the presence of glitches, i.e.,

transient signal propagation through combinatorial circuit inevitable in the common hardware design. In TI, for a target function f, we compose a set of functions called the sharing $\{f_a, f_b, f_c\}$ given by

$$X_a = f_a(x_b, x_c), \qquad X_b = f_b(x_c, x_a), \qquad X_c = f_c(x_a, x_b) \tag{1}$$

wherein $[x_a, x_b, x_c]$ and $[X_a, X_b, X_c]$ are the input and output shares. $\{f_a, f_b, f_c\}$ should satisfy the following three properties:

(1) **Non-completeness** The sharing $\{f_a, f_b, f_c\}$ is *non-complete* if each of f_a, f_b, and f_c receives only a proper subset of the input share $\overline{x} = [x_a, x_b, x_c]$. The sharing given by Equation (1) satisfies *non-completeness* because f_a, f_b, and f_c do not accept x_a, x_b, and x_c, respectively.

(2) **Correctness** The sharing $\{f_a, f_b, f_c\}$ is *correct* if it represents the original function f, i.e., $X = X_a \oplus X_b \oplus X_c = f(x)$.

(3) **Uniformity** The sharing $\{f_a, f_b, f_c\}$ is said to satisfy *uniformity* if it preserves the uniform distribution: a uniform sharing generates a uniformly distributed output share given a uniformly distributed input share. The uniform distribution of the input share is the necessary condition for the TI's security. With *uniformity*, we can feed the output share to the next sharing thereby enabling cascaded connection between sharings.

3.2.1. Composing a Sharing for a Target Function

Designing a sharing satisfying the above three properties for a given target function (e.g., S-box) is an important challenge in TI. Besides, we want to minimize the number of shares because the hardware cost grows quadratically to the number of shares. For a function having an algebraic degree of d, there is a generic way to construct the *correct* and *non-complete* shared function using $d + 1$ shares [31]. Since $d > 1$ for a non-linear function such as an S-box, three is the the minimum number of shares. Target functions with $d > 2$ are commonly decomposed into sub-functions with $d = 2$ for realizing a 3-share sharing.

3.2.2. Lack of Uniformity and Refreshing

The availability of *uniformity* depends on the target function [11]. Consequently, many lightweight algorithms, including GIFT and SKINNY, choose an S-box with which a 3-share and uniform sharing is available [21]. In contrast, the AES and Keccak S-boxes did not have a uniform sharing until very recently. The early AES TI used a non-uniform sharing, and compensated for the lack of *uniformity* by refreshing the output share by adding fresh randomness [23–25]. Although the refreshing saves non-uniform sharing, the need for a lot of fresh randomness raised another implementation problem: the previous implementations need 2560–10,240 random bits for each encryption, which requires a considerable cost in terms of circuit area and power consumption [27].

3.3. Changing of the Guards

Daemen successfully realized a 3-share and uniform Keccak S-box by introducing an elegant technique called the *changing of the guards* [26]. The idea is to construct a sharing of a layer of S-boxes instead of each S-box. Figure 2 (left) shows the *changing of the guards* sharing of the three parallel application of a target function f. In Figure 2 (left), $\{f_a, f_b, f_c\}$ are non-uniform sharing, and we refresh their outputs by adding the shares representing zero, i.e., $[x_c^{i-1} + x_b^{i-1}, x_c^{i-1}, x_b^{i-1}]$ constructed from the neighboring input. This makes the final share uniform again in the same way as the conventional refreshing with a fresh randomness. In other words, the *changing of the guards* recycles the previous input shares as a substitute of fresh randomness in a provable way.

This technique is applicable to any bijective mapping including the Keccak S-box, but its application to the AES S-box was not trivial: field multiplication appears in decomposing the S-box to reduce the number of shares, which is non-bijective. Wegener and Moradi successfully decomposed

the AES S-box into a series of bijective mapping and applied the *changing of the guards*, but it required four shares instead of three [32].

Finally, Sugawara generalized the *changing of the guards* [27] to support non-bijective mapping, and realized the first uniform and 3-share sharing of AES. The idea is to consider the unshared representation of the Daemen's *changing of the guards* as shown in Figure 2 (right). The key point is extending the original function f into a modified function

$$f' : (t, x^i) \mapsto (t + f(x^i), x^i) \tag{2}$$

followed by a null function $\perp$ that maps anything to zero, which ensures the availability of a uniform sharing. By generalizing this extension to a non-bijective mapping by using the generalized Feistel network, we can construct the *changing of the guards* sharing of non-bijective functions including field multiplication. By applying the generalized *changing of the guards* to each stage of the decomposed AES S-box, Sugawara proposed the first 3-share TI without using fresh randomness shown in Figure 3. As a side effect of extending a non-bijective function, the datapath width should be extended from 8 to 14 bits.

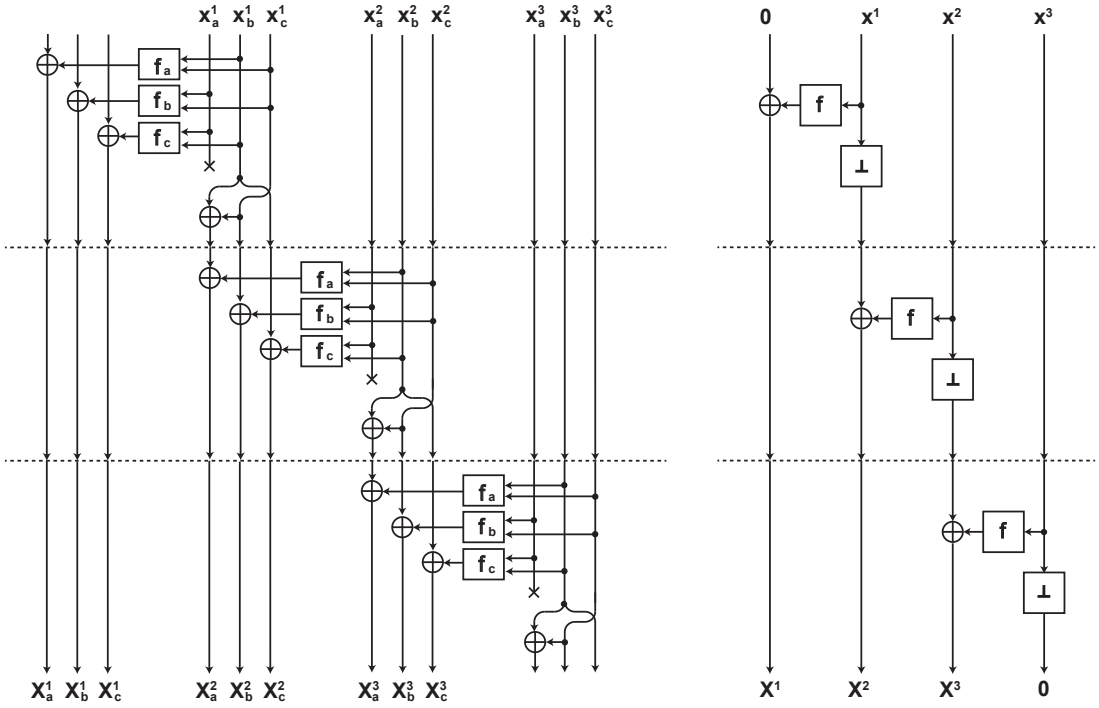

Figure 2. The changing-of-the-guards sharing (**left**) and its unshared representation (**right**).

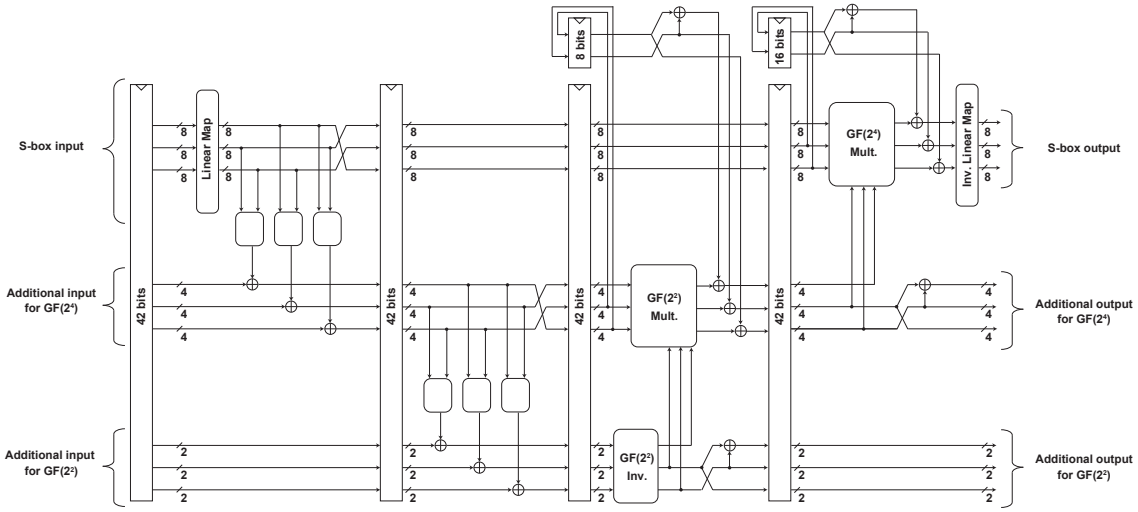

Figure 3. 3-share and uniform TI of the AES S-box based on the generalized *changing of the guards* [27].

4. Design Challenges

Our approach of implementing SAEAES is to extend the previous AES implementation using the generalized *changing of the guards* [27]. In this section, we discuss several challenges we face in the extension.

4.1. Byte Order and Serialization

The byte-serial architecture scans the 16-byte AES state a byte at a clock cycle, which is commonly used for a compact AES implementation. There are the row-oriented and column-oriented serializations. Many of the conventional lightweight AES implementations, including the previous SAEAES implementation [15], follow the rigorously optimized architecture by Moradi et al. [23] that uses the row-oriented state and key arrays in Figure 4.

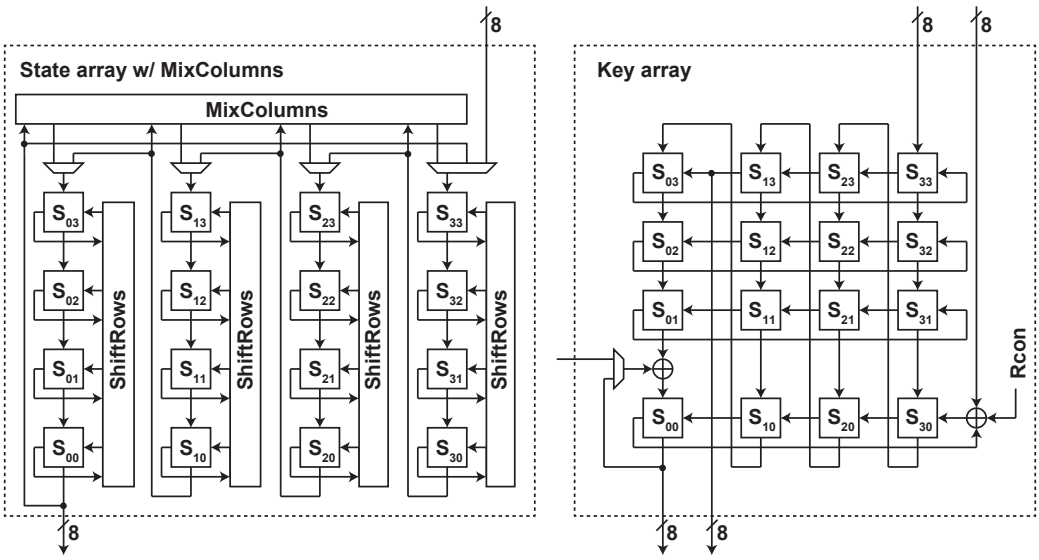

Figure 4. The row-oriented state and key arrays in the conventional compact AES implementation [23].

One drawback of the row-oriented serialization is its incompatibility with the AES's native byte order. This incompatibility has no problem as far as considering a single AES call because we can absorb the difference by redefining the data representation.

However, we cannot change the data representation with a mode of operation because it specifies the byte order for supporting arbitrary-length (i.e., block-unaligned) messages. Table 1 shows the timing we feed an 8-byte string $m_7 \parallel m_6 \parallel \cdots \parallel m_0$ in the column- and row-oriented serialization. With the column-oriented serialization, we can feed the bytes in the original order every cycle. With the row-oriented serialization, on the other hand, we need to reorder the bytes and feed them in an interleaved manner. This reordering and synchronization costed an additional 56-bit shift register in the previous SAEAES implementation [15].

Table 1. The byte order for feeding an 8-byte byte string $m_7 \parallel m_6 \parallel \cdots \parallel m_0$ in the column- and row-oriented serialization.

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
column-oriented	m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0	—	—	—	—	—	—	—	—
row-oriented	m_7	m_3	—	—	m_6	m_2	—	—	m_5	m_1	—	—	m_4	m_0	—	—

4.2. On-The-Fly Key Schedule

On-the-fly key schedule is a common technique for reducing the register cost that updates the secret key in place for key schedule. Although overwriting an original key is not a problem for a particular application that uses a single AES call (e.g., challenge–response authentication), SAEAES needs the same secret key for processing the next block. There are two ways to get the same key again: (i) storing it in another register or (ii) recovering the original data by implementing the reverse key schedule. In this paper, we use the first approach—discussed in Section 5.3.

4.3. Scan Flip Flop and Clock Gating

The previous design further optimize the row-oriented arrays (Figure 4) using the netlist-level (cf. register-transfer level (RTL)) optimization [23]. The design uses scan flip flops (SFFs), a register with a builtin selector, which is more efficient than an individual register and selector combined. Moreover, the design uses a gated clocking to control the data flow instead of implementing an enable signal using a selector with a feedback line. These techniques are very efficient and followed by many designs [13,14,21,23].

Meanwhile, the SFF's original purpose is to instrument a design with a scan-path chain and provides a way for testing after fabrication, and not for optimizing a design. A logic synthesizer does not infer an SFF unless a designer explicitly instantiate the cell in the code. This type of optimization binds the design to a specific standard-cell library, and is unavailable to some designers who releases a synthesizable RTL design (e.g., IP vendors). Also, some conservative coding/design rules prohibit such an aggressive optimization for a possible incompatibility with an automated scan-path insertion. Moreover, avoiding glitches on the gated clock signal needs a careful design using a dual-phase or dual-edge clock that increases the engineering cost. To avoid these disadvantages, we use neither SFF nor gated clock, as we discuss in Section 5.1.

5. Proposed Design

5.1. Design Policy

We use a conservative design policy based on the discussion in Section 4. We describe the design at the register-transfer level (RTL); we use no netlist-level optimization, including the direct standard-cell instantiation of SFFs, so that the design will not bound to a specific library. The design completely synchronizes to a single-edged clock and uses no gated clock.

The circuit area is the primary performance target. By considering the design choices in software/hardware codesign, and following the common practice of letting hardware do regular operations for improving the efficiency, we choose the coprocessor interface aiming at accelerating the main time-consuming part of AD processing, encryption, and decryption. Meanwhile, the design

relies on an external processor for handling exceptional cases such as padding following the previous works [10,13–15,28]. Meanwhile, there is another approach of including the padding circuit inside the design, which is more suitable for a high-speed design with a direct memory access [33]. For implementing one-zero padding for SAEAES, we will need an additional input for indicating the end of the message, and some selector and AND gates for overwriting the incoming data stream.

The implementation holds several parameters during their lifetime. In particular, it preserves the secret key over multiple AES calls to eliminate the hidden cost of an external key register. The design assumes an asynchronous register interface: an external controller needs no cycle-accurate synchronization.

5.2. Column-Oriented Serialization

We propose the column-oriented arrays in Figure 5 to address the issue of the row-oriented serialization discussed in Section 4.1.

5.2.1. State Array

Figure 5 (left) shows the column-oriented state array. Figure 5 explicitly shows selectors attached to registers because we do not use SFFs. The circuit uses the vertical links for shifting the serialized data, and the horizontal links for MixColumns and ShiftRows. In particular, we realize ShiftRows by shifting the data using the horizontal links while controlling the number of shifts using enable signals. This array finishes an AES round using 27 cycles, as shown in Table 2: 16 cycles for S-box, 4-cycles latency for a pipelined S-box, 3 cycles for ShiftRows, and 4 cycles for MixColumns.

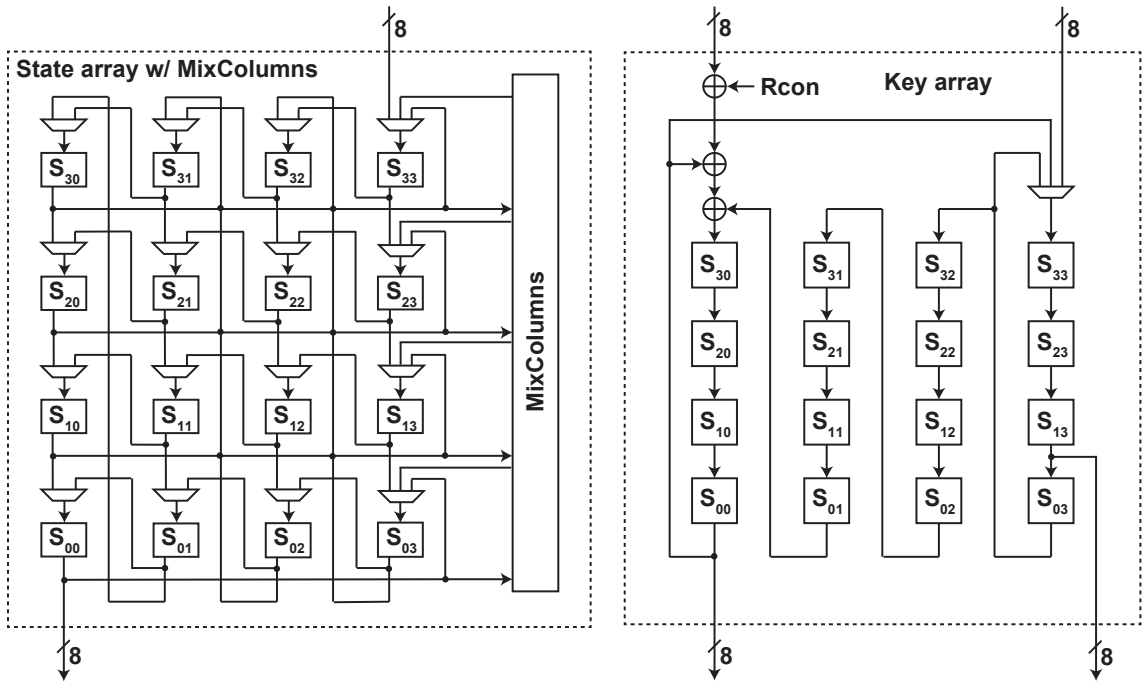

Figure 5. The column-oriented state and key arrays. The all registers have an enable signal for controlling their data flow.

Table 2. AES round in 27 cycles: operations of the state and key arrays with a 4-stage pipelined S-box.

Cycle	State Array	Key Array
1	S-box lookup	Output
⋮	⋮	⋮
16	S-box lookup	Output
17	Pipeline Latency	S-box lookup
18	Pipeline Latency	S-box lookup
19	Pipeline Latency	S-box lookup
20	Pipeline Latency	S-box lookup
21	ShiftRows	Pipeline Latency
22	ShiftRows	Pipeline Latency
23	ShiftRows	Pipeline Latency
24	MixColumns	Pipeline Latency
25	MixColumns	—
26	MixColumns	—
27	MixColumns	—

5.2.2. Key Array

As shown in Figure 5 (right), the column-oriented key array has a simplified datapath for the AES's key schedule being column-oriented. It uses no horizontal link, which significantly reduces the circuit area without an SFF, i.e., when a selector is not for free. Table 2 summarizes the key array's operations in each cycle. The key array works as a shift register for the first 16 cycles. It then feeds the bytes in the fourth column to S-box for SubWord during the 17–20th cycles. Here, we use S_{13} (cf. S_{03}, see Figure 5) as an S-box input to realize RotWord. The XOR gate connected to S_{30} calculates the XOR between the neighboring columns while shifting the data in the 1–16th cycles.

5.2.3. Comparing Row-Oriented and Column-Oriented Arrays

Table 3 compares the circuit areas of the row- and column-oriented arrays after logic synthesis (See Section 6 for the tool, library, and conditions for the performances evaluation). The row-oriented arrays, (SR) and (KR) in Table 3, implement the ones in Figure 4 using SFFs, while the columns-oriented arrays (SC) and (KC) implement the ones in Figure 5 using an ordinary register with an enable signal. The SFF is more efficient than a register with a selector, and (SC) is larger than (SR). Meanwhile, (KC) is smaller than (KR) because the simplified data flow eliminates most of the selectors on the registers. Although the row-oriented design is still better by 161 GE in total, it is relatively minor compared to the entire circuit area. Thus, it is a reasonable cost for unbinding the design from a specific standard cell library and reducing the engineering cost for handling multiple clocks.

Table 3. Performance comparison of the row- and column-oriented arrays.

Identifier	Target	Diagram	Orientation	Primitive	Area [GE]
(SR)	State Array	Figure 4-left	Row	Scan FF	1219
(SC)	State Array	Figure 5-left	Column	FF with enable	1388
(KR)	Key Array	Figure 4-right	Row	Scan FF	1075
(KC)	Key Array	Figure 5-right	Column	FF with enable	1067

5.3. AES Implementation

Figure 6 shows the proposed SAEAES design, including AES implementation composed of the key store, S-box implementation, and column-oriented arrays. The 8-bit buses receive the message and key bytes in the AES's native order, i.e., the column orientation. A single AES round takes 27 cycles (see Table 2), and the entire AES operation finishes in 282 cycles.

The key store is the 8-bit and 16-stage shift register with a feedback that stores the original key (see Section 4.2), and transfers it to the key array at the beginning of an AES encryption. We avoid a reverse key schedule because it has a significant impact on the key array and the S-box circuit in addition to doubling the latency. We use the Canright's AES S-box representation [34] divided into four pipelined stages, which is necessary for TI, following the previous work [27].

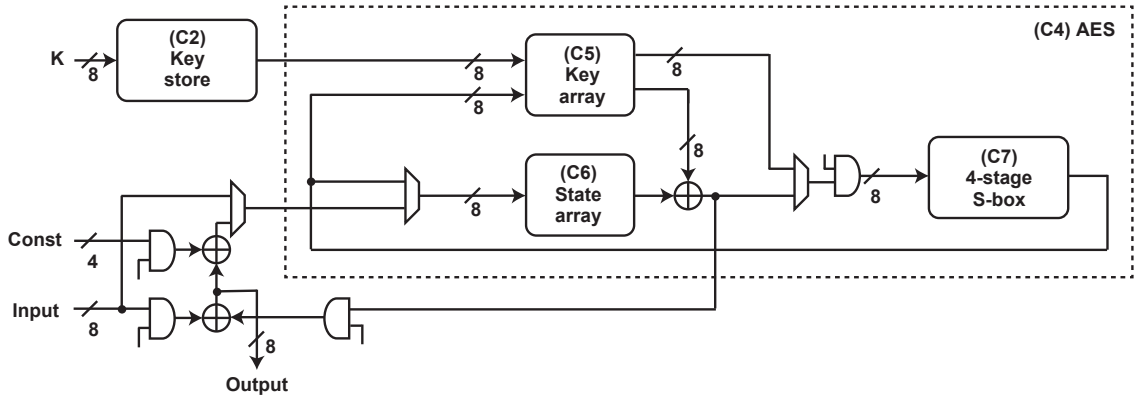

Figure 6. Datapath diagram of our design without threshold implementation (TI). (C1)–(C8) are identifiers used in Table 6.

5.4. SAEAES Implementation

The mode of operation is a thin wrapper by following the previous SAEB and SAEAES implementations [13,15], as shown in Figure 6. The wrapper consists of some 8-bit AND, XOR, and selector gates for changing the datapath depending on the target operation.

The SAEAES implementation supports five commands: $\mathsf{Com}_{\mathsf{INIT}}$, $\mathsf{Com}_{\mathsf{E}}$, $\mathsf{Com}_{\mathsf{N}}$, $\mathsf{Com}_{\mathsf{D}}$, and $\mathsf{Com}_{\mathsf{T}}$ that involve at most one AES call, as summarized in Table 4. Figure 1 shows how these commands realize the SAEAES's HASH, Encryption, and Decryption. Figure 7 illustrates the active path on the simplified diagrams for each command. The XOR and AND gates control the next input to AES by combining the previous AES output, the input message/ciphertext byte, and a domain-separation constant. We use the same XOR for generating the output and tag.

The circuit has an 8-bit FIFO-like interface: a user pushes input bytes into the circuit, which updates the output bytes simultaneously. The circuit starts an AES encryption after receiving the sufficient number of bytes.

Table 4. Five commands that SAEAES implementation supports. See Figure 1 for the corresponding operation in SAEAES, and Figure 7 for the active datapath in the proposed design.

Command	Description
$\mathsf{Com}_{\mathsf{E}}$	Absorbing 64-bit data block in Hash and Encryption
$\mathsf{Com}_{\mathsf{INIT}}$	A special case of $\mathsf{Com}_{\mathsf{E}}$ without a feedback
$\mathsf{Com}_{\mathsf{N}}$	Absorbing a 120-bit nonce at the end of Hash
$\mathsf{Com}_{\mathsf{D}}$	Absorbing a 64-bit ciphertext block in Decryption
$\mathsf{Com}_{\mathsf{T}}$	Output the entire 128-bit state as a tag

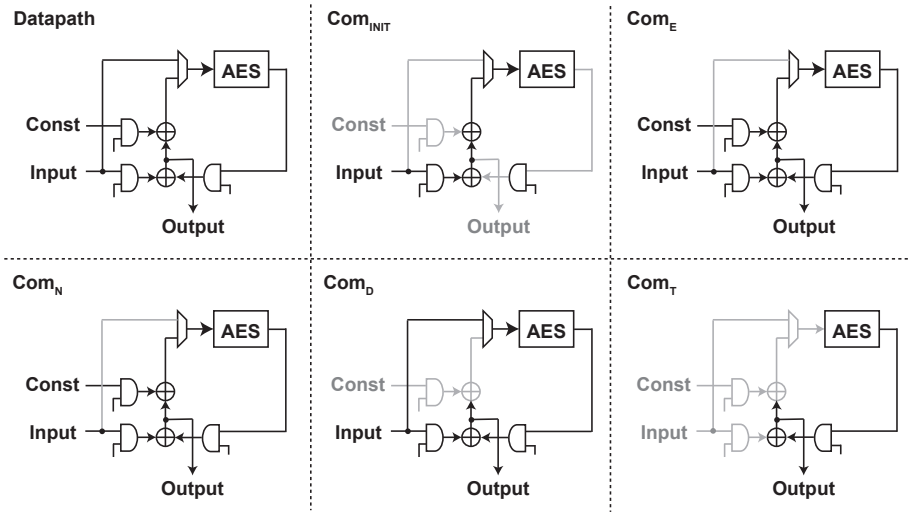

Figure 7. The active datapaths in each of the $\mathsf{Com}_{\mathsf{INIT}}$, $\mathsf{Com}_{\mathsf{E}}$, $\mathsf{Com}_{\mathsf{N}}$, $\mathsf{Com}_{\mathsf{D}}$, and $\mathsf{Com}_{\mathsf{T}}$ command.

5.5. Threshold Implementation

Figure 8 shows our design that implements a 3-share and uniform TI with a protected key schedule that provides the security up to the first-order attacks.

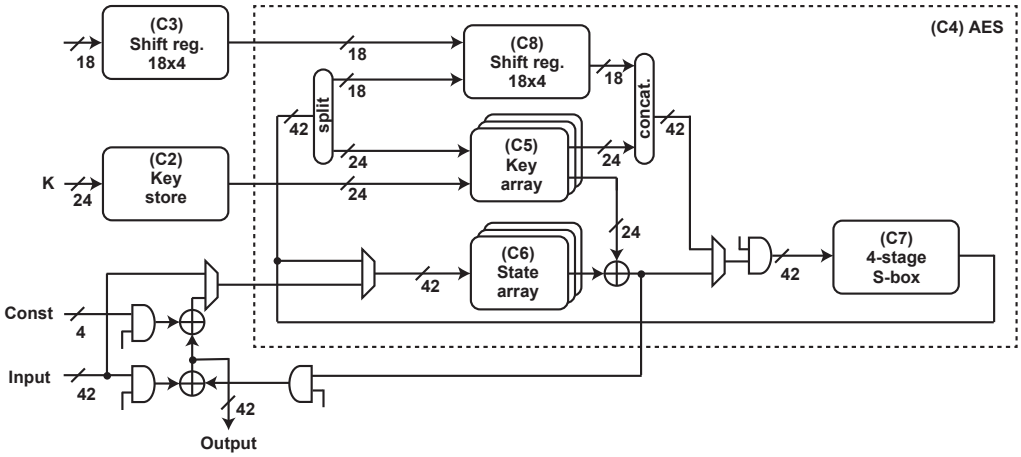

Figure 8. Datapath diagram of our design with TI. (C1)–(C8) are identifiers used in Table 6.

5.5.1. S-box

We use the 3-share and uniform S-box [27] in Figure 3 both for the round operation and key schedule (see Table 2 for the timing). The S-box circuit has the 42-bit datapath width, and we extend the entire datapath, including the input and output buses, to 42 bits. A user feeds the shared representation of a message/key, 42 bits at a time, by taking 16 cycles. The timing is the same as the unprotected implementation, and an AES call takes 282 cycles in total.

Table 5. Register-size comparison of our unprotected and TI designs.

Component	Unprotected [bits]	TI [bits]
Total	416	2208
State array	128	672 $(= 14 \times 16 \times 3)$
Key array	128	456 $(= 8 \times 16 \times 3 + 18 \times 4)$
Key store	128	456 $(= 8 \times 16 \times 3 + 18 \times 4)$
S-box	32 $(= 4 \times 8)$	192 $(= 14 \times 4 \times 3 + 16 + 8)$

5.5.2. State Array

We need to extend the S-box's input size from 8 to 14 bits for the generalized *changing of the guards*. To store these 14-bit data, we extended the column-oriented arrays to store 224 (=14×16) bits. We then duplicate the extended arrays to store the intermediate data in a shared representation. As a result, the state uses 672 (=$14 \times 16 \times 3$) bits, as summarized in Table 5.

5.5.3. Key Array and Key Store

We store the secret key in the duplicated key arrays and an independent shift register. The previous implementation [27] extends the entire key array from 8 to 14 bits, similar to the state array, resulting in 672 bits of registers. However, since only 4 out of 16 bytes go through the S-box calculation, the previous design wastes the 216 (= $(16-4) \times 6 \times 3$) extended bits. The previous work pointed out this inefficiency but gave no concrete realization [27].

Instead of extending the bit width of the key array, we add an independent 18-bit and 4-stage shift register for storing the extended bits ((C8) in Figure 6). As a result, the duplicated key array and the new shift register combined use 456 bits of registers as summarized in Table 5, reducing 216 bits from the previous design. We implement the key store in the same way using 456 bits of registers ((C2) and (C3) in Figure 6).

For each AES call, we need 448 (= $(14 \times 16 \times 2)$) random bits to make a shared representation of the AES's input. We use 304 (= $(8 \times 16 \times 2 + 6 \times 4 \times 2)$) random bits for making a key share for rekeying: 256 bits for a shared representation of the 128-bit key, and additional 48 bits for the extended bits. In addition, we need 24 random bits for initializing the S-box circuit once at the time of boot.

6. Evaluation

6.1. Performance Evaluation

We synthesized the designs using Synopsys Design Compiler with the NanGate 45-nm standard cell library [35]. For a component-wise comparison, we preserved the module hierarchy up to the major components. Table 6 summarizes the post-synthesis performances of each component.

Table 6. Post-synthesis performance of our designs. See the diagrams in Figures 6 and 8 for the identifiers (C1)–(C8).

Identifier	Component	Unprotected [GE]	TI [GE]
(C1)	Total	4690	18,288
(C2)	Total/Key Store	961	2877
(C3)	Total/Shift Register 18×4	—	545
(C4)	Total/AES	3423	14,256
(C5)	Total/AES/Key Array	1067	3222
(C6)	Total/AES/State Array	1373	6553
(C7)	Total/AES/S-box	533	3218
(C8)	Total/AES/Shift Register 18×4	—	545

Our unprotected SAEAES implementation uses 4690 GE. Considering that our design has the key store (961 GE), this size is comparable to that of the previous SAEAES implementation (3502 GE [15]) that needs an external key register. Our design has some disadvantages due to the conservative design policy and the compatibility with TI: (i) the state and key arrays are larger for not using the netlist-level optimization (see Table 3) and (ii) the S-box circuit is pipelined. These disadvantages, however, are canceled out by the elimination of the 56-bit shift register (roughly 400 GE) needed in the previous design for reordering the bytes [15] (see Section 4.1). Our design has room for further optimization with a more aggressive design policy.

Our TI design uses 18,288 GE. The underlying AES implementation uses 14,256 GE, which is smaller than the previous implementation with 17.1 kGE [27]. Reducing the key-related registers from

672 to 456 bits (see Section 5.5 and Table 5) is the main reason for this improvement. The sizes of the state array and the S-box circuit are similar to the previous design.

The mode of operation uses 610 (=18288 − 2877 − 545 − 14256) gates or 3.3% with TI, i.e., AES occupies 97% of the total area. The SAEB's minimum state size and XOR-only properties [15] contribute to this small footprint. AES-GCM, on the other hand, needs additional 512 bits of registers corresponding to 3844 GE, which expands to 1280 bits or 9610 GE with TI (with the estimated register cost of $\frac{961}{128}$ GE/bit based on (C2) in Table 6). We note that AES-GCM also needs an independent protection to its GF(2^{128}) multiplication [36].

The register storing the shared representation of the key occupies 5161 GE or 28% (obtained by subtracting the key-related size in the unprotected implementation (2028 = 961 + 1067) from that in the TI implementation (7189 = 2877 + 3222 + 545 + 545)). Some of the previous implementations have an unprotected key schedule by considering non-profiling attacks only [21,25,37]. If we use such an unprotected key schedule in our design, the total circuit will be roughly 13 kGE by saving 5161 GE.

6.2. Comparison with SAEB-GIFT and Other AEADs

Table 7 compares our design with the previous implementation of SAEB-GIFT: SAEB instantiated with the GIFT block cipher [15]. The SAEB-GIFT implementation [13] uses 6229 GE, which is roughly 1/3 of our SAEAES implementation. Since both implementations use the same mode of operation (SAEB), the difference comes from that of AES and GIFT. The key store, non-linear key schedule, and S-box are the key factors of the difference.

6.2.1. Key Store

In comparing the unprotected implementations, the key store (961 GE in Table 5) is the major reason for the SAEB-GIFT implementation (2761 GE) being smaller than ours by 1929 GE. As discussed in Section 5, the SAEAES implementation uses the key store because the reverse key schedule is so expensive in AES. Meanwhile, the SAEB-GIFT implementation uses an efficient reverse key schedule [15] by exploiting the GIFT key schedule defined as a simple nibble permutation [20]. The key store becomes even larger with TI for storing the shared representation of the key.

6.2.2. Non-Linear Key Schedule

In contrast to the non-linear AES key schedule that needs 3 shares, we can protect the GIFT's linear key schedule with only 2 shares. This 2-share representation reduces the memory capacity for the key array. Indeed, the recent TI-friendly authenticated encryptions [13,14] exploit this linear part to improve the performance with TI. As a result, the GIFT's key array is 2410 GE with TI, while AES needs 7189 GE for storing the key ((C2), (C3), (C5), and (C8)).

6.2.3. S-Box

Even with an unprotected key schedule, our implementation is larger than that of SAEB-GIFT by 8084 GE. The main reason is the increased state size by the generalized *changing of the guards*: the need for extending the data width from 8 to 14 bits increase the memory size by $\times$1.75 (=14/8). In contrast, GIFT needs no such extension because the designers chose an S-box that has a uniform sharing [21]. We can reduce the memory size by using the previous non-uniform sharing [23–25], but we need to implement an efficient random numbers as discussed in Section 3.

6.2.4. Other AEADs

Table 7 also shows the performances of the other authenticated encryptions: (i) the Arribas et al. KETJE-JR implementation based on the *changing of the guards* sharing [9] and (ii) the Caforio et al. implementations of several NIST LWC candidates based on GIFT-128 [10], which have the similar circuit sizes compared to that of SAEAES. These implementations are larger because they traded the circuit area

with speed: they finish a round function each cycle by using multiple S-box circuits (cf. 27 cycles/round in our implementation). This comparison gives another insight about the cost of using AES.

Table 7. Comparison with SAEB-GIFT.

Target	Unprotected [GE]	TI [GE]	TI † [GE]	Ref.
SAEAES	4690	18,288	13,121	This work
SAEB-GIFT	2761	6229	5037	[13]
KETJE-JR	6109	20,032	—	[9]
GIFT-COFB (3-share)	4700	—	16,386	[10]
SUNDAE-GIFT (3-share)	3548	—	13,297	[10]
HYENA (3-share)	3850	—	14,796	[10]

† TI with unprotected key scheduling.

7. Conclusions

We presented the first TI of the authenticated encryption algorithm SAEAES. We used the Sugawara's 3-share and uniform TI of AES S-box [27], but completely redesigned the internal data structures (the state and key arrays) because SAEAES prefers the column-oriented serialization. We show that our design achieves 18,288 GE with TI. Meanwhile, it is roughly three times larger than the SAEB-GIFT implementation using 6229 GE [13]. Since both implementations use the same mode of operation (SAEB), AES is responsible for the larger area: the main difference comes from the larger number of registers needed for the non-linear key schedule and the larger states extended by the generalized *changing of the guards*.

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Bogdanov, A.; Knudsen, L.R.; Leander, G.; Paar, C.; Poschmann, A.; Robshaw, M.J.B.; Seurin, Y.; Vikkelsoe, C. PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems—CHES 2007, Proceedings of the 9th International Workshop, Vienna, Austria, 10–13 September 2007*; Paillier, P., Verbauwhede, I., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4727, pp. 450–466. [CrossRef]
2. National Institute of Standards and Technology (NIST). Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process. 2018. Available online: https://csrc.nist.gov/Projects/lightweight-cryptography (accessed on 7 August 2020).
3. Kocher, P.C.; Jaffe, J.; Jun, B. Differential Power Analysis. In *Advances in Cryptology—CRYPTO '99, Proceedings of the 19th Annual International Cryptology Conference, Santa Barbara, CA, USA, 15–19 August 1999*; Wiener, M.J., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1666, pp. 388–397. [CrossRef]
4. Mangard, S.; Oswald, E.; Popp, T. *Power Analysis Attacks—Revealing the Secrets of Smart Cards*; Springer: Berlin/Heidelberg, Germany, 2007.
5. Sönmez, M. On the NIST Lightweight Cryptography Standardization. In Proceedings of the 23rd Workshop on Elliptic Curve Cryptography (ECC 2019), Bochum, Germany, 2–4 December 2019.
6. Nikova, S.; Rechberger, C.; Rijmen, V. Threshold Implementations Against Side-Channel Attacks and Glitches. In *Information and Communications Security, Proceedings of the 8th International Conference, ICICS 2006, Raleigh, NC, USA, 4–7 December 2006*; Ning, P., Qing, S., Li, N., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2006; Volume 4307, pp. 529–545. [CrossRef]

7. Ishai, Y.; Sahai, A.; Wagner, D.A. Private Circuits: Securing Hardware against Probing Attacks. In *Advances in Cryptology—CRYPTO 2003, Proceedings of the 23rd Annual International Cryptology Conference, Santa Barbara, CA, USA, 17–21 August 2003*; Boneh, D., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2729, pp. 463–481. [CrossRef]
8. Groß, H.; Wenger, E.; Dobraunig, C.; Ehrenhöfer, C. Suit up!—Made-to-Measure Hardware Implementations of ASCON. In Proceedings of the 2015 Euromicro Conference on Digital System Design, Funchal, Portugal, 26–28 August 2015; pp. 645–652.
9. Arribas, V.; Nikova, S.; Rijmen, V. Guards in Action: First-Order SCA Secure Implementations of Ketje Without Additional Randomness. In Proceedings of the 21st Euromicro Conference on Digital System Design (DSD), Prague, Czech Republic, 29–31 August 2018; pp. 492–499.
10. Caforio, A.; Balli, F.; Banik, S. Energy Analysis of Lightweight AEAD Circuits. Cryptology ePrint Archive, Report 2020/607, 2020. Available online: https://eprint.iacr.org/2020/607 (accessed on 7 August 2020).
11. Beyne, T.; Bilgin, B. Uniform First-Order Threshold Implementations. In *Proceedings of the International Conference on Selected Areas in Cryptography, Ottawa, ON, Canada, 10–12 August 2016*; Springer: Cham, Switzerland, 2016; Volume 10532, pp. 79–98
12. Gao, S.; Roy, A.; Oswald, E. Constructing TI-Friendly Substitution Boxes Using Shift-Invariant Permutations. In *Topics in Cryptology—CT-RSA 2019, Proceedings of the The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, 4–8 March 2019*; LNCS: Springer: Berlin/Heidelberg, Germany, 2019; Volume 11405, pp. 433–452.
13. Naito, Y.; Sugawara, T. Lightweight Authenticated Encryption Mode of Operation for Tweakable Block Ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, *2020*, 66–94. [CrossRef]
14. Naito, Y.; Sasaki, Y.; Sugawara, T. Lightweight Authenticated Encryption Mode Suitable for Threshold Implementation. In *Advances in Cryptology—EUROCRYPT 2020, Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, 10–14 May 2020*; Part II; Canteaut, A., Ishai, Y., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2020, Volume 12106, pp. 705–735. [CrossRef]
15. Naito, Y.; Matsui, M.; Sugawara, T.; Suzuki, D. SAEB: A Lightweight Blockcipher-Based AEAD Mode of Operation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, *2018*, 192–217.
16. National Institute of Standards and Technology (NIST). *Federal Information Processing Standards Publication 197: ADVANCED ENCRYPTION STANDARD (AES)*; National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2001.
17. Gueron, S.; Jha, A.; Nandi, M. COMET: COunter Mode Encryption with Authentication Tag. 2019. Available online: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/comet-spec-round2.pdf (accessed on 7 August 2020).
18. Chakraborty, B.; Nandi, M. MixFeed. 2019. Available online: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/mixFeed-spec-round2.pdf (accessed on 7 August 2020).
19. Naito, Y.; Matsui, M.; Sakai, Y.; Suzuki, D.; Sakiyama, K.; Sugawara, T. *SAEAES*. 2019. Available online: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/SAEAES-spec-round2.pdf (accessed on 7 August 2020).
20. Banik, S.; Pandey, S.K.; Peyrin, T.; Sasaki, Y.; Sim, S.M.; Todo, Y. GIFT:A Small Present—Towards Reaching the Limit of Lightweight Encryption. In Proceedings of the Cryptographic Hardware and Embedded Systems (CHES), LNCS, Taipei, Taiwan, 25–28 September 2017; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10529, pp. 321–345.
21. Beierle, C.; Jean, J.; Kölbl, S.; Leander, G.; Moradi, A.; Peyrin, T.; Sasaki, Y.; Sasdrich, P.; Sim, S.M. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In *Advances in Cryptology—CRYPTO 2016, Proceedings of the 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, 14–18 August 2016*; Part II; Robshaw, M., Katz, J., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9815, pp. 123–153. [CrossRef]
22. National Institute of Standards and Technology (NIST). Announcing the ADVANCED ENCRYPTION STANDARD (AES). FIPS PUB 197. 2001. Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf (accessed on 7 August 2020).

23. Moradi, A.; Poschmann, A.; Ling, S.; Paar, C.; Wang, H. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *Advances in Cryptology—EUROCRYPT 2011, Proceedings of the 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, 15–19 May 2011*; Paterson, K.G., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6632, pp. 69–88. [CrossRef]
24. Bilgin, B.; Gierlichs, B.; Nikova, S.; Nikov, V.; Rijmen, V. Trade-Offs for Threshold Implementations Illustrated on AES. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2015**, *34*, 1188–1200. [CrossRef]
25. Ueno, R.; Homma, N.; Aoki, T. Toward More Efficient DPA-Resistant AES Hardware Architecture Based on Threshold Implementation. In *Constructive Side-Channel Analysis and Secure Design, Proceedings of the 8th International Workshop, COSADE 2017, Paris, France, 13–14 April 2017*; Revised Selected Papers; Guilley, S., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10348, pp. 50–64. [CrossRef]
26. Daemen, J. Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing. In *Cryptographic Hardware and Embedded Systems—CHES 2017, Proceedings of the 19th International Conference, Taipei, Taiwan, 25–28 September 2017*; Fischer, W., Homma, N., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10529, pp. 137–153. [CrossRef]
27. Sugawara, T. 3-Share Threshold Implementation of AES S-box without Fresh Randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2019**, *2019*, 123–145. [CrossRef]
28. Balli, F.; Caforio, A.; Banik, S. Low-latency Meets Low-Area: An Improved Bit-Sliding Technique for AES, SKINNY and GIFT. Cryptology ePrint Archive, Report 2020/608, 2020. Available online: https://eprint.iacr.org/2020/608 (accessed on 7 August 2020).
29. Dworkin, M. *Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*; National Institute of Standards & Technology: Boulder, CO, USA, 2017.
30. Salowey, J.; Choudhury, A.; McGrew, D. *RFC5288: AES Galois Counter Mode (GCM) Cipher Suites for TLS*; Internet Engineering Task Force: Fremont, CA, USA, 2008.
31. Nikova, S.; Rijmen, V.; Schläffer, M. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *J. Cryptol.* **2011**, *24*, 292–321. [CrossRef]
32. Wegener, F.; Moradi, A. A First-Order SCA Resistant AES Without Fresh Randomness. In *Constructive Side-Channel Analysis and Secure Design, Proceedings of the 9th International Workshop, COSADE 2018, Singapore, 23–24 April 2018*; Springer: Cham, Seitzerland, 2018; pp. 245–262. [CrossRef]
33. Rezvani, B.; Diehl, W. Hardware Implementations of NIST Lightweight Cryptographic Candidates: A First Look. *IACR Cryptol. ePrint Arch.* **2019**, *2019*, 824.
34. Canright, D. A Very Compact S-Box for AES. In *Cryptographic Hardware and Embedded Systems—CHES 2005, Proceedings of the 7th International Workshop, Edinburgh, UK, 29 August–1 September 2005*; Rao, J.R., Sunar, B., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3659, pp. 441–455. [CrossRef]
35. NanGate. NanGate FreePDK45 Open Cell Library. Available online: https://si2.org/open-cell-library (accessed on 7 August 2020).
36. Oshida, H.; Ueno, R.; Homma, N.; Aoki, T. On Masked Galois-Field Multiplication for Authenticated Encryption Resistant to Side Channel Analysis. In *Constructive Side-Channel Analysis and Secure Design, Proceedings of the 9th International Workshop, COSADE 2018, Singapore, 23–24 April 2018*; Springer: Cham, Seitzerland, 2018; pp. 44–57. [CrossRef]
37. Poschmann, A.; Moradi, A.; Khoo, K.; Lim, C.; Wang, H.; Ling, S. Side-Channel Resistant Crypto for Less than 2300 GE. *J. Cryptol.* **2011**, *24*, 322–345. [CrossRef]