2010 International Conference on Reconfigurable Computing

On FPGA-based implementations of the SHA-3 candidate Grøstl

Bernhard Jungk and Steffen Reith
Hochschule RheinMain, Wiesbaden, Germany
Email: {bernhard.jungk, steffen.reith}@hs-rm.de

Abstract.—**The National Institute of Standards and Technology (NIST) has started a competition for a new secure hash standard. A significant comparison between the submitted candidates is only possible, if third party implementations of all proposed hash functions are provided.**

Our work is for the most part motivated by future developments of mass markets, where cryptographic infrastructures will become more and more important. One core component of such an infrastructure is a secure cryptographic hash function, which is used for a lot of applications like challenge-response authentication systems or digital signature schemes. We chose to evaluate the Grøstl hash function as one of the candidates heavily influenced by the AES algorithm, because there is reasonable hope to reduce the area of the cryptographic infrastructure by integrating AES and one of these hash algorithms on a FPGA. Hence, Grøstl serves as an example for the hash functions related to the AES approach. Our focus on low budget cryptographic solutions makes it natural to investigate possible optimizations for area efficient implementations, alongside our high-throughput variant.

Our results show, that - while Grøstl is inherently quite large compared to AES - it is possible to implement the Grøstl algorithm on small and low budget FPGAs like the second smallest available Spartan-3, while maintaining a reasonably high throughput.

Key words: Cryptography, Hash Function, Grøstl, SHA-3, FPGA Implementation

I. Introduction

The National Institute of Standards and Technology (NIST) has started a competition for a completely new hash function, very similar to the past AES competition (cf. [1]), to overcome the problems related to SHA-1 (cf. [2]) and the SHA-2 family (e.g. [3], [4]) of hash functions. Similar to the former effort, the rules of this competition require third party software and hardware implementations of all proposed candidates to evaluate the overall performance and resource requirements.

In the present paper, the focus lies on implementations of the SHA-3 candidate Grøstl (cf. [5]). The Grøstl hash function borrows many ideas from the Rijndael/AES algorithm (cf. [6]). This property makes Grøstl along with several other proposed hash functions an interesting SHA-3 candidate, because there is reasonable hope, that this will lead to area efficient implementations of cryptographic infrastructures, which contain both AES and a suitable hash function together. Moreover, it should be possible to adapt some well-known optimization techniques previously applied to the AES algorithm.

FPGA implementations of cryptographic primitives are advantageous, because they can offer better performance at a lower cost compared to software implementations or greater flexibility as custom ASIC chips. A very interesting and important application of cryptographic primitives are low-end and slow embedded platforms for the mass market (e.g. automotive or automation applications). Therefore one main goal of the present work is a compact implementation of Grøstl.

Two FPGA-based implementations were developed and evaluated to explore the possible area-throughput trade-off. Most of the applied optimizations are of architectural nature, reducing the number of LUTs by arranging the necessary registers, RAMs and logic or by pipelining. A more mathematical optimization approach uses composite field arithmetic to reduce the area of the S-box used in the Grøstl algorithm. This idea was first proposed by Rijmen in [7] and was subsequently investigated by many researchers (e.g. [8], [9], [10], [11]).

To our best knowledge, this work reports the smallest 256 bit implementations available[1], using only 967 slices on a Spartan-3 and 355 slices on a Virtex-5. This is a reduction of 71% compared to our high-throughput implementation for the Spartan-3 FPGA and 75% for the Virtex-5 implementation. Similar results are achieved for the 512 bit digests.

II. Previous work

The Grøstl algorithm is described in detail in [5], where results and estimates on several hardware (ASIC/FPGA) implementations are provided. Other recent hardware implementations are reported in [12], [13], [14], [15] and [16]. However, these results are only partially comparable to our work, because either they only report ASIC implementation results (cf. [12], [13]), they are not fully autonomous implementations (cf. [15]) or they do not focus on compact implementations (cf. [14]). Nevertheless, some ideas from the work on ASIC implementations are applicable to the present work, too. For example, the throughput of the serialized and hence smaller versions can be very similar to a fully parallel design, if the compression function is pipelined.

[1]Available at http://sha3.cs.hs-rm.de

The similarity between Grøstl and the AES cipher is beneficial for the optimization of the hash function, because some of the optimizations applied to AES (e.g. [17], [8], [18], [19], [20]) can be adapted to Grøstl. Good examples are the ideas for a compact AES implementation described in [18]. Especially the iterative design of this implementation can be applied to Grøstl after some modifications. Other examples are AES S-box optimizations (e.g. [8]).

Some optimizations are probably not very useful for Grøstl. For example, pipelined high-throughput implementations of AES often use the ECB mode, which does not depend on the output of the previous computation (e.g. [20]). In contrast, Grøstl uses a Merkle-Damgård construction (cf. [21], [5]), hence the output is fed back into the processing of the next message block and thus the usefulness of pipelining for high-throughput implementations is limited, especially in combination with unrolling the Grøstl rounds.

III. The Grøstl Hash Function

Following the Merkle-Damgård construction Grøstl consists of a padding function, a compression function f and the output transformation Ω. The input to the padding function is a message m of any size (in bits). The output is a padded message m', with size a multiple of l bits. The submission of Grøstl (cf. [5]) fixes l for 224 and 256 bit digests to $l = 512$ and for 384 and 512 bit digests to $l = 1024$.

The padding consists of three parts. The first part is a single bit, which is set to one. The second part consists of $|m| \bmod l - 65$ zeros, such that the message size will be 64 bits short of being a multiple of l. The third and last part is the number of message blocks $\#(m_i')$ encoded in 64 bits. Each message block m_i' is one part of the padded message m' with l bits.

After the padding, the compression function f will be executed for each message block m_i', where f uses the permutations P and Q to compute $h_i = f(h_{i-1}, m_i') = P(h_{i-1} \oplus m_i') \oplus Q(m_i') \oplus h_{i-1}$. The value h_0 is an initial value, depending on the desired hash length.

The permutations are both composed of four different sub-transformations `AddRoundConstant`, `SubBytes`, `ShiftBytes` and `MixBytes`. These sub-transformations are sequentially computed for n rounds, fixed to $n = 10$ for the 224 and 256 bit hashes and $n = 14$ for 384 and 512 bits. After each round, the output of the previous round is fed back as input for the next round. For the description of the sub-transformations it is convenient to map each message block to a matrix representation with eight rows and eight or sixteen columns, depending on l. Each entry of this matrix is one byte of the message block.

The number of already executed rounds is counted and used by the `AddRoundConstant` sub-transformation, which adds (XOR) the value of the counter `round` to an element in this matrix representation. The P-instance of this sub-transformations adds `round` to the first element in the first row, whereas the Q-instance adds `0xFF` $\oplus$ `round` to the first element in the eighth row.

The `SubBytes` sub-transformation is the exact same S-box used by the AES (cf. [6]). The `ShiftBytes` sub-transformation performs a cyclic left shift for each row. The first to the seventh row is shifted 0 to 6 columns to the left, whereas the eighth row is shifted 7 columns for $l = 512$, or 11 columns for $l = 1024$.

The `MixBytes` sub-transformation performs a matrix multiplication over the finite field $\mathbb{F}_{256}$:

$$m_i' \leftarrow \begin{pmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{pmatrix} \times m_i,$$

where m_i is the message block mapped to the matrix representation.

After all message blocks are compressed by f, the final output h_{last} is post-processed by the output transformation Ω, which first computes $P(h_{\text{last}}) \oplus h_{\text{last}}$ and then truncates the result to the desired digest size.

IV. Hardware Interface

One important aspect of hardware architectures is the interface. Especially for compact implementations, the interface may have a major impact on the overall area. Thus not all comparisons of hardware implementations are meaningful, if the interface differs.

The implemented interface is compliant to the Fast Simplex Link (FSL) specification (cf. [22]). The FSL is a popular method to connect IP cores to microprocessors, e.g. the Xilinx Microblaze softcore processor. The FSL interface is a generic 32 bit wide unidirectional link with an optional FIFO and optional clocks on the master and slave side, which then may be asynchronous. Two synchronous links form the complete bidirectional interface of our Grøstl implementations (see Tab. I).

Signal Name	I/O	Description
`FSL_Clk`	I	FSL Clock for synchronous FIFO mode
`FSL_Rst`	I	Peripheral reset
`FSL_M_Data`	O	Master input data (32 bits)
`FSL_M_Write`	O	Master writes data to the FIFO
`FSL_M_Full`	I	Master FIFO is full
`FSL_S_Data`	I	Slave output data (32 bits)
`FSL_S_Read`	O	Slave reads data from the FIFO
`FSL_S_Exists`	I	Data exists in the slave FIFO
`FSL_S_Control`	O	Control signal

Tab. I: Relevant parts of the FSL interface without the support for an asynchronous FIFO.

The incoming link (slave) is utilized to transfer the input to the Grøstl hash function in the following manner:

- Each input message block, consisting of $\{512, 1024\}$ bits, is sent through the 32 bit wide interface.
- Afterwards, the length of the message block is transfered as a 9 bit vector for 256 bit hashes or as a 10 bit vector for 512 bit hashes. If the padding function is not included, the `FSL_S_Control` signal is used to signal the end of the message instead.

If necessary, the message block has to be filled with zeros to be of the correct length. The output is handled analogous using the outgoing link (master) without sending the accompanying length information, because the length is fixed.

For the area and speed measurements, the FSL implementation, consisting of two FIFOs, is not included, because the implementation details are configurable and thus they vary greatly, depending on the requirements of the application.

V. Optimizations

A. High-throughput Implementation

The design of the high-speed implementation is a quite straight forward implementation of the Grøstl algorithm, with some optimizations applied (see Fig. 1).

The padding function receives all message blocks and passes them to the compression function f, padding the message blocks as necessary. The compression function then takes each message block and applies the P and Q permutations alternately. The sub-transformations are applied to the input message block for the first round and to the output of the previous round otherwise. When the last round is complete, a new value for h is computed and fed back into P combined (XORed) with the next message block.

After all message blocks are processed, the output transformation Ω is applied and the final hash value is placed in an output register. The output transformation reuses the instance of P of the compression function f by using a slightly modified padding and compression function, such that the computation of the output transformation before the truncation is equivalent to the output of the compression function $f_\Omega(h_{\text{last}}, m_\Omega) = P(h_{\text{last}} \oplus m_\Omega) \oplus Q(m_\Omega) \oplus h_{\text{last}}$. This works, if the output of the padding function is $m_\Omega = 0$ and if the output of Q is ignored. The modified compression function for the output transformation Ω is then $f_\Omega(h_{\text{last}}, m_\Omega) = P(h_{\text{last}}) \oplus h_{\text{last}}$.

Speeding up the execution of the interleaved P and Q transformations is possible by introducing a pipeline step in the round computation, hence the clock frequency can be effectively doubled, while maintaining the number of clock cycles for both permutations (cf. Fig. 1) and thus the throughput of the interleaved version is comparable to a parallel implementation, although it is much smaller.

B. Reduction of the Datapath Width

Additional area savings can be achieved, if the parallelism is limited even more, by reducing the datapath width. The general idea is to decompose the computation of a complete round into eight smaller parts. Thus only one eighth of the original S-boxes and `MixBytes` calculations are required for the 256 bit Grøstl variant, at the expense of an eightfold increase of clock cycles necessary for the computation of the complete compression function.

Our implementations consist of three main details:

- Usage of distributed RAM.
- An implicit `ShiftBytes` transformation.
- Pipelining of the round transformation.

We can use LUTs configured as $16/32$ bit deep and 32 bit wide distributed RAM instead of flip-flops, because the complete $512/1024$ bit state is never required in one clock cycle. For the Grøstl hash function, two memories are necessary, one for each permutation P and Q. Both RAMs consist of eight individual RAMs representing the rows of the state matrix (Fig. 2). The usage of the distributed RAM makes it possible to implement the `ShiftBytes` sub-transformation implicitly, by calculating appropriate read addresses.

The last important part of the optimization is the pipelining of the Grøstl round transformation. In addition to the speed-up, we gain additional area savings. This is only possible, if we add enough pipeline stages, to store the complete internal state in the pipeline, before the first part of the computation is completed. Then, we may read and write to the same addresses in the distributed RAM in each Grøstl round, which otherwise would not be possible and thus require an additional round counter as offset to the read and write addresses (cf. [23]).

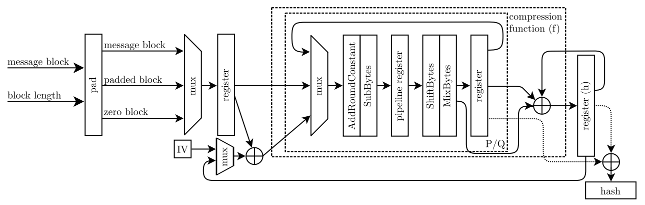

Fig. 1: The serialized Grøstl implementation.

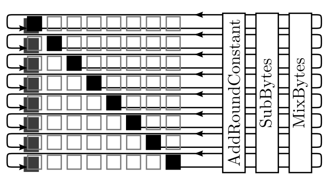

Fig. 2: RAM implementation for the P/Q permutations of the Grøstl round.

The optimization is similar to the one proposed for AES in [18]. The main difference is the removal of the second memory necessary for the proposed AES implementation, which results in a significant additional area reduction for Grøstl due to its large internal state.

An additional RAM of the same size is needed for the storage of the intermediate output h of the compression function, which is very similar to the other memories.

C. Pipelining

The already mentioned pipelining significantly increases the throughput and reduces the area, compared to the earlier implementation (cf. [23]). The pipelining itself can be considered in two independent parts:

- The padding function.
- The compression function.

For the round computation the padding function uses three pipeline steps to copy and pad the message to the compression function. First, we copy the message block from RAM to the first set of pipeline registers and increase the message block counter. Then, if applicable, we append the '1' bit and the first half of the counter. In the third clock cycle, the second half of the counter is appended. The counter is not appended in one clock cycle, because it would need a larger number of multiplexers with a longer logic delay.

The more obvious place to introduce a pipeline is the compression function. The best number of additional pipeline steps is 8 for the 256 bit implementation (16 for the larger variant). This choice follows from the decomposition of the complete Grøstl state in eight sub-states and the interleaving of the P and Q instances. The eight sets of pipeline registers are evenly distributed throughout the Grøstl round, most of them are inserted in our decomposed S-Box implementation, which makes it possible to have less logic delay per pipeline step. The actual distribution of the pipeline registers is rebalanced by the Xilinx Synthesis Toolkit (XST) to achieve better timings.

D. Optimized S-box

The optimization of the S-box is based on finite field arithmetic, which is used to calculate each value on-the-fly instead of the usage of a lookup table. The basic idea is a change of the representation of each finite field element to a computationally more efficient one. This change works, because all finite fields with the same cardinality are isomorphic (cf. [24], Theorem 2.5).

The optimization decomposes the finite field $\mathbb{F}_{256}$, defined by the AES polynomial, using the fields $\mathbb{F}_{2^2}$, $\mathbb{F}_{(2^2)^2}$ and $\mathbb{F}_{((2^2)^2)^2}$. In this new representation, the inversion of an element of $\mathbb{F}_{256}$, which is necessary for the calculation of the `SubBytes` sub-transformation, uses arithmetic operations in the sub-fields, resulting in a smaller implementation.

Canright [8] and others (e.g. [9], [10], [11]) have explored this optimization technique quite in-depth for ASIC implementations. To our knowledge there is no such work specific for FPGAs. The work for ASICs counts the number of elementary binary logic gates (e.g. XORs and ANDs in [8]). However, the LUTs used by FPGAs implement larger Boolean functions and thus, the previous measurements are only partially applicable to FPGAs because of the technological differences.

	Digest	**Slices**	**Frequency (MHz)**	**Throughput (MBit/s)**
new	256	1125	183	243
see [8]	256	1179	183	230

Tab. II: Compact Grøstl implementations (Spartan-3)

Hence, instead of just taking some of the previous results, we built an optimization framework, which optimizes the finite field arithmetic for LUT-based FPGAs. Following some basic assumptions of Canright (cf. [8]) the AES field is similarly decomposed into the field $\mathbb{F}_{((2^2)^2)^2}$ using the following choices for irreducible polynomials. Over $\mathbb{F}_2$ there is only one irreducible polynomial $f(x) = x^2 + x + 1$, over $\mathbb{F}_{2^2}$, there are two irreducible polynomials of the form $g(y) = y^2 + y + u$ and over $\mathbb{F}_{(2^2)^2}$ eight polynomials of the form $h(z) = z^2 + z + v$, combined this results in 432 possible representations[2].

All of these 432 different representations were analyzed for LUT-based FPGAs. The optimization framework creates optimized VHDL code for the conversion matrices and the arithmetic operations in $\mathbb{F}_{2^2}$, $\mathbb{F}_{(2^2)^2}$ and $\mathbb{F}_{((2^2)^2)^2}$. The VHDL code for each representation is then synthesized and further optimized using the Xilinx toolchain.

The optimizations applied by the optimization framework are similar to the high level optimizations described in [8]. However, the final results after the placement and routing are dependent on the optimizations applied by the Xilinx toolchain. This might be more controllable by extending the optimization framework to be able to generate `LUT4`/`LUT6_2` instances (cf. [25]) instead of the current high level VHDL code generation.

For comparison we synthesized Grøstl with the S-box of Canright (cf. [8]). A fair comparison is achieved, by synthesizing all implementations with the Xilinx toolchain. The throughput is similar, but the new S-box implementation results in a slightly smaller overall area (Tab. II).

VI. Evaluation

We have implemented several different variants and synthesized each of them for Spartan-3 and Virtex-5 FPGAs. The main differences between the implemented variants (see Tab. III and Tab. V) are the S-box implementations and the datapath width. The high-throughput variants use the maximum width datapath and implement the S-box as BRAM.

[2]For each irreducible polynomial, there are two polynomial and one normal bases, thus we have $3^3 \cdot 2 \cdot 8 = 432$ possible representations.

Digest	Datapath	Slices	BRAM	MHz	MBit/s	MBit/s/Slice
512	1024	6807	63	95	3474	0.51
		6150	63	100	3657	0.59
256	512	3692	31	100	2560	0.69
		3367	31	108	2764	0.82
512	64	1372	0	174	397	0.29
		1179	0	177	404	0.34
256	64	1125	0	183	585	0.52
		967	0	182	582	0.60

Tab. III: Implementation results for Spartan-3 FPGAs.

Reference	Digest	Slices	BRAM	MHz	MBit/s	MBit/s/Slice
[5]	512	20233	n/a	80.7	5901	0.29
[5]	256	6582	n/a	86.7	4439	0.67

Tab. IV: Third party results for Spartan-3 FPGAs.

Digest	Datapath	Slices	BRAM	MHz	MBit/s	MBit/s/Slice
512	1024	3120	33	241	8813	2.82
		2692	34	246	8996	3.34
256	512	1708	17	302	7731	4.52
		1381	17	295	7552	5.46
512	64	644	0	334	763	1.18
		604	0	331	756	1.25
256	64	470	0	354	1132	2.40
		355	0	357	1142	3.21

Tab. V: Implementation results for Virtex-5 FPGAs.

Reference	Digest	Slices	BRAM	MHz	MBit/s	MBit/s/Slice
[14]	512	4525	0	113	3619	0.79
		4845	0	123		
[5]	512	5419	n/a	210.5	15395	2.84
[15]	512	3155	n/a	325	11798	3.73
[16]	256	2616	n/a	154	7885	3.01
[5]	256	1722	n/a	200.7	10276	5.96
[15]	256	1716	n/a	350	8545	4.97
[14]	256	2579	0	78	3242	1.25
		2391	0	101		

Tab. VI: Third party results for Virtex-5 FPGAs.

The compact implementations (Tab. III) have a reduced size datapath and use the optimized S-box. Furthermore, we have removed the padding functionality to explore the impact of the padding on the area and the clock frequency. The implementation results without the padding are given in the second row of each variant in Tab. III and Tab. V. We did not study parallel implementations of the P and Q permutations, because the interleaved implementations have roughly the same throughput and require much less area than the parallel versions. A similar effect was already observed in [13] for ASIC implementations.

One of the most obvious fact of our results shows, that the padding function has a significant influence on the overall size and the throughput-area ratio. For the Spartan-3 implementations about 9% to 15% of the area is consumed by the padding function, for the Virtex-5 implementations between 7% and almost 25%. The throughput-area ratio is even more influenced by the padding, when the clock frequency improves, which is not always the case. These deviations of the clock frequency are most likely caused by the Xilinx toolchain and not inherently linked to the padding.

Comparing the fastest and smallest implementations, our compact implementations are clearly much smaller (about 20% to 30% of the high throughput implementations). Unfortunately the throughput drops to about 9% to 10% for 512 bit digests and to about 10% to 15% for 256 bit digests. Therefore, the throughput-area ratio of the compact implementations is worse compared to the ones designed for high-throughput. The drop for the 512 bit implementations is more substantial because it needs 448 clock cycles for the 14 rounds (14 rounds, 16 sub-rounds for P and the same amount for Q, thus $14 \cdot 16 \cdot 2 = 448$), whereas the 256 bit implementations only require 160 clock cycles (10 rounds, 8 sub-rounds for P and the same amount for Q, thus $10 \cdot 8 \cdot 2 = 160$). Note, that the high-throughput implementations use block RAM and thus the comparison is not entirely fair.

Tab. IV and Tab. VI show some results of other FPGA implementations known to us. The fastest implementations are reported in [5]. Since the submission document reveals not enough details, we do not know where the differences are. The results from [15] do not implement the padding function. They are bigger and faster than the new implementations of our work. Beside these differences, the throughput-area ratio is about the same magnitude, therefore it is not unlikely, that the implementations are similar. The implementations from [14] do not use block RAM and thus they necessarily consume more area and are slower. For the result reported in [16], it is likely, that the implementation does not use block RAM, because the implementation consumes an amount of slices very similar to the 256 bit implementation from [14] and the report does not say anything about the usage of block RAM. Overall, all third party implementations except the implementations from [14] perform very similar.

Looking at the throughput-area ratio for Spartan-3 implementations, we can see an improvement by the new implementations over existing ones, especially for the 512 bit variant. For Virtex-5 FPGAs, the result for 256 bit hashes from the submission document [5] has still the best throughput-area ratio. If we discount the padding function, our 256 implementation is on the second place and about 20% smaller. For the compact implementation, no meaningful comparison can be achieved for obvious reasons.

VII. Conclusion and Further Work

The present paper focuses on FPGA implementations of the SHA-3 candidate Grøstl. Several optimized variations were implemented and evaluated. Overall the Grøstl-256 hash function fits on small sized FPGAs like the Spartan-3 XC3S200. Additional reduction of the area is possible through further reduction of the datapath width (8 bit) and thus Grøstl may fit on the smallest Spartan-3 FPGA. Our experience with the current implementation suggests, that the throughput may still be reasonable, if enough pipeline steps are introduced. However, comparing the Grøstl implementations to a compact AES implementation (e.g. 222 slices on a Spartan-2 [18]) shows, that Grøstl will probably remain rather area consuming, because of its larger internal state.

Acknowledgment

We would like to thank Jürgen Apfelbeck for his help and insightful comments on various aspects of this work. This research was supported in part by BMBF grant 17N1308.

References

[1] R. F. Kayser, "Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family," in *Federal Register*. National Institute of Standards and Technology, November 2007, vol. 72, pp. 62 212–62 220.

[2] X. Wang, Y. L. Yin, and H. Yu, "Finding Collisions in the Full SHA-1," in *Proceedings of Crypto*, ser. Lecture notes in computer science, vol. 3621. Springer, 2005, pp. 17–36.

[3] S. Sanadhya and P. Sarkar, "New collision attacks against up to 24-step SHA-2," in *Progress in Cryptology-INDOCRYPT*, ser. Lecture notes in computer science, vol. 5365. Springer, 2008.

[4] T. Isobe and K. Shibutani, "Preimage attacks on reduced Tiger and SHA-2," in *Fast Software Encryption*, ser. Lecture notes in computer science, vol. 5665. Springer, 2009.

[5] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen, "Grøstl – a SHA-3 candidate," Submission to NIST, 2008. [Online]. Available: http://groestl.info/Groestl.pdf

[6] V. Rijmen and J. Daemen, *The Design of Rijndael*. Springer, 2002.

[7] V. Rijmen, "Efficient implementation of the Rijndael S-Box," Katholieke Universiteit Leuven, Tech. Rep., 2000.

[8] D. Canright, "A Very Compact S-Box for AES," in *Proceedings of 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2005, pp. 441–455.

[9] X. Zhang and K. K. Parhi, "On the Optimum Constructions of Composite Field for the AES Algorithm," in *IEEE Transactions on Circuits and Systems*, vol. 53, 2006, pp. 1153–1157.

[10] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede, "A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box," in *Topics in Cryptology - CT-RSA 2005*, ser. Lecture Notes in Computer Science, vol. 3376. Springer-Verlag, 2005, pp. 232–333.

[11] S. Nikova, V. Rijmen, and M. Schläffer, "Using Normal Bases for Compact Hardware Implementations of the AES S-Box," in *Security and Cryptography for Networks*, ser. Lecture Notes in Computer Science, vol. 5229. Springer-Verlag, 2008, pp. 236–245.

[12] S. Tillich, M. Feldhofer, W. Issovits, T. Kern, H. Kureck, M. Mühlberghuber, G. Neubauer, A. Reiter, A. Köfler, and M. Mayrhofer, "Compact Hardware Implementations of the SHA-3 Candidates ARIRANG, BLAKE, Grøstl, and Skein," Cryptology ePrint Archive, Report 2009/349, 2009.

[13] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely, "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.

[14] B. Baldwin, N. Hanley, M. Hamilton, L. Lu, A. Byrne, M. O'Neill, and W. Marnane, "FPGA Implementations of the Round Two SHA-3 Candidates," The second SHA-3 Candidate Conference, 2010.

[15] K. Gaj, E. Homsirikamol, and M. Rogawski, "Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," Cryptology ePrint Archive, Report 2010/445, 2010.

[16] S. Matsuo, M. Knežević, P. Schaumont, I. Verbauwhede, A. Satoh, K. Sakiyama, and K. Ota, "How Can We Conduct "Fair and Consistent" Hardware Evaluation for SHA-3 Candidate?" The second SHA-3 Candidate Conference, 2010.

[17] D. Canright and D. A. Osvik, "A More Compact AES," *Selected Areas in Cryptography: 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*, pp. 157–169, 2009.

[18] P. Chodowiec and K. Gaj, "Very compact FPGA implementation of the AES algorithm," in *Proceedings of 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2003, pp. 319–333.

[19] N. Pramstaller, S. Mangard, S. Dominikus, and J. Wolkerstorfer, "Efficient AES Implementations on ASICs and FPGAs," in *Advanced Encryption Standard – AES*. Springer-Verlag, 2005, pp. 98–112.

[20] M. McLoone and J. McCanny, "High Performance Single-Chip FPGA Rijndael Algorithm Implementations," in *Proceedings of 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. London, UK: Springer-Verlag, 2001, pp. 65–76.

[21] A. Regenscheid, R. Perlner, S. jen Chang, J. Kelsey, M. Nandi, and S. Paul, "Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition," NIST, Tech. Rep., 2009.

[22] *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)*, Xilinx, 2010.

[23] B. Jungk and S. Reith, "On FPGA-based implementations of Grøstl," Cryptology ePrint Archive, Report 2010/260, 2010.

[24] R. Lidl and H. Niederreiter, *Finite Fields (Encyclopedia of Mathematics and its Applications)*. Cambridge University Press, 1996.

[25] *Spartan-3 Generation FPGA User Guide*, Xilinx, 2009.