# Sharing Resources Between AES and the SHA-3 Second Round Candidates Fugue and Grøstl

Kimmo Järvinen*

Aalto University, School of Science and Technology
Department of Information and Computer Science
P.O. Box 15400, FI-00076 Aalto, Finland
kimmo.jarvinen@tkk.fi

**Abstract.** Four out of the 14 second round candidates of the NIST SHA-3 cryptographic hash algorithm competition are so-called AES-inspired algorithms which share common structure and features with AES or even use it as a subroutine. This paper focuses on two of them, Fugue and Grøstl, and studies how efficiently logic can be shared in implementations combining them with AES. It will be shown that adding AES into the data paths is cheap both in terms of area and delay and, consequently, combined implementations are feasible in practice. Especially Grøstl achieves very small overheads. Such implementations have importance in a large variety of applications because they offer high-speed computations of a cryptographic hash algorithm and a block cipher with an area cost that is only slightly larger than a hash algorithm implementation alone. The paper presents methods to embed AES computation(s) into the data paths of both Fugue and Grøstl and presents prototype implementations on an Altera Cyclone III FPGA.

## 1 Introduction

Certain candidate algorithms submitted to the SHA-3 competition are AES-inspired meaning that their design has been strongly influenced by the design of AES [1]: they share common structure and features or even use AES as a subroutine. AES-inspired algorithms in the second round of the competition are ECHO [2], Fugue [3], Grøstl [4], and SHAvite-3 [5]. They offer a unique opportunity to design implementations that share most of the resources (silicon area, programmable logic resources, program code, etc.) between AES and a hash algorithm and consequently provide both algorithms with significantly fewer resources. This paper studies resource sharing from the hardware point-of-view and presents several prototype implementations on a field-programmable gate array (FPGA).

ECHO and SHAvite-3 use AES directly as a subroutine. Therefore, they can support plain AES computations with negligible overheads. Grøstl and Fugue, on the other hand, are only inspired by AES: they share common structure and features with AES, but do not use it directly. It is, therefore, uncertain how easily support for AES could be added and how much overhead it would introduce into area requirements and computation delays. In this paper, we show that both of them (and, especially, Grøstl) can be efficiently combined with AES. Implementations of Fugue have been introduced in [3, 6] and Fugue has been shown to provide good performance with area requirements which are smaller than for most other candidates [6]. Implementations of Grøstl have been more intensively studied and they are available in [4, 6–10]. Grøstl has been shown to scale to both low-cost [10] and high-throughput [6, 9] applications. Studies comparing candidates have shown that Grøstl offers a good balance between area and speed [6, 7, 9, 10]. Grøstl offers faster performance than Fugue, but requires more area [6]. None of the papers discussed combining Fugue or Grøstl with AES and this is the first paper studying this issue for any of the SHA-3 candidates.

Sharing the data path between two or more cryptographic algorithms is not a new idea. Combined architectures have been presented for MD5 and SHA-1 in [11–13], for MD5 and RIPEMD-160 in [14], and for MD5, SHA-1, and RIPEMD-160 in [15]; i.e., all the above work has shared the data path of two or more cryptographic hash algorithms into one compact design. However, it would be of much greater relevance to have a compact, high-performance combined implementation of a block cipher and a cryptographic hash algorithm because then two truly different functionalities could be offered at once. It would be even more

* The author is supported by the EU FP7 project CACE

important if both algorithms would be widely used in many applications—something that AES already is and SHA-3 will most likely become.

Previous efforts to provide support for cryptographic hash algorithms and block ciphers with a single design are generic cryptoprocessors, such as, CRYPTONITE [16] or CryptoManiac [17], or instruction set extensions for cryptography, e.g., in [18]. Because of the generality of such designs, their performance and area requirements are necessarily worse than the ones of algorithm-specific implementations, such as the ones that will be described in this paper.

Intel is including an AES instruction set extension in the Intel Core family processors starting from the beginning of 2010 [19]. The adaptability of these instructions for AES-inspired SHA-3 candidates was recently studied in [20] where they concluded that ECHO and SHAvite-3, which use AES directly as a subroutine, are the only second round candidates that are likely to benefit from the instruction set extension. Grøstl and Fugue's designs were stated to be "too distant" to be able to take advantage from it [20]. However, we shall see in this paper that the differences between the data paths of AES, Fugue, and Grøstl are minor and, as a consequence, it could be possible to incorporate support for Fugue or Grøstl into AES instruction set extensions with reasonable overheads.

The remainder of the paper builds up as follows. Section 2 presents AES, Fugue, and Grøstl. Sections 3 and 4 discuss the similarities and differences of AES with Fugue and Grøstl, respectively, together with methods to combine the algorithms. Architectures based on the findings are introduced in Sect. 5 and 6 and implementation results on an Altera Cyclone III FPGA are provided in Sect. 7. Section 8 ends the paper with comparisons of the implementations, discussion about the consequences of the findings, and suggestions for future work.

## 2 Algorithms

This section describes the three algorithms studied in the paper. Some details considered unrelevant for this paper are omitted in order to keep the discussion clear; interested readers are referred to the standard [1] and the submission documents [3, 4] for details.

The algorithms have several variants. AES can use either 128-bit, 192-bit, or 256-bit keys and the variants are called AES-128, AES-192, or AES-256 depending on the key length. Fugue and Grøstl both come with four variants: Fugue-224, Fugue-256, Fugue-384, Fugue-512, Grøstl-224, Grøstl-256, Grøstl-384, and Grøstl-512 where the values denote the lengths of the hashes. Again, in order to keep the discussion clear, we focus on specific variants: AES-128 (128-bit key), Fugue-256, and Grøstl-256 (256-bit hashes). In the following, Fugue refers to Fugue-256, Grøstl to Grøstl-256, and AES to AES-128 unless explicitly stated otherwise. Most of the ideas can be straightforwardly generalized to the other variants.

### 2.1 Advanced Encryption Standard (Rijndael)

AES encrypts messages in 128-bit blocks with a 128-bit encryption key $K$. Each block is encrypted by iterating a round transformation 10 times. Each round involves a 128-bit round key, $k$, derived from $K$ with KeyExpansion routine. A block is represented as a $4 \times 4$ matrix of bytes called the State. Each byte is interpreted as an element of the finite field $GF(2^8) : GF(2)[x]/x^8+x^4+x^3+x+1$. The round transformation consists of the following transformations:

**SubBytes** handles each byte $b$ of the State separately. It consists of two steps: (1) a multiplicative inverse in $GF(2^8)$ (00 maps to itself) and (2) an affine transformation defined by $b'_i = b_{(i+4) \bmod 8} + b_{(i+6) \bmod 8} + b_{(i+7) \bmod 8} + c_i$, where $b_i$ is the $i^{\text{th}}$ bit of the byte $b$ and $c = 63$.

**ShiftRows** shifts the rows of the State cyclically to the left by $i$ bytes, where $0 \leq i \leq 3$ is the index of the row; i.e., the first row is not shifted, the second is shifted by one byte, the third by two, and the last by three.

**MixColumns** operates each column of the State separately. A column is interpreted as a polynomial over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with the polynomial $03x^3 + 01x^2 + 01x + 02$. MixColumns can be seen as a matrix multiplication where a 4-byte vector (the column) is multiplied with a $4 \times 4$ matrix from the left (the matrix is given in (3) in Sect. 3.2). MixColumns is omitted in the last round.

**AddRoundKey** adds a roundkey to the State with a bitwise exclusive-or (xor).

**KeyExpansion** derives 128-bit round keys from the encryption key. The expansion is carried out iteratively. Four bytes of the previous 128-bit round key are mapped with SubBytes and a round constant, $02^i$ where $i$ is the round index, is xorred to one of these bytes. The bytes are then shifted by one byte, after which a new round key is obtained with four 32-bit xors. The reader is referred to [1] for more details.

### 2.2 Fugue

Fugue-256 generates a 256-bit hash $H$ for a message $M$ which is first padded (see [3] for details) and split into $t$ 32-bit blocks $m_i$. Fugue-256 uses a 960-bit chaining value, $h$, which is initialized with an initial value $h_0 = iv$ and split into 32-bit words $S_i$ where $0 \leq i \leq 29$. Then, $h$ is updated for each $m_i$ with the following transformation sequence: TIX, ROR3, CMIX, SMIX, ROR3, CMIX, and SMIX (see below for details), of which the AES-inspired SMIX transformation is by far the most complex transformation.

When all $m_i$ have been processed, the computation ends with a final round consisting of the following transformations: ROR3, CMIX, SMIX, ROR15, ROR14, and 32-bit bitwise xors. Most notably, SMIX is applied 36 times during this sequence. Finally, $H$ is constructed as $S_1|S_2|S_3|S_4|S_{15}|S_{16}|S_{17}|S_{18}$. Next, the aforementioned transformations are discussed with more details.

**TIX and CMIX** are both sequences of 32-bit bitwise xors. TIX is the following sequence of operations: $S_{10} = S_{10} \oplus S_0$, $S_0 = m_i$, $S_8 = S_8 \oplus S_0 = S_8 \oplus m_i$, and $S_1 = S_1 \oplus S_{24}$. CMIX performs the following xors: $S_0 = S_0 \oplus S_4$, $S_1 = S_1 \oplus S_5$, $S_2 = S_2 \oplus S_6$, $S_{15} = S_{15} \oplus S_4$, $S_{16} = S_{16} \oplus S_5$, and $S_{17} = S_{17} \oplus S_6$.

**ROR3, ROR14, and ROR15** rotate the 32-bit words of the chaining value to the right by as many positions as described in the name: $S_i = S_{(i-r) \bmod 30}$ where $r$ is 3, 14, or 15 for ROR3, ROR14, and ROR15, respectively.

**SMIX** updates the first 128 bits of $h$, i.e., $S_0 \ldots S_3$, in two steps. The first step is SubBytes of AES: each byte is mapped with a multiplicative inverse in $GF(2^8)$ followed by the affine transformation. The second step called Super-Mix is inspired by MixColumns of AES. However, it differs from MixColumns so that MixColumns handles each column separately whereas Super-Mix introduces cross-mixing between the columns. Similarly as MixColumns, also Super-Mix can be seen as a matrix multiplication: a 16-byte vector $(S_0| \ldots |S_3)$ is multiplied from the left with a $16 \times 16$ matrix (the matrix is given in (4) in Sect. 3.2).

### 2.3 Grøstl

Grøstl-256 generates a 256-bit hash $H$ for a message $M$ with a compression function $f$. $M$ is padded (see [4] for details) and split into $t$ 512-bit message blocks $m_i$. The function $f$ updates a 512-bit chaining value $h$ (with an initial value $h_0 = iv$) by iterating $h_i \leftarrow f(h_{i-1}, m_i)$ for $i = 1, \ldots, t$. The compression function $f$ is constructed from two permutations, $P$ and $Q$, as follows:

$$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h \ . \tag{1}$$

When $f$ has been applied to all $t$ message blocks, $H$ is obtained with an output transformation $\Omega$ in the following way:

$$H = \Omega(h) = \mathrm{trunc}_{256}(P(h) \oplus h) \tag{2}$$

where $\mathrm{trunc}_{256}$ returns the rightmost 256 bits of its input.

As shown above, the core of Grøstl is formed by the $P$ and $Q$ permutations. They are strongly influenced by the design of AES. $P$ and $Q$ are almost identical and the following description applies to both of them unless explicitly stated otherwise. The 512-bit inputs of the permutations are represented as an $8 \times 8$ matrix of bytes, called the State, and each byte is, again, interpreted as an element of $GF(2^8)$. The permutations are performed by applying a round transformation 10 times. The round transformation consists of four transformations which are described below:

**AddRoundConstant** adds (xor) a round-dependent constant to the State. The constant is $i\,00 \ldots 00$ for the $P$ permutation and $00\,00\,00\,00\,00\,00\,00\,(i \oplus \mathrm{ff})\,00 \ldots 00$ for the $Q$ permutation, where $i$ is the index of the round.

**SubBytes** is similar to SubBytes of AES. The only exception is that Grøstl has a 512-bit State and, therefore, SubBytes operates on 64 bytes instead of 16 as in AES.

**ShiftBytes** is also principally the same operation as ShiftRows in AES: again, a row is shifted cyclically to the left by $i$ bytes. The difference is that there are eight rows.

**MixBytes** operates on each column of the State separately. A column is interpreted as a polynomial over $GF(2^8)$ and multiplied modulo $x^8+1$ with the polynomial $02x^7+03x^6+04x^5+05x^4+03x^3+05x^2+07x+02$. The matrix for the matrix multiplication is given in (5) in Sect. 4.4.

## 3 Observations: Fugue

In the following, we focus on SMIX because it is the only AES-inspired part of Fugue. A natural way to implement SMIX is to use a 128-bit data path which we will discuss in the following.

### 3.1 SubBytes

SubBytes can be shared between AES and Fugue's SMIX without any modifications. This is a major advantage because SubBytes is the most significant contributor to both area requirements and critical path delay of a typical AES [21] or Fugue [3] implementation.

### 3.2 Super-Mix and MixColumns

As mentioned, MixColumns and Super-Mix can be represented as a matrix multiplication $B \times A$ where $A$ is either the 4-byte column or the 16-byte vector, $S_0|S_1|S_2|S_3$, for AES and Fugue, respectively. In the case of AES, $B$ is the following $4 \times 4$ matrix:

$$B_{\text{AES}} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} . \tag{3}$$

Super-Mix of Fugue employs the following $16 \times 16$ matrix:

$$B_{\text{S-Mix}} = \begin{bmatrix} 01\ 04\ 07\ \mathbf{01} & 01\ 00\ 00\ 00 & 01\ 00\ 00\ 00 & 01\ 00\ 00\ 00 \\ 00\ 01\ 00\ 00 & 01\ 01\ 04\ 07 & 00\ 01\ 00\ 00 & 00\ 01\ 00\ 00 \\ 00\ 00\ 01\ 00 & 00\ 00\ 01\ 00 & 07\ 01\ 01\ 04 & 00\ 00\ 01\ 00 \\ 00\ 00\ 00\ 01 & 00\ 00\ 00\ 01 & 00\ 00\ 00\ 01 & 04\ 07\ 01\ 01 \\ 00\ 00\ 00\ 00 & 00\ 04\ 07\ \mathbf{01} & 01\ 00\ 00\ 00 & 01\ 00\ 00\ 00 \\ 00\ 01\ 00\ 00 & 00\ 00\ 00\ 00 & 01\ 00\ 04\ 07 & 00\ 01\ 00\ 00 \\ 00\ 00\ 01\ 00 & 00\ 00\ 01\ 00 & 00\ 00\ 00\ 00 & 07\ 01\ 00\ 04 \\ 04\ 07\ 01\ 00 & 00\ 00\ 00\ 01 & 00\ 00\ 00\ 01 & 00\ 00\ 00\ 00 \\ 00\ 00\ 00\ 00 & 07\ 00\ 00\ 00 & 06\ 04\ 07\ \mathbf{01} & 07\ 00\ 00\ 00 \\ 00\ 07\ 00\ 00 & 00\ 00\ 00\ 00 & 00\ 07\ 00\ 00 & 01\ 06\ 04\ 07 \\ 07\ 01\ 06\ 04 & 00\ 00\ 07\ 00 & 00\ 00\ 00\ 00 & 00\ 00\ 07\ 00 \\ 00\ 00\ 00\ 07 & 04\ 07\ 01\ 06 & 00\ 00\ 00\ 07 & 00\ 00\ 00\ 00 \\ 00\ 00\ 00\ 00 & 04\ 00\ 00\ 00 & 04\ 00\ 00\ 00 & 05\ 04\ 07\ \mathbf{01} \\ 01\ 05\ 04\ 07 & 00\ 00\ 00\ 00 & 00\ 04\ 00\ 00 & 00\ 04\ 00\ 00 \\ 00\ 00\ 04\ 00 & 07\ 01\ 05\ 04 & 00\ 00\ 00\ 00 & 00\ 00\ 04\ 00 \\ 00\ 00\ 00\ 04 & 00\ 00\ 00\ 04 & 04\ 07\ 01\ 05 & 00\ 00\ 00\ 00 \end{bmatrix} . \tag{4}$$

Clearly, it is possible to represent the four independent column operations computed in MixColumns of AES as a single matrix multiplication. In that case, the resulting $16 \times 16$ matrix is such that its diagonal consists of four $B_{\text{AES}}$ matrices and the other elements are zeros. Comparing this matrix to (4) reveals that four elements are the same in these matrices (bolded in (4)) and, consequently, they can be shared in an implementation.

### 3.3 ShiftRows

SMIX does not have an operation that would relate to ShiftRows of AES. Because SubBytes targets to individual bytes, ShiftRows can be performed either before or after SubBytes. The superiority of these two cases is highly platform dependent. In the first case, we either rearrange the bytes (AES) or do nothing (Fugue) before SubBytes. In the second case, it would be natural to embed ShiftRows into the combined Super-Mix and MixColumns transformation; i.e., we constuct the $16 \times 16$ matrix performing the four column operations of MixColumns so that it performs also ShiftRows. Unfortunately, this results in a situation where no elements can be shared in an implementation between MixColumns and Super-Mix (cf. Sect. 3.2).

### 3.4 KeyExpansion

The above discussion considered only the main data path of AES, but each round also requires a round key from KeyExpansion. There are essentially two options to provide the round keys: (1) they are input into the data path from an external source or (2) their computation is embedded into the data path itself.

The option (1) is trivial: either KeyExpansion is computed simultaneously with an external circuitry or the round keys are stored into memory before the computation from where the correct round key is fetched for each round. Both of these alternatives are used in numerous AES implementations and they can be directly adapted to a combined AES and Fugue (or Grøstl) implementation; hence, we do not discuss this option further.

The option (2) reuses the resources of SMIX data path for KeyExpansion. The heaviest operation in KeyExpansion is SubBytes transformation applied to four bytes and this can be shared entirely by utilizing SubBytes needed in both AES and SMIX. This option is discussed further in Sect. 5.3 after presenting the architecture of the data path.

## 4 Observations: Grøstl

Although Grøstl is defined almost identically to AES, AES implementations or instructions cannot be directly utilized in implementing Grøstl [20]. The difficulties originate from the facts that ShiftBytes and MixBytes differ slightly from ShiftRows and MixColumns and the States have different sizes.

Grøstl can be efficiently implemented using 512-bit data path and registers [4, 6–9]. Therefore, it is tempting to ask whether the data path and registers could be shared so that they would enable four parallel AES encryptions, each having 128-bit data path and registers. In the following, we shall concentrate on this possibility because support for fewer parallel encryptions can be easily derived from it by dropping out logic of some encryptions.

### 4.1 States

The most straightforward way of fitting four AES States into one Grøstl State would be to simply concatenate the four 128-bit States: $m_1|m_2|m_3|m_4$. In this case, one AES State would occupy two columns of the Grøstl State. This approach would complicate the combination of ShiftBytes and ShiftRows and, hence, we use the following representation.

Let $m_{i,j}$ denote the $j^{\text{th}}$ 32-bit word of $m_i$ so that $m_{i,1}$ is the most significant (the leftmost) word and $m_{i,4}$ is the least significant (the rightmost) word. The message blocks are then concatenated as follows: $m_{1,1}|m_{2,1}|m_{1,2}|m_{2,2}|m_{1,3}|m_{2,3}|\ m_{1,4}|m_{2,4}|m_{3,1}|m_{4,1}|m_{3,2}|m_{4,2}|m_{3,3}|m_{4,3}|m_{3,4}|m_{4,4}$. Now, $m_1$ forms the upper-left corner, $m_2$ the lower-left corner, $m_3$ the upper-right corner, and $m_4$ the lower-right corner of the $8 \times 8$ matrix. This setup is depicted in Fig. 1a.

### 4.2 SubBytes

SubBytes can be shared between AES and Grøstl without any modifications. This is, again, a major advantage because SubBytes plays a major role also in Grøstl implementations [7].

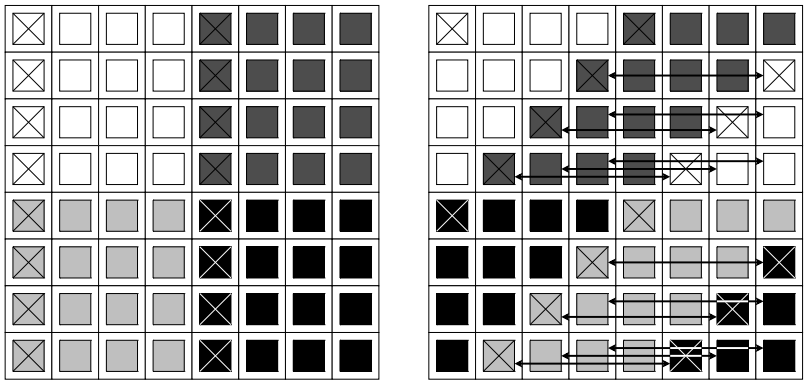

Fig. 1: Merge of ShiftBytes and ShiftRows. (a) Four AES States (shown in different shades of gray) fit into one Grøstl State; bytes of the leftmost columns of the AES States are highlighted with X. (b) States after applying ShiftBytes; the arrows denote the 12 swaps required to construct four parallel ShiftRows (the two lower States swap positions).

### 4.3 ShiftBytes and ShiftRows

Although both ShiftBytes and ShiftRows rotate the row $i$ by $i$ bytes, the differences in the sizes of the States complicate combining these transformations. Applying ShiftBytes to four AES States moves parts of the States on the left into the regions of the States on the right, and vice versa; see Fig. 1b. Hence, 16 byte swaps are necessary in order to restore the original setup where the four AES States occupy the corners of the Grøstl State matrix: 6 byte swaps for the upper half and 10 for the lower half. However, if we allow the two States in the lower half to swap positions, only 12 byte swaps are needed (six for both halves). The order of the States is correct in the end of an encryption because AES (also AES-192 and AES-256) consists of an even number of rounds; hence, ShiftRows is applied an even number of times. Fig. 1b shows this construction.

### 4.4 MixBytes and MixColumns

The matrix multiplication performing MixBytes multiplies an 8-byte vector $A$ with the following $8 \times 8$ matrix:

$$B_{\text{Grøstl}} = \begin{bmatrix} 02\ 02\ 03\ 04\ 05\ 03\ 05\ 07 \\ 07\ 02\ 02\ 03\ 04\ 05\ 03\ 05 \\ 05\ 07\ 02\ 02\ 03\ 04\ 05\ 03 \\ 03\ 05\ 07\ 02\ 02\ 03\ 04\ 05 \\ 05\ 03\ 05\ 07\ 02\ 02\ 03\ 04 \\ 04\ 05\ 03\ 05\ 07\ 02\ 02\ 03 \\ 03\ 04\ 05\ 03\ 05\ 07\ 02\ 02 \\ 02\ 03\ 04\ 05\ 03\ 05\ 07\ 02 \end{bmatrix} . \tag{5}$$

Because one column of the $8 \times 8$ matrix includes two columns from two different AES States, the matrix $B_{\text{AES}}$ is extended to perform two column operations simultaneously (cf. Sect. 3.2 where the matrix was extended to perform four column operations):

$$B_{\text{AES, ext.}} = \begin{bmatrix} \mathbf{02}\ 03\ 01\ 01\ 00\ 00\ 00\ 00 \\ 01\ \mathbf{02}\ 03\ 01\ 00\ 00\ 00\ 00 \\ 01\ 01\ \mathbf{02}\ 03\ 00\ 00\ 00\ 00 \\ \mathbf{03}\ 01\ 01\ \mathbf{02}\ 00\ 00\ 00\ 00 \\ 00\ 00\ 00\ 00\ \mathbf{02}\ 03\ 01\ 01 \\ 00\ 00\ 00\ 00\ 01\ \mathbf{02}\ 03\ 01 \\ 00\ 00\ 00\ 00\ 01\ 01\ \mathbf{02}\ 03 \\ 00\ 00\ 00\ 00\ \mathbf{03}\ 01\ 01\ \mathbf{02} \end{bmatrix} . \tag{6}$$

Ten elements (bolded in (6)) are the same also in $B_{\text{Grøstl}}$ and, hence, they can be shared in an implementation.

### 4.5 KeyExpansion

As discussed in Sect. 3.4, there are essentially two options to provide the round keys: (1) they are provided from an external source or (2) the data path is reused for KeyExpansion. The option (2) is especially attractive in the case of Grøstl. In Sect. 4.1, we placed four AES States into a Grøstl State. If KeyExpansion is embedded into the data path, we dedicate either one or two of these slots for the round key(s) and, consequently, the number of parallel AES computations decreases to three or two, respectively. In the first case, all three AES computations share a common key whereas both computations can have different keys in the second case. This option is discussed further in Sect. 6.3.

## 5 Implementations: Fugue

The observations of Sect. 3 can be exploited (with minor modifications) in implementations with different data path widths, but in the following we shall consider the simplest case: the 128-bit data path for SMIX. Fig. 2a presents the high-level architecture and Fig. 2b shows the 128-bit main data path (SMIX).

### 5.1 High-level Architecture

The high-level architecture shown in Fig. 2a includes the main data path (Fig. 2b), a register for $h$, logic for the simple transformations, and a multiplexer for selecting the inputs for the data path. The control signals (select signals for multiplexers) are omitted for clarity. The architecture has a 128-bit input interface and it returns an 288-bit output instead of the 32-bit input and 256-bit output of a straightforward Fugue implementation.

Fugue requires two clock cycles for every 32-bit $m_i$. TIX, ROR3, CMIX, and SMIX transformations are computed during the first clock cycle and ROR3, CMIX, and SMIX during the second. When all $m_i$ have been processed ROR3, CMIX, and SMIX are iterated 10 times after which XOR1, ROR15, and SMIX and XOR2, ROR14, and SMIX are both iterated 13 times. The hash $H$ is in the most significant 256 bits of the output: $S_1|\ldots|S_4|S_{15}|\ldots|S_{18}$ ($S_4$ and $S_{15}$ are updated with xors of the XOR1 block).

AES computation starts by setting the register to zero and by selecting $m \oplus K$ as the input, where $K$ is the encryption key. The result is returned after the data path has been iterated ten times (by using the multiplexer's second input from the right). The AES-related inputs of the multiplexer are arranged so that they perform ShiftRows. The encrypted message is in the least significant 128 bits of the output: $S_0|\ldots|S_3$.

### 5.2 The Data Path

The 128-bit data path shown in Fig. 2b implements the round transformations of AES and Fugue's SMIX. Each signal in Fig. 2b represents one byte. Bytes are ordered column-wise so that the first column of the AES State is on the left. The data path operates from top to bottom as follows.

**SubBytes** is computed with 16 S-boxes and the transformation is the same for both AES and Fugue. Implementation of SubBytes mostly determines the efficiency of AES implementations and, consequently, a lot of work has been devoted for optimizing it. In the implementations of this paper, SubBytes uses the composite field S-boxes from [22], but all other implementation options could be used without touching the other parts of the data path.

**Super-Mix and MixColumns** are implemented in two steps. Bytes are first fed into a multiplication layer which multiplies bytes $b_i$ with 02, 03, 04, 05, 06, and 07, as shown in (3) and (4), using bitwise xors and xtime-blocks. An xtime-block multiplies a byte with $x$ (02) in $GF(2^8)$. The results, together with 00 and $b_i$, are routed through the shift net to the xor layer. The xor layer computes the result of the matrix multiplication with sixteen xor trees. The inputs to the xor trees are selected with multiplexers. For example, the xor tree of the leftmost byte computes $01b_0 \oplus 04b_1 \oplus 07b_2 \oplus 01b_3 \oplus 01b_4 \oplus 01b_8 \oplus 01b_{12}$ for Fugue and $02b_0 \oplus 03b_1 \oplus 01b_2 \oplus 01b_3$ for AES. The multiplexers corresponding to the bolded values in (4) were removed because the values are the same for both AES and Fugue. Multiplexers are used also in the end for selecting whether the transformations are performed or not. The latter case is needed only for the last round of AES which omits MixColumns. The implementation of Super-Mix and MixColumns is analogous to the implementation of MixBytes and MixColumns which is discussed in Sect. 6.2 and depicted in Fig. 5.

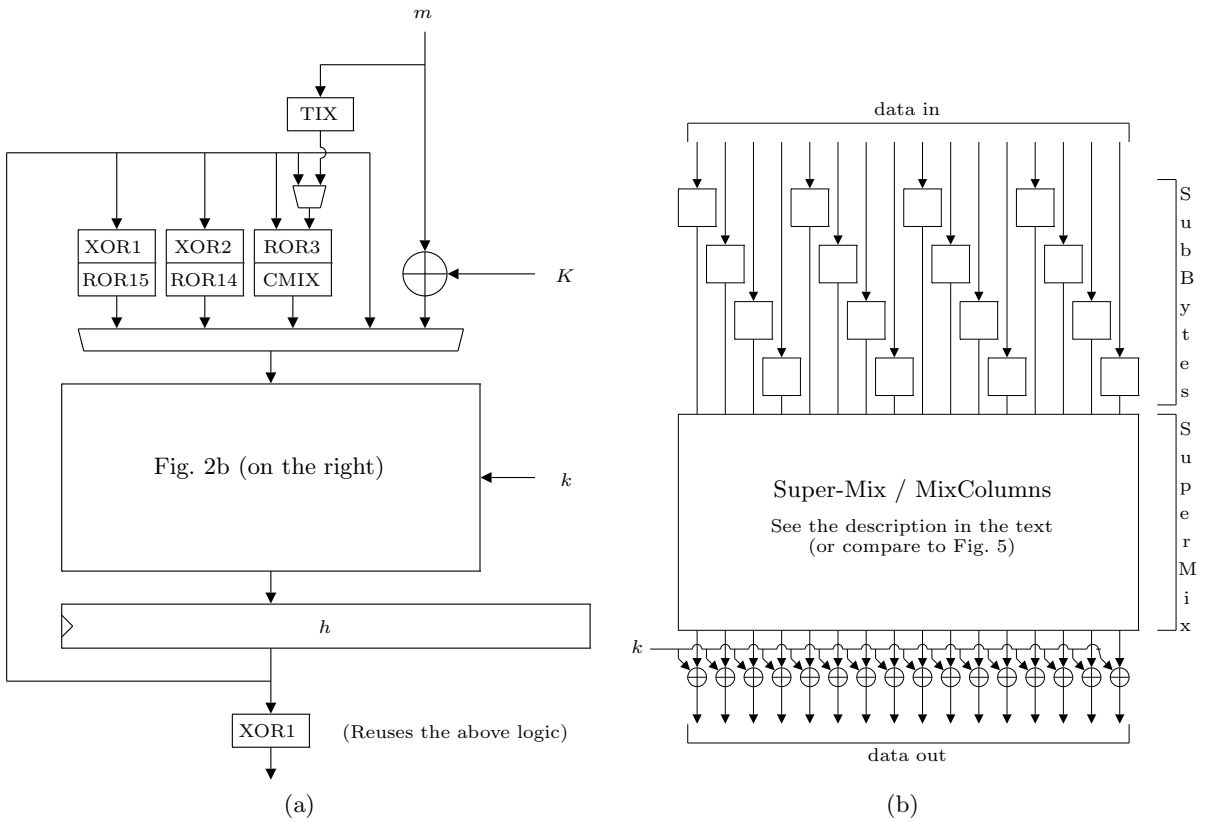


Fig. 2: (a) High-level architecture of the combined AES and Fugue implementation; (b) 128-bit data path combining AES and Fugue's SMIX.

**AddRoundKey** is implemented with the xors in the bottom of Fig. 2b. Each byte of the round key, $k$, is xorred with the corresponding byte of the State. The key is set to $k = 00 \ldots 00$ when the data path computes Fugue.

### 5.3 KeyExpansion in the Data Path

This section decribes how KeyExpansion can be embedded into the data path. The round key computation uses SubBytes for four bytes (see Sect. 2.1) which can be shared with the combined AES and Fugue data path. The remaining of the round key computation differs significantly: the combined AES and Fugue data path performs Super-Mix or MixColumns whereas KeyExpansion adds a round constant, performs byte shifts, and computes four 32-bit xors. Hence, the logic is not shared beyond SubBytes. It is, however, noteworthy that the majority of resources used in KeyExpansion are for SubBytes. The round key is stored in $h$ in $S_4|\ldots|S_7$ (the AES State is in $S_0|\ldots|S_3$) and it is routed to the $k$ input of the data path shown in Figs. 2a and 2b. AES encryption proceeds so that, first, one computes the round key with KeyExpansion and, then, the round using that round key. Hence, the latency of encryption doubles from ten to twenty clock cycles compared to the case where round keys are provided by an external source.

## 6 Implementations: Grøstl

Also in this case, the observations of Sect. 4 can be exploited (with minor modifications) in implementations with different data path widths, but in the following we shall consider the simplest case: the 512-bit data path.

Fig. 3 presents the high-level architecture. The 512-bit main data path is depicted in Fig. 4 and the combined MixBytes and MixColumns is given in Fig. 5.

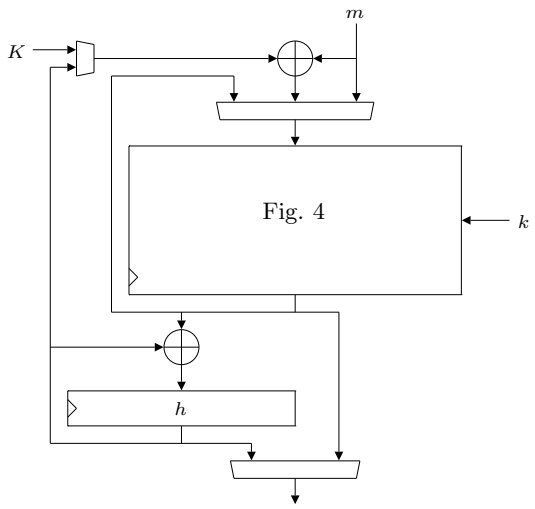


Fig. 3: High-level architecture of the combined AES and Grøstl implementation.

### 6.1 High-level Architecture

The high-level architecture shown in Fig. 3 includes the main data path (Fig. 4), a register for $h$, xors, and multiplexers for selecting the inputs for the data path.

Grøstl computation starts with $P(h \oplus m)$, i.e., the input $h \oplus m$ is selected with the multiplexers, where $h$ is the initialization vector ($h_0$) or the hash value from the previous iteration. When all 10 rounds have been computed by selecting the output of the data path with the multiplexer, the result is added to the register ($P(h \oplus m) \oplus h$). The computation proceeds with $Q(m)$ and the final result is stored to the register ($P(h \oplus m) \oplus Q(m) \oplus h$). If the last message block has been processed, $\Omega$ is computed: $P(h \oplus 0) \oplus h$ and the result is in the register.

AES computation starts by setting the register to zero and by selecting $m \oplus K$ as the input. The result is returned after the data path has been iterated ten times.

### 6.2 The Data Path

The 512-bit data path shown in Fig. 4 implements the round transformations of AES and Grøstl. Each signal in Fig. 4, again, represents one byte and they are ordered so that the first column of the Grøstl State is on the left and the last on the right. AES States fold as presented in Fig. 1a. The data path operates from top to bottom as follows.

**AddRoundConstant** is implemented with the two xors and multiplexers on the upper-left corner of Fig. 4. The one on the left adds the round index $i$ and, consequently, implements AddRoundConstant of the $P$ permutation. The one on the right adds $i \oplus \mathrm{ff}$ and it is used for the $Q$ permutation. Selecting the right input of the multiplexer skips AddRoundConstant transformation for that byte.

**SubBytes** is computed with 64 S-boxes and the transformation is the same for both AES and Grøstl. We use S-boxes introduced in [22] also in this case.

**ShiftBytes and ShiftRows** are implemented as presented in Sect. 4.3. The 12 byte swaps needed to restore the AES States are performed with 24 multiplexers below the shift net; the left inputs of the multiplexers are for Grøstl and the right inputs for AES.

**MixBytes and MixColumns** are implemented as depicted in Fig. 5. Bytes are first fed into a multiplication layer which multiplies each byte $b_i$ with 02, 03, 04, 05, and 07 using bitwise xors and xtime-blocks. The results, together with 00 and $b_i$, $0 \leq i \leq 7$, are routed through the shift net to the xor layer. The xor layer computes

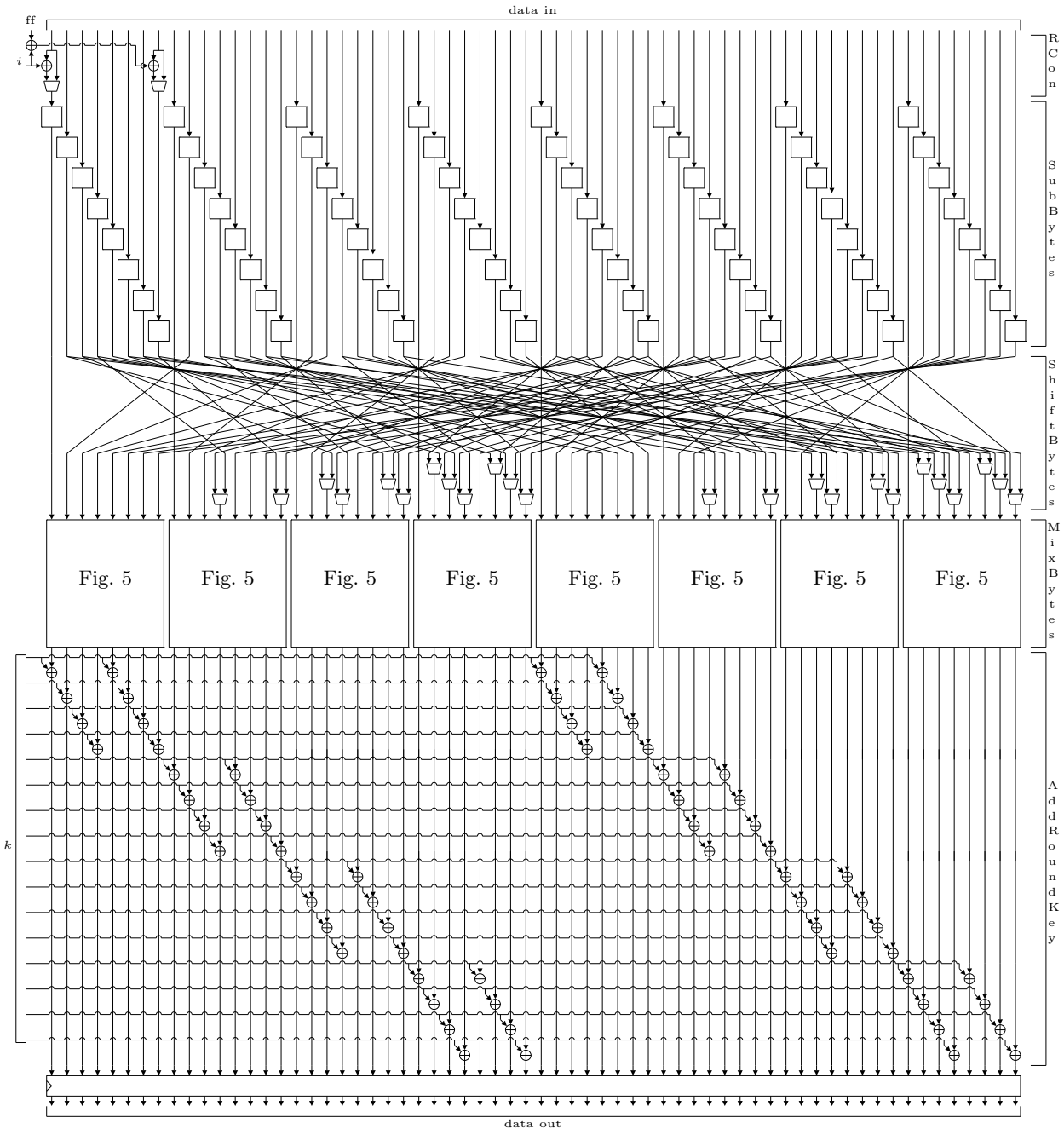


Fig. 4: 512-bit data path combining AES and Grøstl.

the result of the matrix multiplication with eight eight-to-one byte xor trees. The inputs to the xor trees are selected with the multiplexers: the left input is for Grøstl and the right input for AES. The rightmost xor tree, for instance, computes $02b_0 \oplus 03b_1 \oplus 04b_2 \oplus 05b_3 \oplus 03b_4 \oplus 05b_5 \oplus 07b_6 \oplus 02b_7$ for Grøstl (with the left inputs) and $00b_0 \oplus 00b_1 \oplus 00b_2 \oplus 00b_3 \oplus 03b_4 \oplus 01b_5 \oplus 01b_6 \oplus 02b_7 = 03b_4 \oplus 01b_5 \oplus 01b_6 \oplus 02b_7$ for AES (with the right inputs). The multiplexers corresponding to the bolded values in (6) were removed; this saves 10 multiplexers. The multiplexers in the bottom are used for selecting whether the transformations are performed or not.

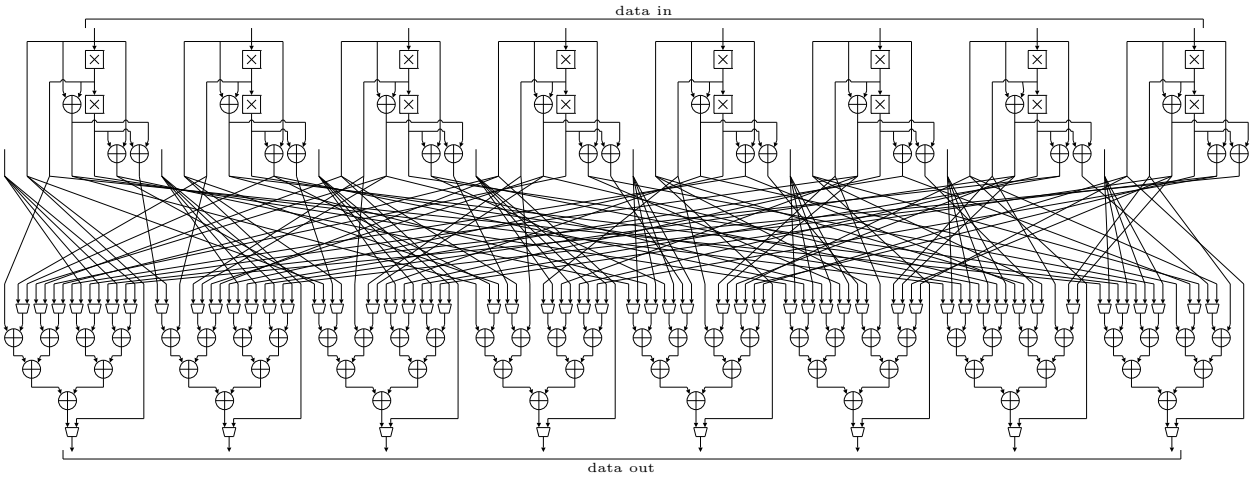


Fig. 5: Combined MixBytes and (two) MixColumns transformations.

**AddRoundKey** is implemented with the xors in the end of Fig. 4 where each byte is depicted separately (cf. Fig. 2b) in order to highlight how the same key is used for the parallel encryptions. The roundkey, $k$, is input from the left and each byte of the State is xorred with the corresponding byte of $k$. In Fig. 4, a single 128-bit round key is used for all four AES encryptions. However, the data path can be changed to support different keys for different encryptions without increasing the critical path (one xor). The key input is set to $k = 00 \ldots 00$ when the data path computes Grøstl.

### 6.3 KeyExpansion in the Data Path

This section describes how KeyExpansion is embedded into the data path by replacing one (or two) encryptions from the data path; hence, KeyExpansion is computed in parallel with the encryptions and it does not increase the latency of encryption. Also in this case, the round key computation uses SubBytes for four bytes (see Sect. 2.1) which can be shared with the Grøstl data path. The remaining of the round key computation differs significantly: Grøstl performs ShiftBytes and MixBytes transformations whereas KeyExpansion adds round constant, shifts bytes, and computes four 32-bit xors. Consequently, sharing resources beyond SubBytes would, again, be complicated; hence, other parts of KeyExpansion are computed with logic that is not shared with Grøstl. When the round key is ready, it is routed to AddRoundKey transformation of the three (or two) AES encryptions and stored to the register (e.g., into the upper-left (and upper-right) corner(s) of the State).

## 7 Results

The following eight designs were implemented based on the above described architectures.

`fugue` is the reference design of Fugue. It implements Fugue with an architecture and a data path similar to the ones described in Sect. 5.1 and 5.2, but without the support for AES. The implementation is comparable with the implementations presented in [3, 6]. This design is used for measuring the overheads of adding AES into the Fugue data path.

`fugue_aes` implements Fugue and AES encryption with the same data path. The design implements the architecture presented in Sect. 5.1 and 5.2. exactly as depicted in Figs. 2a and 2b.

`fugue_aes_ke` implements Fugue, AES encryption, and KeyExpansion with the same data path. The design implements the architecture presented in Sect. 5.1 and 5.2 with the modifications described in Sect. 5.3.

`groestl` is the reference design of Grøstl. It implements Grøstl with an architecture and a data path similar to the ones described in Sect. 6.1 and 6.2, but without the support for AES. The implementation is comparable with the FPGA implementations presented in [4, 7, 8]. This design is used for measuring the overheads of adding AES into the Grøstl data path.

`groestl_aes` implements Grøstl and one AES encryption with the same data path. The design implements the architecture presented in Sect. 6.1 and 6.2 so that support for AES is included only for the upper-left corner; all other parts support only Grøstl and are similar to `groestl`.

`groestl_aes_ke` implements Grøstl and one AES encryption with KeyExpansion in the data path. The design implements the architecture presented in Sect. 6.1 and 6.2 with the modifications described in Sect. 6.3. KeyExpansion and the AES State are placed into the upper-left and upper-right corners of the Grøstl State, respectively.

`groestl_4aes` computes Grøstl and four parallel AES encryptions with the same data path. The design implements the architecture presented in Sect. 6.1 and 6.2 exactly as depicted in Fig. 3–5. The design does not have KeyExpansion and it is assumed that round keys are fed into the design from an external source.

`groestl_3aes_ke` implements Grøstl, three parallel AES encryptions with the same key, and KeyExpansion computing the round keys on-the-fly. The design implements the architecture presented in Sect. 6.1 and 6.2 with the modifications described in Sect. 6.3. The key is input as a part of data in.

All designs were fit into the same 32-bit interface in order to make them compatible with the NIOS II (Altera's soft core processor) component interface. The designs were written in VHDL, simulated with ModelSim-Altera 6.5b, and synthesized for an Altera Cyclone III EP3C80F780C7 FPGA with Quartus II 9.1. The post-place&route results are collected in Tables 1 and 2, respectively for Fugue and Grøstl, together with performance values for both algorithms.

Table 1: Fugue results on Altera Cyclone III

| | `fugue` | `fugue_aes` | `fugue_aes_ke` |
|---|---|---|---|
| *Post-place&route results* | | | |
| Logic cells (LC) | 3562 | 4520 (+26.9 %) | 4875 (+36.9 %) |
| Registers | 1005 | 1105 (+10.0 %) | 1113 (+10.7 %) |
| $f_{\max}$ (MHz) | 63.93 | 60.75 (−5.0 %) | 59.81 (−6.4 %) |
| *Fugue performance* | | | |
| Latency (clock cycles) | 2 | 2 | 2 |
| Throughput (Gbps) | 1.023 | 0.972 | 0.957 |
| *AES performance* | | | |
| Latency (clock cycles) | – | 10 | 20 |
| Throughput (Gbps) | – | 0.778 | 0.383 |

Throughputs represent the theoretical maximum: they are computed using $f_{\max}$ and assume that there are no additional interfacing delays and that all parallel AES computations are always active. For Fugue and Grøstl, the latencies and throughput values exclude the final round and the output transformation, $\Omega$, which require 36 clock cycles and 11 clock cycles for Fugue and Grøstl, respectively. Hence, the maximum throughputs are valid for long messages only (the final round and $\Omega$ are insignificant).

## 8 Discussion

### 8.1 Comparison

The implementations show that adding one AES encryption into the data path is relatively cheap for both Fugue and Grøstl. The decreases in maximum clock frequencies are of the same magnitude for both algorithms. Area increase is smaller for Grøstl where adding one AES encryption costs approximately 300 LCs (or 2.5 %). The same value for Fugue is approximately 1000 LCs (or 26.9 %). For both algorithms the ability to share SubBytes with AES gives the most significant advantages because SubBytes is the main contributor

Table 2: Grøstl results on Altera Cyclone III

| | groestl | groestl_aes | groestl_aes_ke | groestl_4aes | groestl_3aes_ke |
|---|---|---|---|---|---|
| *Post-place&route results* | | | | | |
| Logic cells (LC) | 12086 | 12387 (+2.5 %) | 12520 (+3.6 %) | 13723 (+13.5 %) | 13453 (+11.3 %) |
| Registers | 1547 | 1550 (+0.2 %) | 1558 (+0.7 %) | 1550 (+0.2 %) | 1558 (+0.7 %) |
| $f_{\max}$ (MHz) | 57.52 | 54.13 (−5.9 %) | 55.79 (−3.0 %) | 56.03 (−2.6 %) | 53.36 (−7.2 %) |
| *Grøstl performance* | | | | | |
| Latency (clock cycles) | 20 | 20 | 20 | 20 | 20 |
| Throughput (Gbps) | 1.473 | 1.386 | 1.428 | 1.434 | 1.366 |
| *AES performance* | | | | | |
| Latency (clock cycles) | – | 10 | 10 | 10 | 10 |
| Throughput (Gbps) | – | 0.693 | 0.714 | 2.869 | 2.049 |

to both area and delay. Adding support for KeyExpansion and parallel AES encryptions is considerably cheaper and easier for Grøstl because its wider data path can be extended to support these operations in a straightforward manner. This allows high throughputs (2.9 Gbps) if several encryptions are performed with the same key. Adding KeyExpansion into the data path of Fugue halves the throughput of AES computations compared to the option where round keys are provided by an external source. Supporting parallel AES encryptions is difficult because SMIX, the AES-inspired transformation of Fugue, has only a 128-bit data path.

A straightforward iterative AES implementation with a 128-bit data path and KeyExpansion, which was implemented on the same FPGA, has area requirements of 2525 LCs and 527 registers (of which KeyExpansion takes 536 LCs and 136 registers). It runs on 65.56 MHz and receives a throughput of 839 Mbps. Hence, it is clear that sharing resources with the data path of the hash algorithm gives significant area savings with only small drops in throughputs for both Fugue and Grøstl.

As a cautionary note, we state that the Fugue and Grøstl implementations are not entirely comparable in the sense that they use different data path widths and, consequently, have signficantly different area requirements. However, the selected data path widths represent the most straightforward options for both algorithms. If, for example, we had implemented also Grøstl with a 128-bit data path, then one might argue that it gave an advantage for Fugue. Furthermore, implementations with different data path widths have been compared also in other works discussing hardware implementation of SHA-3 candidates; see, e.g., [6].

Both Fugue and Grøstl enjoy the advantage that AES can be efficiently supported in an implementation combining AES with the hash algorithm. This study suggests that Grøstl offers better combined implementations than Fugue. This is mainly caused by the facts that Grøstl uses a wide data path and a register with the same width, which allow computing several AES encryptions (and KeyExpansions) in parallel, and that basically all transformations of Grøstl are inspired by the transformations of AES which ensures that similarities can be exploited throughout the architecture; cf. only SMIX is shared in Fugue. However, the results are for one specific implementation platform and for specific data path widths. Therefore, other researchers are encouraged to study resource sharing of the SHA-3 candidates and AES on different platforms (both hardware and software) and with different design options, because such combined implementations would have significance in many applications.

### 8.2 Concluding Remarks

It was concluded in [20] that Fugue and Grøstl cannot take advantage of Intel's AES instruction set extension. In this respect, ECHO and SHAvite-3, which can take benefit of these instructions as such [20], have a clear advantage over the other candidates. However, in the light of this paper, it seems that it might be possible to add support for Fugue or Grøstl into various instruction set extension for AES, such as Intel's one, with reasonable overheads if one of them gets selected as SHA-3. It is impossible to say exactly what kind of changes would be required because the technical implementation details of Intel's AES instructions are not publically available. Nevertheless, this matter should be studied in the future.

The AES-inspired algorithms offer a unique opportunity to design custom hardware implementations that efficiently combine two different cryptographic functionalities into one design—something that has not been possible in this scale before. Even the combined implementations of MD5 and SHA-1 suffered from larger overheads than the implementations presented in this paper [12]. Hence, the possibility to efficiently combine a candidate with AES is an asset that should be taken into account when selecting SHA-3.

This paper presented the very first steps to this direction and it is easy to see that there is a lot of room for further research. At least the following issues should be studied further:

1. The applications, where the area constraints are strict, are likely to benefit the most from a combined implementation. Hence, it is of utmost importance to be able to reduce the footprint by using narrower data paths. Plenty of work has been done on reducing the footprint of AES, e.g. [23–25], and it would be interesting to study how those ideas could be combined with the ideas of this paper.
2. This paper discussed only basic iterative architectures. Computation speed could be substantially increased with unrolling and pipelining. At the first sight, the principles presented in this paper should be fairly easily exploited in implementations utilizing unrolling and/or pipelining but, nevertheless, this should be studied more deeply.
3. This paper discussed only certain variants of the algorithms. Adding support for other variants of AES encryptions is easy: only the number of rounds needs to be changed. However, KeyExpansion requires more significant changes; especially, more registers are needed for storing $K$. The ideas presented in this paper can be generalized for other variants of Fugue and Grøstl but the implementations may require significantly more area (especially, for Grøstl-384/512 because the size of the State doubles). Both Fugue-224 and Grøstl-224 can be supported with negligible changes. Adding support for AES decryption will be complicated because the designs of both Fugue and Grøstl are based on AES encryptions; e.g., InvSubBytes cannot be efficiently shared with SubBytes used in Fugue and Grøstl. Consequently, the ideas and implementations presented in this paper are advantageous primarily for applications using modes of operations that use block cipher encryptions also for decrypting data, such as the counter mode [26]. The counter mode is one of the prime candidates for combined Grøstl and AES implementations also because it allows different message blocks to be encrypted and decrypted in parallel. Anyway, other researchers are encouraged to study generalizations of the ideas to other variants of the algorithms.
4. Side-channel attacks are nowadays often regarded as the most serious threats against practical cryptosystems. This paper did not discuss countermeasures against side-channel attacks but it is clear that they deserve attention in the future. It seems likely that the vast amount of work done on side-channel countermeasures for AES could be utilized also in combined implementations of AES and AES-inspired hash algorithms.

## References


1. National Institute of Standards and Technology (NIST): Advanced encryption standard (AES). Federal Information Processing Standard, FIPS PUB 197 (2001)
2. Benadjila, R., Billet, O., Gilbert, H., Macario-Rat, G., Peyrin, T., Robshaw, M., Seurin, Y.: SHA-3 proposal: ECHO. Submission to NIST SHA-3 Competition (updated) (2009)
3. Halevi, S., Hall, W.E., Jutla, C.S.: The hash function "Fugue". Submission to NIST SHA-3 Competition (updated) (2009)
4. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST SHA-3 Competition (2008)
5. Biham, E., Dunkelman, O.: The SHAvite-3 hash function. Submission to NIST SHA-3 Competition (updated) (2009)
6. Tillich, S., Feldhofer, M., Kirschbaum, M., Plos, T., Schmidt, J.M., Szekely, A.: High-speed hardware implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein. Cryptology ePrint Archive, Report 2009/510 (2009)
7. Baldwin, B., Byrne, A., Hamilton, M., Hanley, N., McEvoy, R.P., Pan, W., Marnane, W.P.: FPGA implementations of SHA-3 candidates: CubeHash, Grøstl, Lane, Shabal and Spectral Hash. In: Proceedings of the 12th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD 2009), IEEE Computer Society (2009) 783–790
8. Jungk, B., Reith, S., Apfelbeck, J.: On optimized FPGA implementations of the SHA-3 candidate Groestl. Cryptology ePrint Archive, Report 2009/206 (2009)

9. Kobayashi, K., Ikegami, J., Matsuo, S., Sakiyama, K., Ohta, K.: Evaluation of hardware performance for the SHA-3 candidates using SASEBO-GII. Cryptology ePrint Archive, Report 2009/010 (2009)
10. Tillich, S., Feldhofer, M., Issovits, W., Kern, T., Kureck, H., Mühlberghuber, M., Neubauer, G., Reiter, A., Köfler, A., Mayrhofer, M.: Compact hardware implementations of the SHA-3 candidates ARIRANG, BLAKE, Grøstl, and Skein. Cryptology ePrint Archive, Report 2009/349 (2009)
11. Wang, M.Y., Su, C.P., Huang, C.T., Wu, C.W.: An HMAC processor with integrated SHA-1 and MD5 algorithms. In: Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC 2004), IEEE (2004) 456–458
12. Järvinen, K., Tommiska, M., Skyttä, J.: A compact MD5 and SHA-1 co-implementation utilizing algorithm similarities. In: Proceedings of the 2005 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2005), CSREA Press (2005) 48–54
13. Cao, D., Han, J., Zeng, X.Y.: A reconfigurable and ultra low-cost VLSI implementation of SHA-1 and MD5 functions. In: Proceedings of the 7th International Conference on ASIC (ASICON 2007), IEEE (2007) 862–865
14. Ng, C.W., Ng, T.S., Yip, K.W.: A unified architecture of MD5 and RIPEMD-160 hash algorithms. In: Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS 2004). Volume 2., IEEE (2004) 889–892
15. Ganesh, T.S., Frederick, M.T., Sudarshan, T.S.B., Somani, A.K.: Hashchip: A shared-resource multi-hash function processor architecture on FPGA. Integration, the VLSI Journal **40**(1) (2007) 11–19
16. Buchty, R., Heintze, N., Oliva, D.: Cryptonite — a programmable crypto processor architecture for high-bandwidth applications. In: Proceedings of the International Conference on Architecture of Computing Systems (ARCS 2004). Volume 2981 of Lecture Notes in Computer Science., Springer-Verlag (2004) 184–198
17. Wu, L., Weaver, C., Austin, T.: CryptoManiac: a fast flexible architecture for secure communication. In: Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA 2001), ACM Press (2001) 110–119
18. Grabher, P., Großschädl, J., Page, D.: Light-weight instruction set extensions for bit-sliced cryptography. In: Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2008). Volume 5154 of Lecture Notes in Computer Science., Springer-Verlag (2008) 331–345
19. Gueron, S.: Intel® advanced encryption standard (AES) instructions set. Intel Corporation, White paper (2010)
20. Benadjila, R., Billet, O., Gueron, S., Robshaw, M.J.B.: The Intel AES instruction set and the SHA-3 candidates. In: Advances in Cryptology—ASIACRYPT 2009. Volume 5912 of Lecture Notes in Computer Science., Springer-Verlag (2009) 162–178
21. Elbirt, A.J., Yip, W., Chetwynd, B., Paar, C.: An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **9**(4) (August 2001) 545–557
22. Canright, D.: A very compact S-box for AES. In: Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2005). Volume 3659 of Lecture Notes in Computer Science., Springer-Verlag (2005) 441–456
23. Feldhofer, M., Wolkerstorfer, J., Rijmen, V.: AES implementation on a grain of sand. IEE Proceedings — Information Security **152**(1) (2005) 13–20
24. Hämäläinen, P., Alho, T., Hännikäinen, M., Hämäläinen, T.D.: Design and implementation of low-area and low-power AES encryption hardware core. In: Proceedings of the 9th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD 2006), IEEE Computer Society (2006) 577–583
25. Rouvroy, G., Standaert, F.X., Quisquater, J.J., Legat, J.D.: Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for embedded applications. In: Proceedings of the International Conference on Information Technology: Coding and Computing, (ITCC 2004). Volume 2., IEEE Computer Society (2004) 583–587
26. National Institute of Standards and Technology (NIST): Recommendation for block cipher modes of operation. NIST Special Publication 800-38A (2001)