Pushing the Limits Further: Sub-Atomic AES

Markus Stefan Wamser(✉) and Georg Sigl

Lehrstuhl für Sicherheit in der Informationstechnik,
Technische Universität München, 80333 Munich, Germany
{wamser,sigl}@tum.de
http://www.sec.ei.tum.de/

Abstract. The recent trend to connect a plethora of sensors, embedded and ubiquitous systems with low computing power, in short the rise of the Internet of Things, has created a great demand for compact, lightweight and cheap to produce implementations of cryptographic primitives.

One approach to meet this demand is the development and standardisation of new tailored primitives, most prominently PRESENT. Yet, the wide proliferation of the Advanced Encryption Standard and the trust it earned through its long history of withstanding cryptanalysis spurred anew the search for small, lightweight implementations of AES.

Among the smallest published architectures is the AtomicAES design by Banik et al., who reported a design size of just over 2000 GE.

Here we present a new 8-bit serial architecture that has been designed from careful observation of the minimum required connections between storage elements to support all dataflows required for execution of the algorithm. While we reach similar conclusions to previous publications, the new architecture enables us to push the area requirement for a fully featured AES primitive further down by more than 8% from the area requirement of AtomicAES while offering more functionality.

Along the way we also answer in the affirmative the open question whether the AES reverse keyschedule can be implemented with negligible hardware overhead based on the forward keyschedule.

Our design sets a new record for an 8-bit serial architecture with full functionality for encryption and decryption including the keyschedule, as well as for a sole encryption architecture. Furthermore our design is flexible enough to allow scaling the S-Box architecture from single-cycle to multi-stage pipelined approaches as are required for high operation frequencies or for protection against side-channel attacks. We demonstrate this by instantiating the design with a serial version of the S-Box to reduce the area requirement even further.

Keywords: AES · Lightweight · 8-bit-serial · ASIC · Block cypher · S-Box

Published by Springer Nature Switzerland AG 2019
M. Maniatakos et al. (Eds.): VLSI-SoC 2017, IFIP AICT 500, pp. 220–239, 2019.
https://doi.org/10.1007/978-3-030-15663-3_11

1 Introduction

In recent years small and resource-constrained computing platforms and embedded systems became ubiquitously present. With the recent growth of the Internet of Things (IoT), those systems are becoming increasingly connected among each other and with more powerful dedicated servers or applications in the cloud. Along with sensitive data such as firmware or personal data on the devices, all this communication needs cryptographic protection. To solve this problem, many new block cyphers have been conceived, such as NOEKEON [10], PRESENT [6], KATAN [11], PRINTcipher [18], piccolo [24], TWINE [25], LED [15], KLEIN [14], PRINCE [7], SIMON and SPECK [5], RECTANGLE [28] or Midori [2] to name just a few.[1]

While some, such as PRESENT and CLEFIA, have been standardised [17], these new cyphers have not gained as much trust as the well scrutinised Advanced Encryption Standard (AES) nor can they compete with the latter's proliferation. Therefore research on lightweight implementations of AES has gained momentum in the recent years. Moreover, not only interoperability issues, but also the need to defend against side-channel attacks motivates the search for small implementations, because countermeasures against such attacks usually increase the size of the design by a factor, rather than simply adding a constant overhead. The notion of *lightweight* usually denotes low area designs with low energy consumption that "pay" for these optimisations with slightly increased latency. Sometimes the focus is more on (peak) power consumption than on energy consumption. The exact definition of *lightweight* and the ideal tradeoff are usually driven by the actual application scenario. Most of these scenarios, however, have in common that they do not need bulk encryption, but are rather implemented to secure transmission of infrequent small data packages or are used for authentication purposes on SmartCards. In all cases hardware area is at a premium and there is some flexibility in the acceptable latency.

Given the wide range of optimisation targets and the popularity of AES, it is impossible to give a comprehensive account of implementations. Short overviews can be found in [13,22] and [9] (the latter with a focus on FPGA implementations). With respect to compact hardware and a focus on ASICs, some notable publications follow. All these architectures employ serialisation as basic technique of area-runtime tradeoff. The structure of AES suggests a datapath width of 8-bit, given by the S-Box. The S-Box-implementation of [8] is used by all publications unless otherwise noted. Sizes are given in *gate equivalents (GEs)*, the number of 2-input NAND gates from the same cell library that would cover an equivalent area.

Motivated by the column-wise operations of AES, a 32-bit wide serial datapath is used by [23] along with a tower-field approach to the S-Box to achieve a design size of 5389 GE. A better combination of sub-fields for this S-Box-design was subsequently published by [8]. This variant is used in most of the current

[1] For a more comprehensive list see e. g. https://www.cryptolux.org/index.php/Lightweight_Block_Ciphers.

implementations. A significant reduction in area was achieved by [12], who report 3400 GE. The main drawback is the high latency of more than 1000 clock cycles. Improved runtime was reported by [16], who realised the minimum runtime for an 8-bit serial architecture and [21] but for an encryption-only design. The latter design has been extended into an encryption/decryption design by [3,4]. The additional improvement in area is due to a careful selection of the used cell library. The same holds for the design of [20], that was tailored to an Intel 22 nm tri-gate process. While encryption is implemented in less than 2000 GE, no distinct combined implementation, that offers encryption and decryption functionality, is available and the naïve combined implementation weighs in at more than 4000 GE.

Here we discuss in detail the 8-bit serial hardware architecture for encryption and decryption first presented in [27] that is significantly smaller than previously published architectures at the cost of an increase in latency. We show that by carefully designing the datapath we can construct a smaller architecture than by simply extending the architecture of [21] to accommodate decryption as in [4]. Especially we demonstrate that the inverse keyschedule can be realised without significantly increasing the circuit size, something left as an open question in [4]. Beyond that we show how the serial approach to computing the S-Box from [26] integrates nicely into our architecture, giving a further significant reduction in the size of the overall implementation.

The remainder of this chapter is organised in the following way: In Sect. 2 we quickly recapitulate the particulars of AES, before we give the details of our architecture in Sect. 3 through Sect. 6. Section 7 demonstrates how a serial multicycle S-Box-implementation can be used with our architecture, before Sect. 8 presents details of actual implementations and lists related results from the literature. Finally we sum up our results in Sect. 9.

2 Background

2.1 The Advanced Encryption Standard

We give a minimal description of AES, focused on its smallest version, AES-128, to keep the chapter somewhat self-contained. For a much better and more complete introduction we suggest the relevant chapter in [19].

AES is a block cypher taking a 128-bit plaintext for encryption and a key of either 128, 192 or 256 bits. The result is a 128-bit cyphertext. The plaintext is mapped to an internal state, then transformed by iterated application of four operations, which are independent of the key-length. These operations are organised in rounds and only the number of rounds depends on the key-length. Finally the state is serialized again to produce the cyphertext. The state is usually envisioned as a square grid of 4 by 4 bytes where the plaintext/cyphertext is mapped in column-major order. For the purpose of this work we will use indexing from top left to bottom right.

Elementary Operations of AES. AES is build from four elementary operations. Three of these are linear operations, e.g. can be implemented in hardware using only `xor` gates.

ShiftRows is simply a permutation of bytes in the state. From top to bottom the rows of the state are rotated left by 0, 1, 2 and 3 positions respectively. Its purpose is to provide confusion among the columns.

MixColums operates on each of the columns independently. Indexed cyclically, each element is tripled, the double of its predecessor added and the two successors are added unchanged. For these multiplications the byte values are interpreted as elements of a certain finite field. In practice this means that doubling equals a shift left by one bit and in case of a carry out a given constant is added (by `xor`). Tripling is the same as doubling plus adding the original value. The purpose of MixColumns is to provide confusion among the rows, complementing the ShiftRows operation.

AddRoundkey combines the current roundkey with the current value of the state by a simple bit-wise `xor` operation. Its purpose is to repeatedly mix in the secret into the state.

Finally there is a single non-linear operation: *SubBytes* replaces each byte with another value, that can either be computed just-in-time by inverting the value in the aforementioned Galois Field and applying an affine transformation, or it can be taken from a lookup-table with precomputed values. Its purpose is to avoid that the cypher can be modelled as a set of linear equations that is easily solvable.

Roundkey Derivation. For each *AddRoundkey* step a new roundkey is used. The input key is used as-is as first roundkey. Each subsequent roundkey is derived from the previous one in chunks of four bytes. A new chunk is generated by adding (`xor`) the corresponding chunk of the previous roundkey with the last derived chunk. For the first chunk of each roundkey the last chunk of the previous roundkey is taken as previous chunk, but only after applying a three-step transformation on it: First the bytes are cycled by one position in direction of the smaller index, then the *SubBytes* transformation is applied to each of the bytes before a round-specific constant is added to the first byte.

Round Structure. AES-128 encryption starts with a pre-whitening step, where the input key is added as-is. Then nine full rounds, each made up of the sequence SubBytes – ShiftRows – MixColums – AddRoundkey follow. The encryption process finishes with the sequence SubBytes – ShiftRows – AddRoundkey.

Decryption works by applying the inverse operations in reverse order.

It shall be noted here that, due to the byte granularity of the operations, the order of SubBytes and ShiftRows can be swapped.

3 Basic Principles of the Architecture

3.1 Design Rationale

Studying previously published low-area implementations of AES, e.g. [21], we quickly noticed that an 8-bit data-path lends itself nicely to the byte-granularity found in AES. We also noticed that this decision leads to many multiplexers, which the authors of [21] tried to counter by using scan flip flops (scan FFs), which combine a storage element (FF) and a multiplexer in one design unit provided by the cell library used to implement the architecture. In the case of [21] a cell library by UMC was selected. In this library a scan FF uses less area than the two components would require individually, thus saving a considerable amount of area in the implementation.

A significant part of any low-area implementation of a symmetric block cypher is taken by the storage elements. Their number is determined by the algorithm itself, for AES-128 this amounts to at least 128 FFs for the state and 128 FFs for the current roundkey, for a total of 1536 GE, assuming a typical size of 6 GE per FF. To lower this fixed cost, [4] chose a cell library by STM which offers so-called multi-bit FFs, i.e. library cells that offer multiple data I/O ports, but only a single clock port. The internal design of these cells can then be optimised by the library vendor. As a result the *average* area consumption of a FF in [4] is less than 4.5 GE, as can be estimated from the numbers given in [4] and was confirmed by the authors in personal communication.

Those optimisations are specific to a certain technology and therefore not applicable to other cell libraries, e.g. those from TSMC offered to universities through the Europractice[2] program.

We therefore aimed to reduce the area consumption of our implementation through architectural decisions. The aforementioned optimisations may then be applied additionally.

An important insight from this deliberation is that the amount of area available for optimisation through architectural decisions is limited by the difference between the total area consumed by the architecture and the area consumed by the state bits. We will call this the *optimisation gap*. In this work, taking also into account that at least one S-Box is required, it amounts to less than 1000 GE, based on [4].

Our design rationale is therefore an 8-bit datapath that keeps the number of multiplexers low.

We identified two directions of data movement in the square state representation, horizontally to the left and vertically from the bottom up. To reduce multiplexers, we designed each row to rotate one byte per cycle to the left and selected one designated column to also rotate data towards the top. The vertical movement is required for loading and serialisation of the round functions. It can be avoided only at the cost of a 32-bit serial implementation or additional storage and multiplexing, both significantly increasing the required area.

[2] http://www.europractice-ic.com/libraries_TSMC.php (last accessed: 30.03.2017).

For the key we identified that movement is either along all key bytes or – in the same direction – among the last four key bytes only. For data that should not move/be updated we used activation signals generated by the control logic.

Finally we kept the architecture flexible enough to accommodate different implementation options for the SubBytes function, from single cycle implementations to pipelined/staged implementations that are required for increased operating frequency, further serialisation or countermeasures against side-channel attacks. We demonstrate this by giving implementation results for two different S-Box architectures in Sect. 8.

3.2 Area-Runtime Tradeoff

Our goal is to push the area-runtime tradeoff significantly towards smaller area, trading a reduction in area for an increase in runtime. Naturally, as we come closer to the minimal possible area, it becomes harder to make progress and the cost (runtime) increases dramatically. Due to practical constraints, there is a point where the additional control logic offsets any gains from serialisation, further reducing the available optimisation gap.

3.3 Serial MixColumns

We chose to implement MixColumns in a serial fashion to avoid duplication of logic. The basic idea of a serial implementation draws from the fact that MixColumns, when written as a matrix operating on a vector over $\mathrm{GF}(2^8)$ in Rijndael-representation, is a MDS matrix. Especially all rows/columns are just rotated variants of each other. In practice this can be exploited by keeping the function and rotating the argument. To further ease computation, we use the decomposition

$$(2,3,1,1)(a,b,c,d)^T = [(3,2,0,0) \oplus (1,1,1,1)]\,(a,b,c,d)^T$$

in the Galois field. This allows to compute the sum of all inputs ahead and only a must be buffered for the computation of the last element. This leads to a total of 6 clock cycles per column for computing the MixColumns operation. More details on serial implementations of MixColumns can be found in [1].

3.4 SubBytes Implementation

A lot of work has gone into finding small implementations of the SubBytes operation. For a long time the architecture published in [8] was considered the smallest. Smaller ones, at the cost of reduced throughput, have been published recently in [26]. We give results for the former one as our main result as it is also used in the referenced publications and therefore facilitates easier comparison of architectures. Nevertheless, as the implementation can be replaced easily with the one from [26], we also give results with a design based on an improved version of the latter and show how it can be integrated tightly into the overall architecture.

4 Our Architecture

4.1 Architecture Overview

The architecture consists of three main modules as depicted in Fig. 1: a data `state` module implementing storage and all operations exclusive to the state, a `keystate` module implementing storage for one roundkey and implementing all functions exclusive to the key update and a `control` module that schedules and synchronises all operations and provides the public interface of the architecture. Some smaller modules implementing shared resources, such as the SubBytes operation, or otherwise lending themselves to encapsulation, such as the generation of the round constants, are also connected and coordinated through the control module.

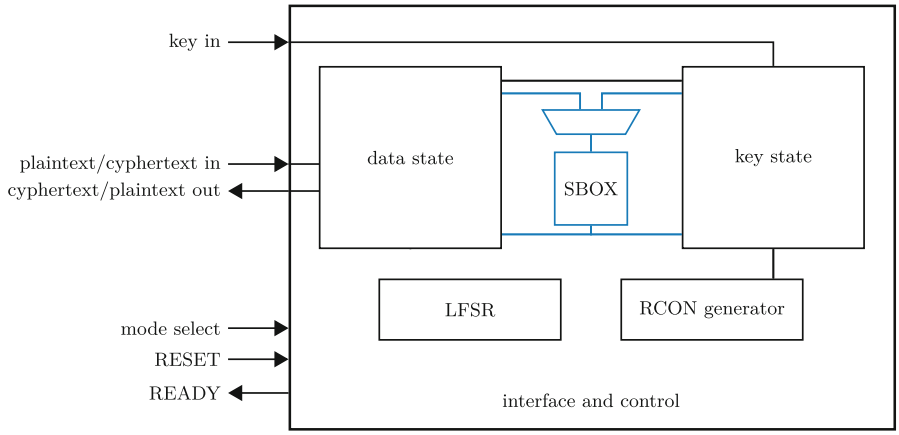

Fig. 1. High level view of the proposed architecture. The control module encapsulates the two state modules and some smaller utility modules. The public interface is described in Subsect. 4.1. The S-Box can be implemented either as a shared module or with a distinct copy for each state.

The public interface has two 8-bit wide ports for inputting data and key, one 8-bit wide port for reading the result, two input bits to select one of the available three modes of operation (see Sect. 5), an input port for the reset signal and an output port to signal availability of the results. Operation details and timings are given in Sect. 6.

4.2 Individual Modules

The **State Module** as shown in Fig. 2a contains all storage elements for the data at its current state of processing along with all logic required for the ShiftRows, MixColumns and AddRoundkey operations. It has two 8-bit wide inputs for data and key, an 8-bit wide output for data and inputs for the relevant control signals. Depending on the implementation of the SubBytes operation (dedicated or shared) another pair of data ports is present.

All data manipulation happens on the rightmost column. Once four bytes have been shifted in from the bottom or fully handled by the round operations, all columns are rotated one position to the left. The control signals can enable/disable shifting per row and individually for each of the bytes in the rightmost column. ShiftRows and its inverse are implemented by selectively activating the rows over the course of three cycles.

The term $3 \cdot [12] \oplus 2 \cdot [13]$ denotes the multiplications in the Galois Field required for a serial implementation of MixColumns. The register `ms` samples, when enabled, a `xor` sum of the rightmost column. `mb` is a register used for buffering during mix columns. Note that key addition can either be chained to MixColumns or computed individually, by virtue of the multiplexer in the input path of `mb`. To implement AddRoundkey we drew inspiration from the logic description of a multiplexer, with the difference that only the key is gated by the selection signal and combination happens by `xor` instead of `or`, hence the $\oplus$ on the multiplexer symbol.

The **Keystate Module** as shown in Fig. 2b contains all storage elements for one roundkey along with all logic required for the key update operations. As with the data state, all manipulation happens on the rightmost column (bytes 12 through 15), with the notable difference that during update the whole keystate gets shifted, since updates of the roundkey can not happen in-place. Rather, each new word is appended to the so far generated key stream. There are two enable signals, one for shifting the whole key by one position and one for only rotating the rightmost column, which is in some cases used to non-destructively read a single word of the key. By properly setting the selectors of the multiplexers, data can either cycle through the whole keystate and from byte 0 back to byte 15 or only in the rightmost column and from byte 12 to byte 15. There is again one multiplexer marked with $\oplus$. This multiplexer has distinct selection signals for each input, so either one or both at the same time can be selected. Combination happens by `xor`, so we have a dual functionality: In case only one input is selected, it is a regular multiplexer; in case both inputs are active we get the `xor` required for key updates. As with ShiftRows for the state, the rotation of the last column by one byte can be swapped with the SubBytes operation. This is handled by the multiplexer at the input of the S-Box. SubBytes is applied three times to the value then at byte 13, and after each application the whole keystate is moved by one byte. The remaining byte to be handled can then be found at byte 9. The round constant is added after the S-Box and is delivered by the control unit as required (e.g. it is zero for all but for the first byte). The updated key can either be streamed out at generation time or read back later. It was experimentally verified for our implementation that loading the key into byte 0 instead of byte 15 results in smaller area (by 3 GE) at the cost of an extra cycle.

For the reverse key schedule exactly the same dataflow can be used with the only caveat that after each word the whole key must be cycled by 8 bytes to correctly position the input values. The only additional logic required is found in the control module.

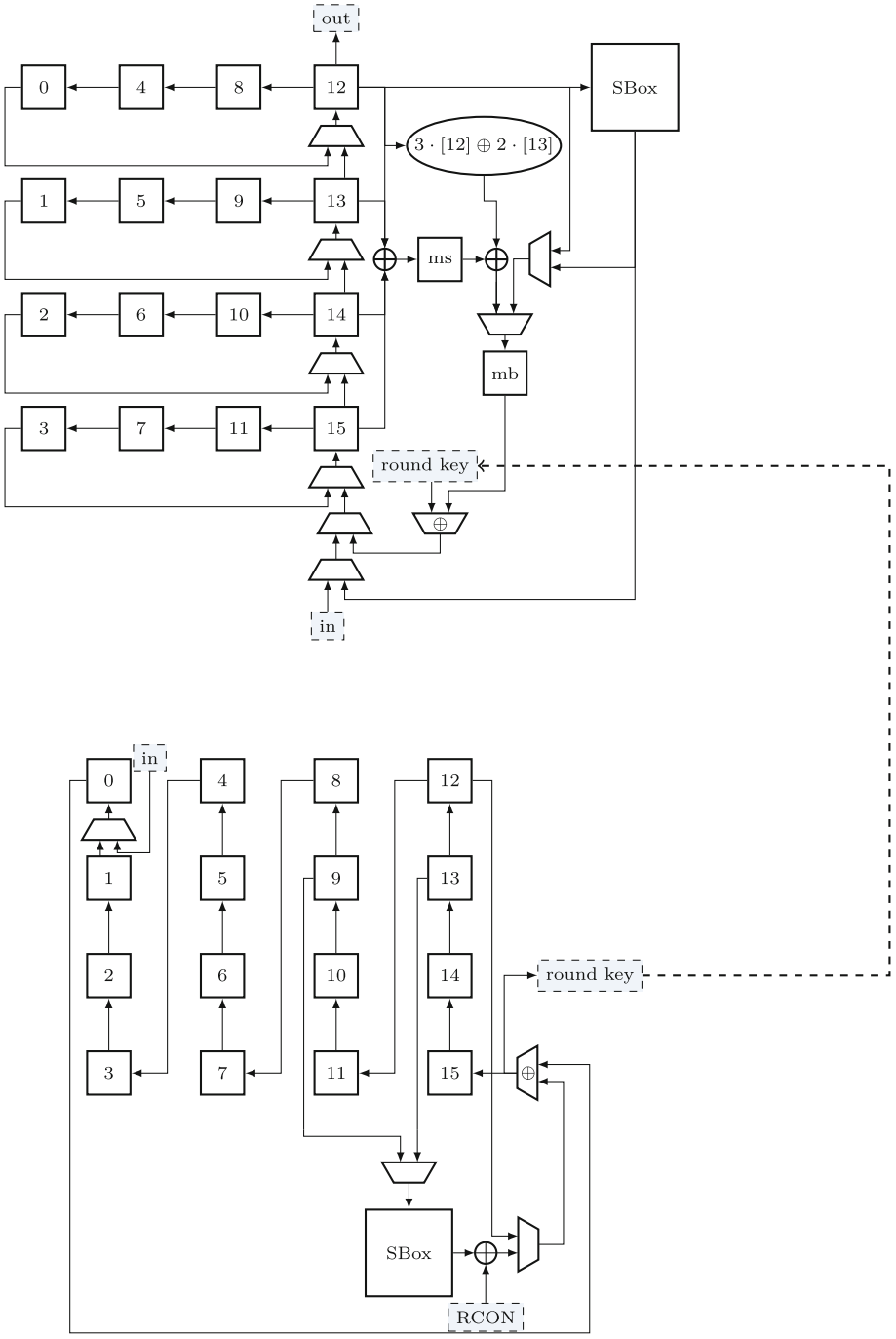

Fig. 2. (a) (upper part) The **state**. All datapaths depicted are 8-bit wide. The S-Box can be implemented as a shared module. (b) (lower part) The **keystate**. All datapaths depicted are 8-bit wide. The S-Box can be implemented as a shared module.

The **Control Module** encapsulates the two modules just presented. It also has three Linear Feedback Shift Registers (LFSRs) and a small 2-bit counter to generate the required control signals. The first LFSR is used during loading and the initial key addition until proper round-operation starts. In the later stages it is used to count the cycles required for the S-Box, which depend on the selected architecture of the latter. The second LFSR coordinates the operations for a single column while the 2-bit counter counts the number of columns. Those counters are coupled, such that a step of a "higher level" counter only occurs when a "lower level" counter wraps or resets itself. The third LFSR generates the round constants required for key scheduling. It is stepped once *during* the round and also serves as round counter for the control module. Once the correct value for the last round is reached and the last round finishes a `ready` signal is generated.

All LFSRs are in Galois configuration, with the one for the round constants being able to enumerate the constants in forward and backward direction.

Control signals are derived from these LFSRs in a straight-forward manner, taking into account the selection of operation mode, namely whether encryption or decryption should occur and in the case of the latter, whether the given key must be expanded or corresponds to the last roundkey of the encryption process and can be directly used for decryption.

Both mode selection signals can be hardwired at design time, allowing for easy synthesis of specialised architectures. The effects for our particular implementations are listed in Sect. 8.

5 Encryption vs. Decryption

The architecture supports encryption and decryption with or without key-expansion, that is for decryption either the same key as for encryption can be given, which is then first expanded before decryption starts, or the last roundkey can be given directly, allowing for flexible application of the architecture. If key-expansion is desired, a regular encryption process is run with a deactivated state. This takes more cycles than actually would be needed for a pure key-expansion, but saves significantly on control logic.

To optimise the architecture, we tried to express the decryption process as much as possible in terms of the encryption process. The inverse to ShiftRows (iSR) is given by swapping the enable signals of row 1 and 3. This is equivalent to three subsequent applications of ShiftRows (SR). As four applications rotate each row by a multiple of four positions, resulting in identity, this results in the inverse operation. Showing that no smaller number of repetitions suffices and that iSR can be expressed in at most three left shifts is trivial.

MixColumns (MC) can be written as a matrix applied to a vector. Computing powers of this matrix resembles repeated application of the MixColumns operation. The fourth power is the smallest power to result in the identity matrix. Therefore the inverse MixColumns (iMC) operation is given by three subsequent applications of MC. AddRoundkey (ARK) is self-inverse and the inverse to SubBytes (SBOX) is given by a dedicated function (iSBOX).

A third step is to align the decryption control flow to the encryption flow. Recall the sequence of operations for encryption:

$$ARK \rightarrow (SR \rightarrow SBOX \rightarrow MC \rightarrow ARK)^9 \rightarrow SR \rightarrow SBOX \rightarrow ARK$$

When reversing this sequence, ShiftRows can be swapped with the application of SubBytes, as the latter operates on isolated state bytes. Since the last round does not carry a MixColumns operation, decryption essentially has almost the same sequence of steps as encryption, with all operations replaced by their inverse counterparts and AddRoundkeys *before* inverse MixColumns:

$$\begin{aligned} &ARK \rightarrow iSR \rightarrow iSBOX \rightarrow (ARK \rightarrow iMC \rightarrow iSR \rightarrow iSBOX)^9 \rightarrow ARK \\ \hat{=}&ARK \rightarrow (iSR \rightarrow iSBOX \rightarrow ARK \rightarrow iMC)^9 \rightarrow iSR \rightarrow iSBOX \rightarrow ARK \end{aligned}$$

For encryption the individual round keys can be derived from the initial key in straightforward order. For decryption, they must be recovered from the last round key and presented to the state in reverse order. A special property of the key schedule is, that no inverse variants of the update operations are required: For each 32-bit word K_i, $i = 0, \ldots, 43$, the update formula for the forward expansion is $K_i = K_{i-4} \oplus f(K_{i-1})$, where $4 < i < 44$ and f is the special key-update transformation for every fourth column and identity otherwise. The update formula for the reverse expansion is then simply $K_i = K_{i+4} \oplus f(K_{i+3})$ with $40 > i > 0$ and f being identity, when $i \neq 0 \mod 4$.

It remains to remark that in both cases all operations aside from ShiftRows and its inverse are restricted to a single column. Thus they can be computed in an interleaved fashion and each column needs to be touched only once per round.

6 Operation and Timing

We first give a rough by-cycle breakdown of the encryption operation, followed by a short enumeration of the differences when the architecture operates in decryption mode.

To prepare the module for encryption of a new block, the `RESET` input must be activated. All inputs are expected to be available at the input ports once the `RESET` is deactivated. During the following 16 cycles data is read into the state. Simultaneously the key is loaded into the keystate. The state is subsequently rotated by one column, the key by one byte. During the next 23 cycles the key is added to the state with three cycles to switch between columns. For decryption, this is followed by another 12 cycles to reposition the key for the subsequent key scheduling.

Next, nine regular rounds follow. Each round is subdivided into four repetitions of a column update. The first iteration begins with a ShiftRows operation taking three cycles, in which the rows are gradually disabled to control the amount of shifting. In the remaining column updates, this part of the sub-round is skipped.

The main part of each sub-round starts by computing the SubBytes operation on byte 12, putting the result into byte 15 and cycling the column upwards by one byte. In the next cycle one SubByte operation for the key schedule is computed, the round constant added to the first key byte and the whole key shifted by one position, appending the result to the key state. A total of 8 cycles are needed to compute all SubByte operations for one column.

Following this, MixColumns and AddRoundkey can be computed immediately on this column. This is done in 6 cycles: First the sum of the bytes 12 through 15 is stored in the register `ms`. Concurrently, byte 12 is copied into the buffer `mb`. For five cycles the expression $3 \cdot [12] \oplus 2 \cdot [13]$ is evaluated, using hardwired shift-and-adds, the value of `ms` added and the result stored in `mb` while the previous value of this register is shifted into byte 15, with the value delivered from the roundkey module added during the last four cycles. The values in the rightmost column of the roundkey are cycled concurrently. A final cycle rotates all columns to the left by one step. Summing up, a total of $3 + 4(8 + 6) = 59$ cycles are required per round, as MixColumns is started while the last S-Box for the keyschedule is evaluated.

In the final round the input multiplexer to `mb` is set to take the value from byte 12 during the whole MixColumns process, effectively bypassing this operation and implementing a pure AddRoundkey operation.

Along with the last cycle of the last round, a ready signal is raised and the result of the computation is made available one byte per cycle at the output by shifting the last column up and rotating left when necessary. During regular computation the output is gated off to avoid unnecessary toggling at the output. This leads to a total latency of $40 + 10 \cdot 59 + 16 = 646$ cycles per encryption, including 32 cycles for loading/storing results.

The MixColumns of the dedicated encryption architecture can be tweaked a bit more for speed, yielding a slightly bigger design that runs in 606 clock cycles. Instead of computing the value for `ms` in a dedicated clock cycle, the value is produced by accumulating the outputs of the S-Box as they are produced.

Decryption has an almost identical control flow, with only a few, but important, differences. First, the output of the S-Box is written to `mb` instead of byte 15, to enable immediate addition of the roundkey. Second, MixColumn is repeated three times to realise the reverse functionality. Finally, after each column, the key must be repositioned to compute the reverse key schedule. This is done concurrently to the MixColumn operation. Altogether, decryption requires an additional 592 cycles compared to encryption for the combined architecture.

Decryption with initial forward key expansion simply runs a full encryption process with deactivated data state, followed by 9 cycles to readjust the key position and switch modes before regular decryption operation commences.

All latencies are listed in Table 1.

Table 1. Area and latency comparison of different architectures. Architectures are annotated with their capability (**E**ncryption/**D**ecryption). If different runtimes apply to combined architectures, they are given as [Encryption]/[Decryption]/[Decryption with full key expansion]. Those marked with * are generated by hardwiring the mode selection inputs of the fully featured architecture.

	Architecture	Technology	Area (GE)	Latency (cycles)
	[12] (**ED**)	Philips 350 nm	3400	1032/1165
	[16] (**E**)	0.13 µm, 1.2 V	3.1k	160
	[21] (**E**)	UMC 180 nm	2400	226
	[20] (**E**)	Intel 22 nm	1947	336
	[20] (**D**)	Intel 22 nm	2090	216
	[4] (**ED**)	STM 90 nm	2060	246/326
	[4] (**ED**)	STM 65 nm	2430	246/326
	[4], re-synth	TSMC 40 nm	2676	246/326
with S-Box from [8]	this (**ED**)	TSMC 40 nm	**2566**	689/1281/1947
	this (**D***) full key-exp	TSMC 40 nm	2569	1947
	this (**D***) rev. key-exp	TSMC 40 nm	2481	1281
	this (**E***)	TSMC 40 nm	2314	689
	this (**E**) (dedicated, small)	TSMC 40 nm	2269	646
	this (**E**) (dedicated, fast)	TSMC 40 nm	2294	606
with S-Box from [26]	this (**ED**)	TSMC 40 nm	**2449**	41419/42011/83407
	this (**D***) full key-exp	TSMC 40 nm	2444	83407
	this (**D***) rev. key-exp	TSMC 40 nm	2350	42011
	this (**E***)	TSMC 40 nm	2279	41419
	this (**E**) (dedicated)	TSMC 40 nm	2244	41326

7 An Even Smaller Variant

To demonstrate the flexibility of our architecture we also instantiate it with S-Box designs following the approach of [26]. This allows us to further reduce the size of the implementation by at least 1.1% up to more than 4.5%, depending on the implemented modes of operation and using the TSMC cell library as detailed in Sect. 8. With other cell libraries (having smaller FFs compared to the respective size of logic gates) the relative savings can be expected to be even more significant.

7.1 The Serial S-Box Architecture

The AES-S-Box is defined as an inversion of elements in a finite field with a given representation, followed by an affine mapping (defined in a different representation of this field). The basic idea of this approach is to serialize the inversion step by exploiting that every element x from the field can uniquely be written as a power g^k of a generating element g. Once the value of g has been selected

from the set of admissible values in the field, a bijection $x = g^k$ between x and k is defined. The inverse x^{-1} of x is then indeed given by $x = g^{-k}$, where the exponents are computed modulo a value determined by the field size. The different flavours of serial inversion algorithms given in [26] operate by continually multiplying with either g or its inverse element and comparing to x or a fixed reference value to obtain k resp. $-k$. The result is then computed as g^{-k} or $\left(g^{-1}\right)^k$.

Since multiplication with a fixed element in a binary extension field can be realised in hardware using a feedback shift-register in Galois configuration, this yields a very small circuit for inversion.

The implementation is based on the two algorithms given hereafter. The first one is a generalised version of Algorithm A from [26], employed to implement the S-Box used in the key scheduling. The other one is a generalised version of Algorithm B2 from the same paper, which can be tightly integrated into the data state.

Algorithm 1.1. S-Box for key scheduling

Input: γ
Output: $\gamma^{-1}\boldsymbol{c}^2$
$r_1 \leftarrow \boldsymbol{c}$;
$r_2 \leftarrow 1$;
while $(r_1 \neq \gamma) \wedge (r_2 \leq 255)$ **do**
  $r_1 \leftarrow \alpha r_1$;
  $r_2 \leftarrow r_2 + 1$;
end
$r_1 \leftarrow \boldsymbol{c}$;
while $r_2 \leq 255$ **do**
  $r_1 \leftarrow \alpha r_1$;
  $r_2 \leftarrow r_2 + 1$;
end

Algorithm 1.2. S-Box for data state

Input: γ
Output: $\gamma^{-1}\boldsymbol{c}^2$
$r_1 \leftarrow \gamma$;
$r_2 \leftarrow 1$;
while $(r_1 \neq \boldsymbol{c}) \wedge (r_2 \leq 255)$ **do**
  $r_1 \leftarrow \alpha^{-1} r_1$;
  $r_2 \leftarrow r_2 + 1$;
end

while $r_2 \leq 255$ **do**
  $r_1 \leftarrow \alpha r_1$;
  $r_2 \leftarrow r_2 + 1$;
end

Both algorithms have the same constant runtime and the counter r_2 is only required to signal the passing of this timespan. Therefore a simple maximum-length linear feedback shift-register can be used for this purpose. In the actual implementation this was chosen as an 8-bit shift register in Galois-configuration using the Reed-Solomon-Polynomial $x^8+x^4+x^3+x^2+1$ as feedback-polynomial.

Note that the algorithms run very uniformly and have almost equivalent structure. The only difference between the two is the first phase in Algorithm 1.2 running backwards in comparison to the first phase of Algorithm 1.2, which spares us resetting the register r_1 in between phases. In return the multiplication with the constant g^{-1} has to be implemented along with that for g leading to a second feedback circuit and some multiplexers.

Furthermore, the counter r_2 can be shared across multiple S-Boxes. Therefore, we chose to implement distinct S-Boxes for the data state as well as the

keystate. This also makes the multiplexers previously required to multiplex the inputs into a single instance redundant and halves the significant runtime penalty of this serial inversion approach. As this counter can also be used to generate the control signals for loading data and key as well as the control signals for returning the results, since no S-Box is active at the time, the implementation cost is further reduced. The remainder of the implementation can be kept as is without any further modifications.

The algorithms just shown realise only the inversion in the finite field. The constant α can be chosen as any of the 128 generating elements of the Rijndael-field. The constant $\boldsymbol{c}$ can be any element of its multiplicative subgroup. Choosing $\boldsymbol{c}$ to be different from the multiplicative neutral element gives an inversion result that is augmented with a multiplication by $\boldsymbol{c}^2$. The correcting computation can then be merged with the required affine transformation, which may lead to a computation that is actually less costly to implement. With other words, some computational effort can be split off from the affine transformation and be done more cheaply by encoding it in the choice of the constant.

By exhaustive search (on the standalone S-Boxes) we found that setting α to the element with the canonical bit-representation (written compactly as hexadecimal value) `46` and $\boldsymbol{c}$ to `01` (the neutral element), yielded the smallest implementation for Algorithm 1.1 for our synthesis setup (detailed in Sect. 8). In the same way we determined the choices `e9` for α and `f1` for $\boldsymbol{c}$ in the context of Algorithm 1.2.

Further area savings can be realised when the S-Boxes are integrated into the overall architecture. Note that the input to Algorithm 1.2 is not used any more once register r_1 has been initialised. This nicely matches the operation of the S-Box in the context of the full AES. Instead of attaching the S-Box to the state byte 12 as in Fig. 2a, this part of the state itself can take the role of this register and the S-Box can operate fully in place. Only the affine transformation is left in the part that is denoted as S-Box in the figure. Unfortunately this optimisation carries not over to the key schedule, where results are only concatenated instead of replaced. Therefore we chose Algorithm 1.1, which is a bit smaller as only one feedback circuit is required, as the inversion core of the S-Box here. On the other hand, this ensures the runtime difference of one cycle between the S-Boxes, so the control logic can work as in the case of a multiplexed single-cycle implementation, if it is halted during S-Box evaluation.

7.2 Runtime Impact

Using the serial S-Box architecture in the implementation adds a latency of 254 cycles per S-Box invocation in the round functions. One more cycle is added per column invocation. The reason is a timing optimisation that can only be applied in the case of the dedicated encryption core with a single-cycle S-Box. This leads to a total of $646 + 160 \cdot 254 + 40 = 41326$ cycles for the dedicated encryption core. For the combined architecture with serial S-Boxes we also decided to spend some idle cycles at the beginning of each column sub-round. This leads to a more uniform execution pattern, matching the execution times of all columns to

the first ones (that otherwise take three cycles more to execute the ShiftRows operation). We gain a reduction of 7 GE for a total runtime penalty of 90 cycles. Therefore encryption in case of the fully-functional core with serial S-Boxes takes $689 + 160 \cdot 254 + 90 = 41419$ clock cycles. In case one prefers the reduced latency, the change is easily done by uncommenting a single line in the VHDL source code.

8 Results

We implemented the architecture(s) presented in this work using VHDL. Functional verification of all implementations was done by simulating the VHDL sources directly with *GHDL* and *Synopsys VCS*. The design was compiled, optimised and mapped to a *TSMC* 40 nm *low power* cell library (`tcbn40lpbwp`) using the `compile_ultra` command in *Synopsys DesignCompiler L-2016.03-SP3*. Enabling/disabling of sequential elements was realised through clock gating. The results of manually defining clock-gating structures matched the results of automatic clock gating using the `-gate_clock` option to the `compile_ultra` command. The resulting netlist was simulated with *Synopsys VCS MX* and *Modelsim* using test vectors from the NIST KAT[3] set of test vectors.

Figure 3 shows the area consumption for different variants of our architecture and its relevant competitors, based on the *TSMC* 40 nm *low power* cell library. For each variant also the ratio between combinational and sequential logic is shown. This demonstrates that a significant part of the area is consumed by FFs. All but the pure encryption designs need forward and inverse S-Box implementations and therefore have increased area demands for the combinational part. In case of the serial S-Box variants the share of area used by FFs is even higher.

Finally Table 1 lists our results along with various low-area architectures from the literature, each being one of the smallest designs at the time of publication. A word of caution is needed on the selection of cell libraries: In the UMC 180 nm (and other UMC libraries) the area of a scan-flip-flop cell is 1 GE smaller than the joint areas of a regular flip-flop and a multiplexer cell, therefore in [21] area optimisation was possible by extensive use of scan-flip-flops. The STM 90 nm and STM 65 nm libraries used in [4] offer multibit-flip-flop cells for implementing register banks. Using these the average area consumption of flip-flops can be reduced from 6 GE/bit to roughly 4.5 GE/bit. (This was confirmed through personal communication with one of the authors.) The TSMC libraries used in this work offer none of these features. Therefore we re-synthesized the circuits from [4] based on the VHDL design files given therein. On the contrary we expect our design to significantly benefit from multi-bit cells. Extrapolation from the figures in Table 1 indicate that our design could be the first one to break the 2000 GE-barrier for a fully featured core when synthesized with the STM 90 nm library from [4].

[3] http://csrc.nist.gov/groups/STM/cavp/documents/aes/KAT_AES.zip; last accessed: 1st Feb 2017.

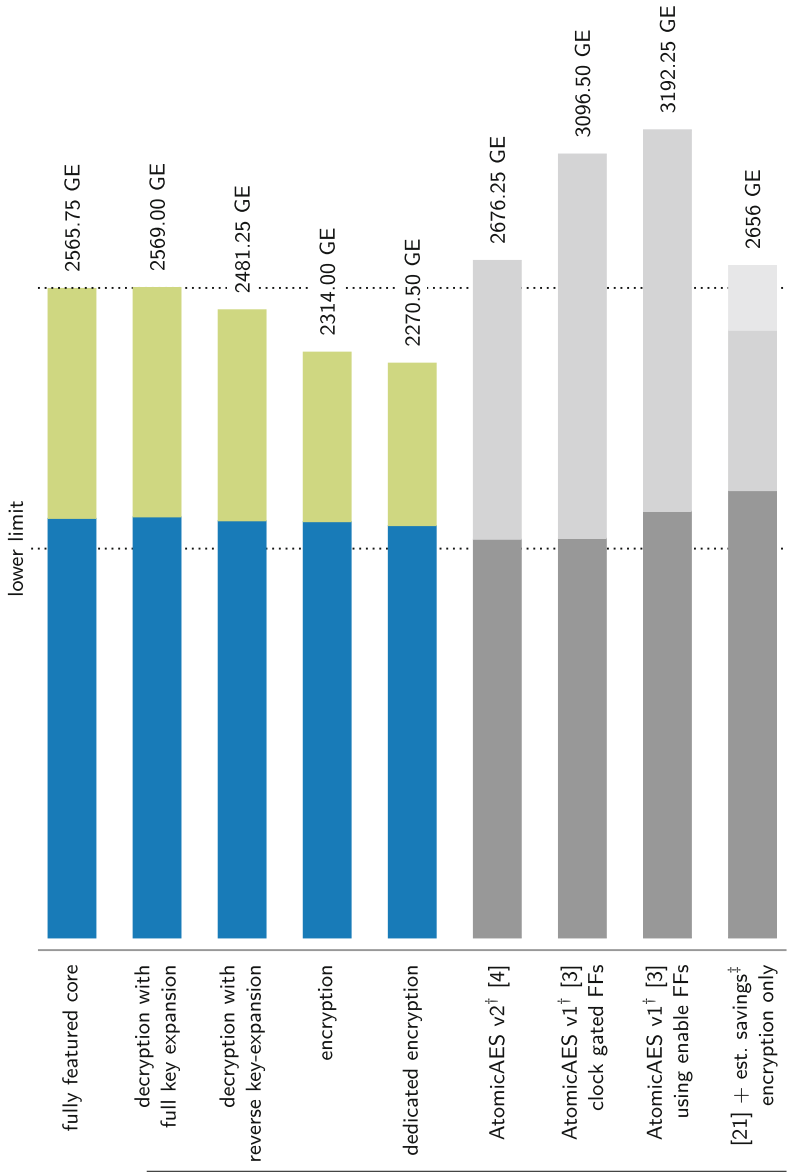

Fig. 3. Comparison of our work with its nearest competitors. The AtomicAES variants were re-synthesized from the source codes linked in [4] using our toolchain and setup. The lower part of each bar signifies the share of sequential elements in the design. Conversely the upper part represents combinational logic. To make a fair comparison we only chose the variant using the S-Box from [8] and conservatively estimated the area saved by using scan-FFs in [21] as 256 GE. The area of 1536 GE required for the state flip-flops is given as a reference lower limit for implementations.

We could not obtain detailed information for the library used in [20], for which it should be noted that the circuit in [20] was designed primarily for energy-efficiency.

Our results show that we can improve the area requirements over [4] for a fully featured AES core by 110 GE, respectively 237 GE with the serial S-Box, which amounts to at least 11%, respectively 23%, of the parts that can be optimised. With other words, everything beyond the inevitable storage for data and key is packed in 1030 GE for the fully featured core and just 734 GE for the dedicated encryption core, less then a third of the total area, when using Canright's S-Box. With the serial S-Boxes this is further reduced to 913 GE (full AES) and 708 GE (encryption only).

Furthermore, by simply fixing the mode selection inputs, optimised circuits can be generated.[4] Modifying the design by hand, also removing unneeded control signals, leads to a very compact dedicated encryption core occupying only 2269 GE, resp. 2244 GE. With the exception of [20], where a proper comparison is not possible due to the differences in the used technologies, these are by far the smallest 8-bit serial implementations of AES reported in the literature.

9 Conclusions

We presented a new 8-bit serial architecture for AES and realised a fully featured implementation and a dedicated encryption variant. Both implementations set new records for low-area consumption at a moderate increase in runtime. For each we also gave results using a different S-Box architecture which further reduced overall area consumption. This makes our architecture especially suited for scenarios where AES needs to be implemented in hardware but is not used for bulk encryption, such as SmartCards, Trusted Platform Modules (TPMs) or IoT nodes.

References

1. Ahmed, E.G., Shaaban, E., Hashem, M.: Lightweight mix columns implementation for AES. In: Proceedings of the 9th WSEAS International Conference on Applied Informatics and Communications, AIC 2009, pp. 253–258. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA (2009). http://portal.acm.org/citation.cfm?id=1628143
2. Banik, S., et al.: Midori: A block cipher for low energy (extended version). Cryptology ePrint Archive, Report 2015/1142, November 2015. http://eprint.iacr.org/2015/1142
3. Banik, S., Bogdanov, A., Regazzoni, F.: Atomic-AES: a compact implementation of the AES Encryption/Decryption core. Cryptology ePrint Archive, Report 2016/927, September 2016. http://eprint.iacr.org/2016/927

[4] The decryption circuit with full key expansion is bigger than the fully featured core, as a FF used to support optimisation in the case of hard-wiring can be removed (manually) from the latter. Keeping this register yields a size of 2573 GE.

4. Banik, S., Bogdanov, A., Regazzoni, F.: Atomic-AES v 2.0. Cryptology ePrint Archive, Report 2016/1005, October 2016. http://eprint.iacr.org/2016/1005
5. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, June 2013. http://eprint.iacr.org/2013/404
6. Bogdanov, A., et al.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74735-2_31
7. Borghoff, J., et al.: PRINCE - a low-latency block cipher for pervasive computing applications (full version). Cryptology ePrint Archive, Report 2012/529, September 2012. http://eprint.iacr.org/2012/529
8. Canright, D.: A very compact S-box for AES. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 441–455. Springer, Heidelberg (2005). https://doi.org/10.1007/11545262_32
9. Chawla, S.S., Aggarwal, S., Kamal, S., Goel, N.: FPGA implementation of an optimized 8-bit AES architecture: a masked S-box and pipelined approach. In: 2015 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), pp. 1–6. IEEE, July 2015. http://dx.doi.org/10.1109/conecct.2015.7383859
10. Daemen, J., Peeters, M., Van Assche, G., Rijmen, V.: The NOEKEON block cipher. Technical report, October 2000. http://gro.noekeon.org/Noekeon-spec.pdf
11. De Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN—a family of small and efficient hardware-oriented block ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 272–288. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04138-9_20
12. Feldhofer, M., Wolkerstorfer, J., Rijmen, V.: AES implementation on a grain of sand. In: IEE Proceedings - Information Security, vol. 152, no. 1, p. 13+ (2005). http://dx.doi.org/10.1049/ip-ifs:20055006
13. Feldhofer, M., Lemke, K., Oswald, E., Standaert, F.X., Wollinger, T., Wolkerstorfer, J.: State of the art in hardware architectures. Note: deliverable with a special focus on AES hardware architectures. ECRYPT Deliverable No. D.VAM2, September 2005. http://www.iaik.tugraz.at/content/research/krypto/AES/VAM2-IAIK-17-D.VAM2-1_0.pdf
14. Gong, Z., Nikova, S., Law, Y.W.: KLEIN: a new family of lightweight block ciphers. In: Juels, A., Paar, C. (eds.) RFIDSec 2011. LNCS, vol. 7055, pp. 1–18. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-25286-0_1
15. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The LED block cipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 326–341. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23951-9_22
16. Hämäläinen, P., Alho, T., Hännikäinen, M., Hämäläinen, T.D.: Design and implementation of low-area and low-power AES encryption hardware core. In: 9th EUROMICRO Conference on Digital System Design (DSD 2006), pp. 577–583. IEEE (2006). http://dx.doi.org/10.1109/dsd.2006.40
17. ISO/IEC: ISO/IEC 29192–2:2012 - information technology - security techniques - lightweight cryptography - part 2: Block ciphers. Technical report, International Organization for Standardization, January 2012. https://www.iso.org/standard/56552.html
18. Knudsen, L., Leander, G., Poschmann, A., Robshaw, M.J.B.: PRINTcipher: a block cipher for IC-printing. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 16–32. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15031-9_2

19. Knudsen, L.R., Robshaw, M.: The Block Cipher Companion. Springer, Heidelberg (2011). http://link.springer.com/book/10.1007%2F978-3-642-17342-4
20. Mathew, S., et al.: 340 mV-1.1 V, 289 Gbps/W, 2090-gate nanoAES hardware accelerator with area-optimized encrypt/decrypt $GF(2^4)^2$ polynomials in 22 nm tri-gate CMOS. IEEE J. Solid-State Circ. **50**(4), 1048–1058 (2015). http://dx.doi.org/10.1109/jssc.2014.2384039
21. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: a very compact and a threshold implementation of AES. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 69–88. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20465-4_6
22. Pramstaller, N., Mangard, S., Dominikus, S., Wolkerstorfer, J.: Efficient AES implementations on ASICs and FPGAs. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) AES 2004. LNCS, vol. 3373, pp. 98–112. Springer, Heidelberg (2005). https://doi.org/10.1007/11506447_9
23. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact Rijndael hardware architecture with S-box optimization. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 239–254. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45682-1_15
24. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: *Piccolo*: an ultra-lightweight blockcipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 342–357. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23951-9_23
25. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: *TWINE*: a lightweight block cipher for multiple platforms. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 339–354. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35999-6_22
26. Wamser, M.S.: Ultra-small designs for inversion-based S-boxes. In: 17th Euromicro Conference on Digital System Design, pp. 512–519. Department of Computer Science, Università di Verona. IEEE, August 2014. http://dx.doi.org/10.1109/DSD.2014.37
27. Wamser, M.S., Sigl, G.: Pushing the limits further: sub-atomic AES. In: 2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), pp. 1–6 (2017). http://dx.doi.org/10.1109/VLSI-SoC.2017.8203470
28. Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: RECTANGLE: a bit-slice Ultra-Lightweight block cipher suitable for multiple platforms. Cryptology ePrint Archive, Report 2014/084, February 2014. http://eprint.iacr.org/2014/084