Putting Together What Fits Together - GrÆStl

Markus Pelnar[1,2], Michael Muehlberghuber[1], and Michael Hutter[2]

[1] Integrated Systems Laboratory (IIS), ETH Zurich,
Gloriastrasse 35, 8092 Zurich, Switzerland
m.pelnar@student.tugraz.at, mbgh@iis.ee.ethz.ch

[2] Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
michael.hutter@iaik.tugraz.at

Abstract. We present GrÆStl, a combined hardware architecture for the Advanced Encryption Standard (AES) and Grøstl, one of the final round candidates of the SHA-3 hash competition. GrÆStl has been designed for low-resource devices implementing AES-128 (encryption and decryption) as well as Grøstl-256 (tweaked version). We applied several resource-sharing optimizations and based our design on an 8/16-bit datapath. As a feature, we aim for high flexibility by targeting both ASIC and FPGA platforms and do not include technology or platform-dependent components such as RAM macros, DSPs, or Block RAMs. Our ASIC implementation (fabricated in a 0.18 μm CMOS process) needs only 16.5 kGEs and requires 742/1,025 clock cycles for encryption/decryption and 3,093 clock cycles for hashing one message block. On a Xilinx Spartan-3 FPGA, our design requires 956 logic slices and 302 logic slices on a Xilinx Virtex-6. Both stand-alone implementations of AES and Grøstl outperform existing FPGA solutions regarding low-area design by needing 79 % and 50 % less resources as compared to existing work. GrÆStl is the first combined AES and Grøstl implementation that has been fabricated as an ASIC.

Keywords: Hardware implementation, AES, Grøstl, ASIC, FPGA, embedded systems, low-resource design.

1 Introduction

Among the most commonly used cryptographic primitives in classical communication protocols are block ciphers and hash functions. The Advanced Encryption Standard (AES) [26] is by far the most widespread block cipher since its standardization in 2001. Grøstl [8] was one of the final round candidates of the SHA-3 cryptographic hash function competition from which Keccak [1] emerged as winner. Both AES and Grøstl share several similarities such as a common S-box or similar diffusion layers which encourage the implementation of a combined hardware architecture.

In this paper, we describe a hardware architecture—called GrÆStl—that combines the functionality of AES-128 and Grøstl-256 in one piece of silicon.

S. Mangard (Ed.): CARDIS 2012, LNCS 7771, pp. 173–187, 2013.
The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-642-37288-9_12

Our design aims for high flexibility supporting both Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) platforms without using technology dependent components such as Random Access Memory (RAM) macros, Digital Signal Processors (DSPs), or Block RAMs. We exploit various optimization techniques to reduce the required area, for example, by sharing registers and a common datapath. The ASIC version of our design has been fabricated using the 0.18 μm Complementary Metal Oxide Semiconductor (CMOS) process technology from the United Microelectronics Corporation (UMC) and therefore represents the first tape-out version of a combined AES/Grøstl architecture in the literature. It requires only 16.5 kGEs in total and needs 742/1,025 clock cycles for AES encryption/decryption and 3,093 clock cycles for hashing. The small area requirements and also the low-power consumption of about 20 μW at 100 kHz make the design applicable to resource-constrained devices such as smart cards or contact-less powered devices. We also compared the results of our stand-alone implementations of AES and Grøstl. It shows that the implementations outperform existing FPGA solutions in terms of low-area. They require up to 79 % less resources on a Spartan-3 as compared to existing implementations.

The remainder of this paper is organized as follows. In Section 2, an overview about related work on area-constrained AES and Grøstl implementations is given. Section 3 presents the architecture of GrÆStl and describes the design starting from the top module down to the implemented (combined) datapath. In Section 4, we present our results and compare them with related work. Finally, we summarize our results and draw conclusions in Section 5.

2 Related Work

There exist many papers in the literature that present low-resource hardware implementations of AES or Grøstl. Since publication of the algorithms, several optimization techniques have been proposed that reduce the area requirements for both ASIC and FPGA platforms. One example is the optimized AES S-box implementation of Canright [3] that has been also used by Feldhofer et al. [6] to realize a very compact version of AES-128 in 2005. Their implementation requires 3.4 kGEs for both encryption and decryption. Similar results have been reported by Hämäläinen et al. [11], Kaps et al. [19], and Kim et al. [23] who reported about 4 kGEs in total. At EUROCRYPT 2011, Moradi et al. [24] presented an area-optimized implementation of (encryption-only) AES which needs about 2.4 kGEs that marks the lowest level of state-of-the-art AES implementations.

As opposed to AES, there exist only a few publications so far that describe low-area optimizations for Grøstl on ASIC devices. Tillich et al. [28] have been the first who presented an implementation requiring 14.6 kGEs. This number also corresponds well to the area estimations given in the Grøstl specification from Gauravaram et al. [8] which reported a size of less than 15 kGEs. Further implementations have been presented by Katashita et al. [21], Guo et al. [10], and Henzen et al. [12] which require between 34.8 and 72 kGEs.

In view of FPGA platforms, large effort has been made to reduce the complexity by reusing existing hardware components, e.g., Block RAMs and DSPs. Chodowiec and Gaj [5] presented a low-resource AES-128 implementation on FPGAs in 2003 and described different optimization techniques. Their implementation needs 222 slices and 3 Block RAMs on a Spartan-2 supporting both encryption and decryption. In particular, they made use of Look-Up Tables (LUTs) to efficiently implement shift registers such that intermediate values can be easily shifted without generating additional address logic. In the upcoming years, improvements have been reported by, e.g., Good et al. [9], Chi-Wu et al. [13], and Bulens et al. [2].

A very compact FPGA implementation of Grøstl has been presented by Jungk et al. [18,16,17] in 2010, 2011, and 2012. Their design needs 967 slices on a Spartan-3 FPGA without interface and 328 slices on a Virtex-6 FPGA requiring no Block RAMs. Sharif et al. [27] reported 1,627 slices (w/o Block RAMs) and 1,141 slices (using 18 Block RAMs) on a Virtex-5. In the same year, Kerckhof et al. [22] presented an implementation that needs only 343 slices on a Spartan-6 and 260 slices on a Virtex-6 FPGA (w/o using any Block RAMs or DSPs). At the final SHA-3 conference, Kaps et al. [20] published a lightweight Grøstl implementation requiring approx. 560 slices and one Block RAM on a Spartan-3 FPGA.

While there exist several papers that analyze the combination of different block ciphers and hash functions (mostly combining MD5 with SHA-1, e.g., [4,7,15,29]), there exists only one publication that focuses on the combination of AES and Grøstl on FPGA platforms. Järvinen [14] analyzed various resource-sharing techniques to reduce the area requirements for an Altera Cyclone III. Their smallest design needs 12,387 Logic Cells (LCs) whereas AES takes an overhead of about 2.5 %, i.e., 300 LCs.

3 Hardware Architecture of GrÆStl

GrÆStl has been designed with the aim for a very compact solution that supports both AES-128 and Grøstl-256 in one piece of silicon. We targeted a low-resource design (primarily area and power optimized) and applied different design techniques that are new or already known, e.g., from existing low-area AES implementations. Low-resource architectures are especially interesting for embedded systems such as contactless smart cards, sensor nodes, and RFID-based devices where area and power are stringent requirements for a practical deployment.

Next to the low-resource requirements, we aim for high flexibility so that the design can be applied on both ASIC and FPGA platforms. We therefore avoid the use of process-dependent technologies like RAM macros or the use of Block RAMs or DSPs on FPGA architectures. The disadvantage of these technologies are that they might not be available in all CMOS libraries and that they have to be recreated in case of a possible CMOS-process variation. Moreover, in case of FPGAs, resources such as Block RAMs or DSPs might be already used by other system components such that the applicability of the design depends on

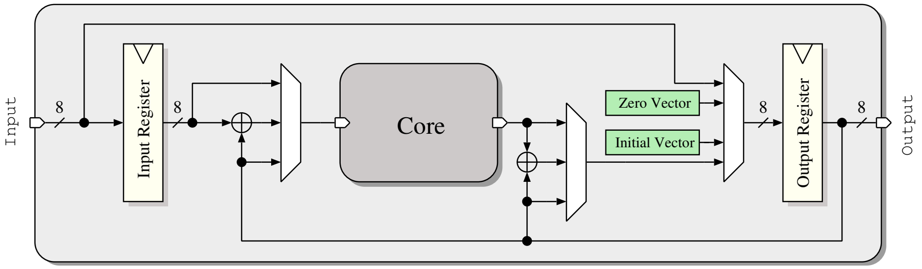


Fig. 1. Architecture of our 8-bit GrÆStl implementation

their availability. In order to avoid those dependencies, we based our design only on standard cells and generic hardware components which makes it very flexible and portable to other platforms.

Since Grøstl needs per se more resources than AES, cf. [8], it is advisable to reuse existing hardware components to keep the overhead for AES as low as possible. We therefore decided to implement a combined datapath that effectively shares resources such as the S-box, temporary registers, or the *State* matrix. We based the top-module design and also the interface on an 8-bit architecture because it reduces the area complexity and power consumption of our design compared to larger datawidths. For the common datapath, we used 16 bits for Grøstl and only 8 bits for AES. This has two advantages. First, since two S-boxes are required for Grøstl, one S-box can be used in parallel to the SubBytes operation of AES to improve the performance of round-key generation. Second, parts of the computation can be switched off in order to reduce the overall power consumption.

An overview of the GrÆStl architecture is shown in Fig. 1. The main components are a common datapath (denoted as *Core* unit) that combines most of the round transformations for AES and Grøstl and two shared 512-bit shift registers. In order to keep the area requirements low, we decided to calculate the P and Q permutations of Grøstl sequentially instead of calculating them in parallel. This reduces the performance of hashing but allows to implement only one shared permutation instance in hardware. The need of two additional 512-bit shift registers is compensated by the fact that they are needed anyway in order to store the original message (needed for the second permutation Q), the result of the first permutation P and intermediate hash values in case of messages not fitting one block. These shift registers feature an 8-bit I/O through which large de-/multiplexers can be avoided. In the following, we present the common datapath and describe the implemented optimization techniques. Afterwards, we make use of the shared registers to improve the performance of the AES round-key generation.

3.1 The Common Datapath

Figure 2 shows the architecture of the common datapath. It has been separated into four main components according to the round transformations of AES and

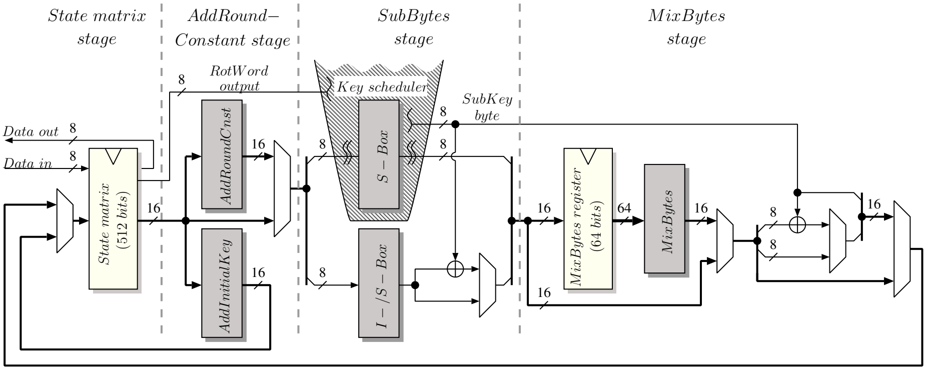


Fig. 2. Core datapath w/o round-key generation and un-/loading of the State matrix

Grøstl: a shared State, an AddRoundConstant/AddInitialKey, SubBytes, and MixBytes unit.

Sharing the State. One of the most obvious ways to share resources between AES and Grøstl is to share the memory resources for the State. The size of the State for AES is 128 bits, i.e., a 4 × 4-byte matrix. Grøstl, in contrast, needs 512 bits (for variants returning a message digest of a size up to 256 bits), i.e., an 8 × 8-byte matrix. Thus, up to four AES States can fit into one Grøstl State which allows to integrate up to four AES encryption/decryption units in parallel to speed up the computation with minimal overhead. The work of Järvinen [14], for example, reported such an integration with an additional overhead of 13.5 % for four parallel AES encryptions and only 2.5 % for one AES encryption (neither of these two architectures contain the key generation). In contrast to our implementation, Järvinen targeted a high-throughput architecture with a datapath width of 512 bits.

For the design of GrÆStl, we decided to avoid any parallel computations to keep the area requirements as low as possible. Thus, we mapped one AES structure into the Grøstl State as illustrated in Fig. 3, i.e., the data (requiring the upper left 4 × 4 bytes) and the round key (requiring the lower left 4 × 4 bytes). In addition to these memory locations, we reused four bytes of the upper right 4 × 4 matrix as temporary registers for the round-key generation, further on denoted as *RotWord* shift-register. The round keys are then generated "on-the-fly" during AES computation.

The common State has been implemented using shift registers. This has several advantages. First, they reduce the area requirements on common FPGA platforms since the Look-Up Table (LUT) in certain logic blocks can be configured as a shift register without using the flip-flops available in each slice as also noticed by Chodowiec et al. [5]. Second, they are very flexible and can be used for both ASIC and FPGA designs as opposed to other memory architectures such as RAM macros or Block RAMs. Third, due to automatic shifts of intermediate

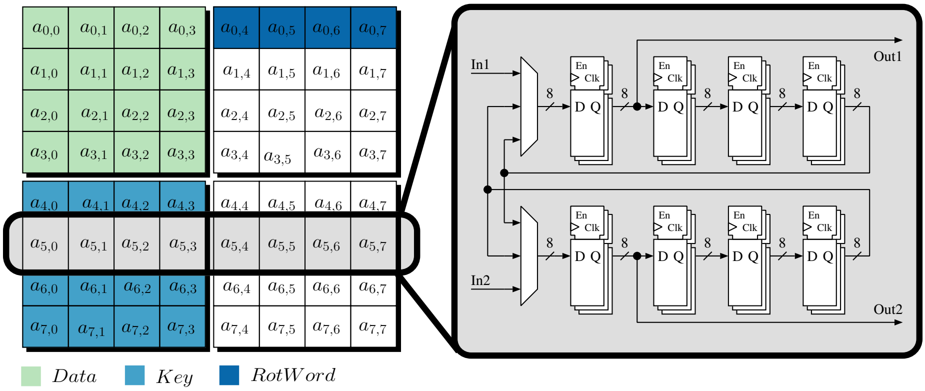

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$ $a_{0,6}$ $a_{0,7}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{1,5}$ $a_{1,6}$ $a_{1,7}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$ $a_{2,6}$ $a_{2,7}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$ $a_{3,6}$ $a_{3,7}$
$a_{4,0}$ $a_{4,1}$ $a_{4,2}$ $a_{4,3}$ $a_{4,4}$ $a_{4,5}$ $a_{4,6}$ $a_{4,7}$
$a_{5,0}$ $a_{5,1}$ $a_{5,2}$ $a_{5,3}$ $a_{5,4}$ $a_{5,5}$ $a_{5,6}$ $a_{5,7}$
$a_{6,0}$ $a_{6,1}$ $a_{6,2}$ $a_{6,3}$ $a_{6,4}$ $a_{6,5}$ $a_{6,6}$ $a_{6,7}$
$a_{7,0}$ $a_{7,1}$ $a_{7,2}$ $a_{7,3}$ $a_{7,4}$ $a_{7,5}$ $a_{7,6}$ $a_{7,7}$
Data
Key
RotWord
In1
In2
Out1
Out2
En
Clk
D Q
8

Fig. 3. Mapping the AES State into the Grøstl State and the construction of a single State matrix row

values, additional address logic and multiplexer stages can be avoided. Thus, no ShiftRows or ShiftBytes units are needed because they are implicitly performed by the applied shift registers.

Each row in the State has been implemented as an 8-byte shift register that can be split into two 4-byte shift registers with two independent inputs. Figure 3 shows one internal row composed of two 4-byte shift registers. When AES is performed, only 4-byte shift registers are used, 8-byte shift registers are used during Grøstl computations. In order to reduce the power consumption during AES computation, we applied an operand-isolation technique that switches off unused parts of the matrices, e.g., the lower right 4×4 bytes of the State matrix. Furthermore, we applied clock gating cells to minimize toggling activity.

Tweaked AddRoundConstant Stage. For Grøstl, the State is modified through the AddRoundConstant function, which varies with the type of the required permutation. In case of the P permutation, the first row gets modified through fixed constants whereas the rest stays untouched. This changes for the Q permutation, where instead of the first row, the last row gets modified. Additionally, the rest of the State bits get flipped, cf. [8]. Note that in contrast to the Grøstl-0 specification (from the 31th of October 2008), where only a single byte gets modified (for both permutations), in the tweaked Grøstl version (from the 2nd of March 2011, version 2.0.1) multiple bytes get modified. Compared to existing work, which mostly presents solutions for Grøstl-0, e.g., in [28], we present an implementation that considers the tweaked variant including the modified round constants and initial vectors.

For AES, the State gets modified through the AddRoundKey function, which simply adds (XOR-operation) the round key located in the lower left 4×4 bytes of the State matrix to the data in the upper left 4×4 bytes.

Reusing SubBytes for AES Round-Key Generation. The SubBytes transformation in Grøstl can be efficiently combined with the S-box operation of AES, because both algorithms make use of the same S-box transformation. Minor effort has to be made in order to provide the inverse S-box transformation required for AES decryption.

There exist several implementation optimizations for the AES S-box, e.g., given in [3], [25], or [30]. Most of the related work transformed the finite-field operations over $GF(2^8)$ into a composite of smaller fields, e.g., $GF((2^4)^2)$. We implemented the method proposed by Wolkerstorfer et al. [30], where an S-box is composed of two transformations, namely the calculation of a multiplicative inverse in the finite field $GF(2^8)$ and an affine transformation. For AES decryption, the affine transformation is exchanged with its counterpart and executed before computing the multiplicative inverse. In order to save some additional gates, one may replace the S-box implementation by Wolkerstorfer et al. with the one by Canright [3]. As our Grøstl version is based on a 16-bit wide datapath, we implemented two S-boxes, one of them providing both transformation directions. AES is based on an 8-bit datapath, thus we exploited the presence of an additional S-box in order to improve the performance of the AES round-key generation as described in the following.

Basically, there exist two possibilities to generate the round keys for AES encryption and decryption. First, the round keys are pre-computed and stored in non-volatile memory. Second, the round keys are computed "on-the-fly". While the first option provides fast access to existing round keys, the second option is cheaper in terms of area requirements since no memory is needed to store the keys. We therefore decided to implement the second option.

While one S-box is used to perform the SubBytes operation of AES, the second S-box can be reused to calculate the round keys in parallel. For the round-key generation, one S-box, XOR operations, and a small LUT is required that holds the round constants. Figure 6 and Fig. 7 in Appendix A illustrate the general forward and backward round-key generation.

The forward round-key generation is done as follows. First, during the initialization of the common State, the last four bytes of the master key are loaded into the *RotWord* shift-register (located in the upper right 4 × 4 matrix as shown in Fig. 3). The output of the *RotWord* shift-register gets substituted by the shared S-box and modified with the round-dependent constant *Rcon* before it gets added to the output of the first row of the key matrix (located in the lower left 4 × 4 matrix as shown in Fig. 3). Afterwards, the result is loaded back into the *RotWord* shift-register and the first row of the key matrix before both get shifted. This is done for the first byte of each row of the key matrix in order to obtain the highest four bytes of the next round key. The following three columns of the next round key get calculated by applying the same procedure while bypassing the S-box and *Rcon* modifications. Due to the similarity of the forward and backward round-key generation, we do not explain the latter in detail.

Combined MixColumns and MixBytes. MixColumns and MixBytes have been combined to a common MixBytes function. It has been implemented such that it takes the 64-bit output of the *MixBytesReg* as input, representing either a column of AES or a column of Grøstl. As for AES one column only has a size of 32 bits, the remaining 32 bits are not used. For both AES and Grøstl the matrices, which have to be multiplied with the State, are circulant and constant. This fact helps in reducing the complexity by implementing from each of the three matrices only one row. The circulant behavior is gained through the *MixBytesReg*, realized as an 8-byte shift register. In the next four clock cycles after this register has been loaded, always 8 bits for AES respectively 16 bits for Grøstl are processed and stored in the State matrix. Note that the MixColumns operation for AES encryption comes for free as it is implicitly computed during a Grøstl MixBytes operation.

Furthermore, the MixColumns operation for AES decryption takes larger coefficients than those required for Grøstl. Therefore, additional effort has been made to provide also the inverse operation of the MixColumns function.

3.2 I/O-Register Sharing

Since two 512-bit registers are required for our Grøstl implementation in order to hold intermediate values and the original input message during the sequential execution of permutation P and Q, we can reuse them to improve the performance of AES decryption as follows. As soon as an AES encryption or decryption finishes, we store its master key together with the corresponding last round key in the output register. If afterwards another decryption takes place, we compare its master key with the previously stored one. In case the keys match, we reuse the previously stored last round key instead of the new master key and can therefore save the time (i.e., 330 clock cycles) for calculating the last round key required for decryption.

4 Results

We implemented GrÆStl-256 in VHDL and synthesized it for both ASIC and FPGA platforms. For the ASIC design, we used *Mentor Graphics ModelSim 6.5c* and *Synopsys DesignCompiler 2010.03* for functional simulations (RTL and post-layout verification) and synthesis. We mapped our architecture onto a standard-cell library based on the 0.18 μm CMOS process by UMC. One single Gate Equivalent (GE) therefore corresponds to the area of a 2-input NAND gate, i.e., 9.3744 μm^2. For FPGA synthesis, we used *Xilinx ISE Design Suite 12.1* and applied the parameter set “Area Reduction with Physical Synthesis”. Furthermore, we removed the reset feature from all shift registers as sub-optimal reset strategies can prevent the use of device-library components, such as shift register look-up tables (SRLs) [31].

In order to evaluate the efficiency of GrÆStl compared to separate implementations, we provide results for stand-alone variants of AES and Grøstl. However, we have to note that a fair comparison of our stand-alone variants with related

Fig. 4. ASIC area results after synthesis

Component	**AES** [GE]	**Grøstl** [GE]	**GrÆStl** [GE]
Top Level Glue Logic	50	460	800
Input/Output Reg.	-	8,250	8,250
Core	2,550	6,340	7,500
State Matrix	1,100	4,200	4,350
AddRndCnst.	-	100	100
SubBytes	330	540	600
MixBytes	490	900	1,100
Core Glue Logic	630	600	1,350
Round-Key Gen.	2,150	-	-
Overall	4,750	15,050	16,550

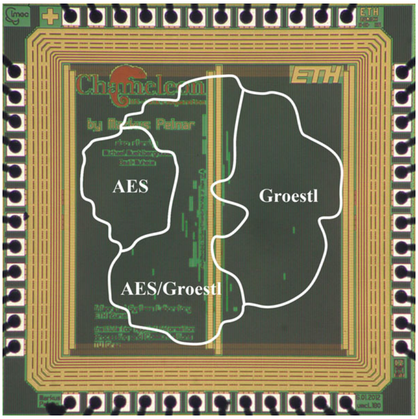


Fig. 5. Chip layout

work is largely infeasible since we targeted a combined version and integrated optimization techniques that do not affect the single-implementation variants. Table 4 lists the area occupation for AES, Grøstl, and GrÆStl for a target frequency of 100 MHz. It shows that GrÆStl needs only 16,550 GEs of area not including an interface. This is about 16 % less than the sum of the two separate implementations. Furthermore, it shows that the overhead for AES is only 1.5 kGEs which is smaller than the smallest available stand-alone AES implementation given in the literature, cf. [6,11].

In view of execution time, our stand-alone version of AES and GrÆStl need 945 clock cycles for encryption. This includes an 8-bit four-phase-handshaking that requires 203 clock cycles to load/unload the input/output register. Decryption needs 1,558 clock cycles. Our GrÆStl implementation even reduces the number of clock cycles by 330 because the round-key generation can be avoided if the last round key is already maintained in the I/O registers. Hashing of a 512-bit block takes 3,465 clock cycles (including 404 clock cycles for the interface) without applying message padding. The number of clock cycles results from the State matrix design, the datapath width of 8 bits and the required *MixBytesReg* through which the datapath is split into halves. In short, this register gets filled in each cycle with one valid byte related to a certain column of AES or Grøstl. After finishing loading, the State matrix gets updated through the MixBytes function. Afterwards the State matrix is shifted by one column and the procedure starts over again.

We taped out our design as an ASIC. For this, we targeted a maximum clock frequency of 125 MHz. This in combination with clock gating and eight scanchains required an additional amount of 750 GEs of chip area. A photo of the fabricated chip, highlighting the floorplan of the three different modules, is illustrated in Fig. 5. To the best of our knowledge, there is no ASIC design of Grøstl (or a combination of AES and Grøstl) available so far, which targets a low-resource implementation and has finally been taped out.

Table 1. ASIC comparison of AES-128 (incl. decryption) and Grøstl-256

Source	**Width**	**Techn.**	**f$_{\text{max}}$**	**Cycles/Block**			**Power**	**Area**	
	[bits]	[μm]	[MHz]	Enc.	Dec.	Hash	[$\mu W/MHz$]	[kGE]	
Hämäläinen [11]	8	0.13	153	160	n/a	-	37 @ 1.2 V	3.9[1]	AES
Feldhofer [6]	8	0.35	80	1,032	1,165	-	45 @ 1.5 V	3.4	
Ours - AES	8	0.18	100	742	1,025	-	130 @ 1.8 V	4.75	
Tillich [28]	64	0.35	56	-	-	196	2,210 @ 3.3 V	14.6	Grøstl
Ours - Grøstl	16	0.18	100	-	-	3,093	200 @ 1.8 V	15.05	
Ours - GrÆStl	8/16	0.18	100	742	1,025	3,093	200 @ 1.8 V	16.55	GrÆStl

[1] The area for the design, including the decryption has been estimated with additional 25 % of the original (encryption only) AES design.

Table 1 gives a comparison with related work. While our stand-alone implementation of AES is about 1 kGE larger than existing work [6,11], only a small overhead is required for Grøstl to support also AES, i.e., 10 %. Moreover, our GrÆStl implementation is only slightly larger than the work by Tillich et al. [28] which can be attributed to the design decisions to support both Grøstl (tweaked version instead of Grøstl-0) and AES which requires additional area to optimally merge both algorithms (to keep the overall area as low as possible). Regarding power consumption, it shows that our design meets most requirements even for a contactless operation, e.g., for contactless smart cards. However, due to different fabrication technologies and supply voltages, we cannot fairly compare the given numbers.

4.1 FPGA Results

We used Xilinx Spartan-3 and Virtex-6 FPGAs to evaluate the performance of our design on reconfigurable hardware. Table 2 shows the results after place-and-route and without an interface included. On the Spartan-3, our design needs only 956 slices. On the Virtex-6, 302 slices are needed. The sum of slices required by the independent Spartan-3 AES/Grøstl implementations is smaller than those required for the GrÆStl design. This is due to the structure of the State matrix of the inner 8/16-bit datapath. For the Grøstl design, this State matrix is built upon eight 8-byte shift registers which perfectly fit the SRL-16 mode (provided by Xilinx) as such a 8-byte shift register can be achieved within only a single CLB. For GrÆStl, each row of the State matrix is replaced by two independent 4-byte shift registers since AES operates on a smaller State compared to Grøstl. Because of the 4-byte shift registers and the control logic required to form a 8-byte shift register, it is not possible for the compiler to fit the two 4-byte shift registers into one CLB. Particular changes to the design, which target a certain FPGA device (i.e., a more *target-oriented mapping* of the shift registers) may lead to better results for all three designs. However, this would lead to a platform-dependent architecture what would contradict with one of our main goals, i.e., to stay platform independent.

Table 2. Post place-and-route results for various Xilinx FPGAs

	Spartan-3			**Virtex-6**		
	AES	**Grøstl**	**GrÆStl**	**AES**	**Grøstl**	**GrÆStl**
Number of slice LUTs	838	896	1,805	455	531	1,046
Number used as logic.	788	743	1,554	419	443	927
Number used as shift registers	48	128	192	24	64	96
Number used as a route-thru	2	25	59	12	24	23
Number of slice registers	48	293	407	169	279	360
Frequency(MHz)	35	40	30	110	115	80
Number of occupied slices	442	488	956	142	202	302

Table 3. FPGA comparison of AES-128 (incl. decryption) and Grøstl-256

Source	**Width**	**Output**	**Device**	$\mathbf{f_{max}}$	**Cyc./Block**			**Area**	
	[bits]	[bits]		[MHz]	Enc.	Dec.	Hash	[Slices]	
Bulens [2]	128	128	Spartan-3	150	12	12	-	2,150	AES
Ours - AES	8	128	Spartan-3	35	742	1,025[1]	-	442	
Bulens [2]	128	128	Virtex-5	350	11	11	-	550	
Ours - AES	8	128	Virtex-6	110	742	1,025[1]	-	142	
Jungk [16]	64	224/256	Spartan-3	182	-	-	160	967	Grøstl
Ours - Grøstl	8/16	256	Spartan-3	40	-	-	3,093	488	
Kerckh. [22]	64	256	Virtex-6	280	-	-	450	260	
Ours - Grøstl	8/16	256	Virtex-6	115	-	-	3,093	202	
Ours - GrÆStl	8/16	256	Spartan-3	30	742	1,025[1]	3,093	956	GrÆStl
Ours - GrÆStl	8/16	256	Virtex-6	80	742	1,025[1]	3,093	302	

[1] Decryption with stored last round key.

Table 3 shows a comparison with related work. Note that we listed only AES implementations that require no Block RAMs, DSPs, etc. to allow a fair comparison. It shows that our stand-alone AES implementation is the smallest on the Spartan-3 needing only 442 slices. For the Virtex-6, our design needs only 142 slices. Our Grøstl implementation needs 488 slices on the Spartan-3 and is therefore about 2 times smaller than the work of Jungk et al. [16] which requires 967 slices. Compared to the work of Kerckhof et al. [22], we need only 202 slices on the Virtex-6 instead of 260, i.e., a factor of about 1.3.

5 Conclusion

In this paper, we presented GrÆStl—a combined hardware implementation of AES-128 and Grøstl-256. GrÆStl has been designed for embedded systems and therefore shares resources as much as possible to lower the area requirements. We integrated the following optimization techniques: (1) we mapped the AES

State into the GrÆStl State matrix to avoid the need of additional memory, (2) we made use of shift registers to provide high flexibility (for ASICs as well as FPGAs) and avoid the implementation of ShiftBytes and ShiftRows, (3) we implemented the tweaked AddRoundConstant function instead of Grøstl-0 as given in related work, (4) we reused the S-box for AES and Grøstl and reused it also to increase the performance of AES round-key generation, (5) we combined MixBytes and MixColumns, and finally (6) we proposed to share the I/O registers which avoids forward round-key generation during decryption which helps to reduce 330 clock cycles in addition.

As result, GrÆStl is the first combined hardware implementation fabricated as ASIC and occupies 16.55 kGEs in total whereas AES needs an overhead of only 10 %. In particular, on a Spartan-3 FPGA, our stand-alone AES and Grøstl implementations outperform existing solutions by a factor of 4.8 and 2 in terms of area. The small area requirements and the low-power consumption of about 20 μW at 100 kHz make the design right suitable for low-resource devices such as contact-less smart cards and Radio Frequency (RF) communication based devices. Besides that, GrÆStl might be also considered to implement a low-resource authenticated-encryption scheme, since it provides the required cryptographic primitives in a single architecture.

Acknowledgment. The authors would like to thank the team from the Microelectronics Design Center of ETH Zurich, namely Beat Muheim and Frank Gürkaynak, for their constant support during the backend design of the ASIC. We also thank Martin Schläffer for fruitful discussions. This work has been supported by the European Commission through the ICT program under contract ICTSEC- 2009-5-258754 (Tamper Resistant Sensor Node – TAMPRES) and ICT-2007-216646 (European Network of Excellence in Cryptology – ECRYPT II).

References

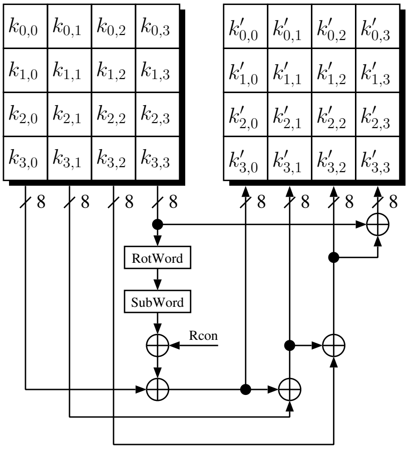
1. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak SHA-3 submission. Submission to NIST (Round 3) (2011)
2. Bulens, P., Standaert, F.X., Quisquater, J.J., Pellegrin, P., Rouvroy, G.: Implementation of the AES-128 on Virtex-5 FPGAs. In: Vaudenay, S. (ed.) Progress in Cryptology – AFRICACRYPT, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14. LNCS, vol. 5023, pp. 16–26. Springer (2008)
3. Canright, D.: A Very Compact S-Box for AES. In: Rao, J.R., Sunar, B. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2005, LNCS, vol. 3659, chap. 32, pp. 441–455. Springer, Heidelberg (2005)
4. Cao, D., Han, J., yang Zeng, X.: A Reconfigurable and Ultra Low-Cost VLSI Implementation of SHA-1 and MD5 Functions. In: International Conference on ASIC Proceeding – ICASIC 2007, 7th International Conference, Guilin, China, October 25-29, 2007. pp. 862–865. IEEE (October 2007)

5. Chodowiec, P., Gaj, K.: Very Compact FPGA Implementation of the AES Algorithm. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems – CHES, 5th International Workshop, Cologne, Germany, September 8-10. LNCS, vol. 2779, pp. 319–333. Springer, Heidelberg (2003)
6. Feldhofer, M., Wolkerstorfer, J., Rijmen, V.: AES implementation on a grain of sand. IEE Proceedings - Information Security 152(1), 13–20 (October 2005)
7. Ganesh, T.S., Sudarshan, T.S.B.: ASIC Implementation of a Unified Hardware Architecture for Non-Key Based Cryptographic Hash Primitives. In: International Conference on Information Technology: Coding and Computing – ITCC, April 4-6, 2005, Las Vegas, Nevada, USA. vol. 1, pp. 580–585. IEEE Computer Society (April 2005)
8. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – A SHA-3 Candidate. Submission to NIST (Round 3) (2011)
9. Good, T., Benaissa, M.: AES on FPGA from the Fastest to the Smallest. In: Rao, J., Sunar, B. (eds.) Cryptographic Hardware and Embedded Systems – CHES, LNCS, vol. 3659, pp. 427–440. Springer, Heidelberg (2005)
10. Guo, X., Huang, S., Nazhandali, L., Schaumont, P.: Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations. In: Second SHA-3 Candidate Conference, 2010 (2010)
11. Hamalainen, P., Alho, T., Hannikainen, M., Hamalainen, T.D.: Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core. In: 9th EUROMICRO Conference on Digital System Design – DSD. pp. 577–583. IEEE Computer Society, Washington, DC, USA (2006)
12. Henzen, L., Gendotti, P., Guillet, P., Pargaetzi, E., Zoller, M., Gürkaynak, F.K.: Developing a Hardware Evaluation Method for SHA-3 Candidates. In: Cryptographic Hardware and Embedded Systems – CHES, 12th International Workshop, Santa Barbara, USA, August 17-20. Lecture Note in Computer Science, vol. 6225, pp. 248–263. Springer, Heidelberg, Santa Barbara, CA (2010)
13. Huang, C.W., Chang, C.J., Lin, M.Y., Tai, H.Y.: Compact FPGA implementation of 32-bits AES algorithm using Block RAM. In: TENCON 2007 IEEE Region 10 Conference. pp. 1–4 (2007)
14. Järvinen, K.: Sharing Resources Between AES and the SHA-3 Second Round Candidates Fugue and Groestl. In: Second SHA-3 Candidate Conference (August 2010)
15. Järvinen, K.U., Tommiska, M., Skyttä, J.: A Compact MD5 and SHA-1 Co-Implementation Utilizing Algorithm Similarities. In: Engineering of Reconfigurable Systems and Algorithms – ERSA 2005, International Conference, Las Vegas, Nevada, USA, June 27-30, 2005. pp. 48–54. CSREA Press (2005)
16. Jungk, B., Reith, S.: On FPGA-Based Implementations of the SHA-3 Candidate Grøstl. In: Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on. pp. 316–321 (December 2010)
17. Jungk, B.: Evaluation Of Compact FPGA Implementations For All SHA-3 Finalists. SHA-3 Conference (March 2012)
18. Jungk, B., Apfelbeck, J.: Area-Efficient FPGA Implementations of the SHA-3 Finalists. In: Athanas, P.M., Becker, J., Cumplido, R. (eds.) ReConFig. pp. 235–241. IEEE Computer Society (2011)
19. Kaps, J.P., Sunar, B.: Energy comparison of AES and SHA-1 for ubiquitous computing. In: Zhou, X., Sokolsky, O., LuYan, Jung, E.S., Shao, Z., Mu, Y., Lee, D.C., Jeong, D.K.Y.S., Xu, C.Z. (eds.) 2nd IFIP International Symposium on Network Centric Ubiquitous Systems – NCUS, Seoul, Korea, August 1-4. LNCS, vol. 4097, pp. 372–381. Springer, Heidelberg (2006)

20. Kaps, J.P., Vadlamudi, P.Y.S., Gurung, K.K.S.S., Habib, B.: Lightweight Implementations of SHA-3 Finalists on FPGAs. SHA-3 Conference (March 2012)
21. Katashita, T.: Groestl compact. http://www.rcis.aist.go.jp/special/SASEBO/ (Februar 2010)
22. Kerckhof, S., Durvaux, F., Veyrat-Charvillon, N., Regazzoni, F., de Dormale, G.M., Standaert, F.X.: Compact FPGA Implementations of the Five SHA-3 Finalists. In: Prouff, E. (ed.) CARDIS. LNCS, vol. 7079, pp. 217–233. Springer (2011)
23. Kim, M., Ryou, J., Choi, Y., Jun, S.: Low Power AES Hardware Architecture for Radio Frequency Identification . In: Yoshiura, H., Sakurai, K., Rannenberg, K., Murayama, Y., Kawamura, S. (eds.) First International Workshop on Security – IWSEC, Kyoto, Japan, October 23-24. LNCS, vol. 4266, pp. 353–363. Springer, Heidelberg (October 2006)
24. 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. (ed.) Advances in Cryptology – EUROCRYPT 2011, LNCS, vol. 6632, pp. 69–88. Springer, Heidelberg (2011)
25. Nikova, S., Rijmen, V., Schläffer, M.: Using Normal Bases for Compact Hardware Implementations of the AES S-box. In: 6th International Conference Security in Communication Networks – SCN
26. NIST: Advanced Encryption Standard (AES) (FIPS PUB 197). National Institute of Standards and Technology (November 2001)
27. Sharif, M.U., Shahid, R., Rogawski, M., Gaj, K.: Use of Embedded FPGA Resources in Implementations of Five Round Three SHA-3 Candidates. In: ECRYPT II Hash Workshop 2011 (2011)
28. Tillich, S., Feldhofer, M., Issovits, W., Kern, T., Kureck, H., Mühlberghuber, M., Neubauer, G., Reiter, A., Köfler, A., Mayrhofer, M.: Compact Hardware Implemenations of the SHA-3 Candidates ARIRANG, BLAKE, Grøstl and Skein. In: Auer, M., Pribyl, W., Söser, P. (eds.) Proceedings of Austrochip 2009, October 7, 2009, Graz, Austria. pp. 69–74 (2009)
29. Wang, M.Y., Su, C.P., Huang, C.T., Wu, C.W.: An HMAC processor with integrated SHA-1 and MD5 algorithms. In: Imai, M. (ed.) Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair – ASP-DAC, Yokohama, Japan, January 27-30. pp. 456–458. IEEE (January 2004)
30. Wolkerstorfer, J., Oswald, E., Lamberger, M.: An ASIC Implementation of the AES SBoxes. In: Preneel, B. (ed.) Topics in Cryptology – CT-RSA, LNCS, vol. 2271, pp. 29–52. Springer, Heidelberg (2002)
31. Xilinx: HDL Coding Practices to Accelerate Design Performance. (May 2012), http://www.xilinx.com/support/documentation/white_papers/wp231.pdf

A AES-128 Key Generation

Figure 6 and Fig. 7 illustrate the general forward and backward round-key generation of AES-128. Each of the round key words is stored in a single column and updating the keys is done column by column.


$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$
$k'_{0,0}$ $k'_{0,1}$ $k'_{0,2}$ $k'_{0,3}$
$k'_{1,0}$ $k'_{1,1}$ $k'_{1,2}$ $k'_{1,3}$
$k'_{2,0}$ $k'_{2,1}$ $k'_{2,2}$ $k'_{2,3}$
$k'_{3,0}$ $k'_{3,1}$ $k'_{3,2}$ $k'_{3,3}$
8
RotWord
SubWord
Rcon

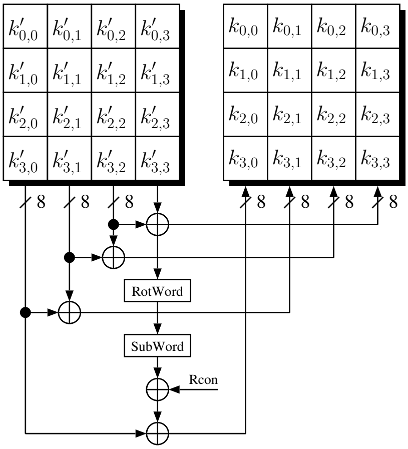

$k'_{0,0}$ $k'_{0,1}$ $k'_{0,2}$ $k'_{0,3}$
$k'_{1,0}$ $k'_{1,1}$ $k'_{1,2}$ $k'_{1,3}$
$k'_{2,0}$ $k'_{2,1}$ $k'_{2,2}$ $k'_{2,3}$
$k'_{3,0}$ $k'_{3,1}$ $k'_{3,2}$ $k'_{3,3}$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$
8
RotWord
SubWord
Rcon

Fig. 6. Forward round-key generation **Fig. 7.** Backward round-key generation