On FPGA-based implementations of Grøstl*

Bernhard Jungk and Steffen Reith

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim Geisenheim
{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. To make a significant comparison between the submitted candidates, third party implementations of all proposed hash functions are needed. This is one of the reasons why the SHA-3 candidate Grøstl has been chosen for a FPGA-based implementation.
Mainly our work is motivated by actual and future developments of the automotive market (e.g. car-2-car communication systems), which will increase the necessity for a suitable cryptographic infrastructure in modern vehicles (cf. AUTOSAR project) even further. 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. Another motivation to evaluate Grøstl is its resemblance to AES.
The automotive market demands, like any mass market, low budget and therefore compact implementations, hence our evaluation of Grøstl focuses on area optimizations. It is shown that, while Grøstl is inherently quite large compared to AES, it is still possible to implement the Grøstl algorithm on small and low budget FPGAs like the second smallest available Spartan-3, while maintaining a reasonable high throughput.

Key words: Cryptography, hashfunction, Grøstl, FPGA, automotive, car2car

1 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.

* Research supported in part by BMBF grant 17N1308.

In the present paper, the focus lies on implementations of the SHA-3 candidate Grøstl (cf. [5]), because the Grøstl hash function borrows many ideas from the Rijndael/AES algorithm (cf. [6,7]). This property makes Grøstl an interesting SHA-3 candidate, because there is reasonable hope, that this will lead to area/space efficient implementations of cryptographic infrastructures, which contain both AES and a suitable hash function together. Moreover it should be possible to adopt well-known optimization techniques for the AES algorithm.

FPGA implementations of cryptographic primitives are interesting, 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). Augmenting these slow embedded processors with dedicated hardware solutions boosts the performance to a level, where for example secure car-2-car communication with higher bandwidth demands becomes possible. Therefore the focus of the present work lies on compact implementations and thus optimizations, which are specifically designed to improve the throughput, are not investigated.

Several FPGA-based implementations were developed and evaluated to give a first exploration of the area-throughput trade-off. The applied optimizations come in two flavors. The first kind of optimizations are of architectural nature, which reduce the number of LUTs by arranging the necessary registers, RAMs and logic. The other optimization approach uses composite field arithmetic to reduce the area requirement for the S-box used in the Grøstl algorithm. This idea was first proposed by Rijmen in [8] and was subsequently investigated by many researchers (e.g. [9,10,11,12]).

Our results (cf. Tab. 2) present a first look at the achievable trade-off between throughput and area consumption for Grøstl, where the smallest implementation for 224/256 bit digests (1276 slices) is about 78% smaller than our high-throughput implementation, but also 93% slower and hence fits on a small Spartan-3 FPGA (XC3S200).

2 Previous work

The Grøstl algorithm is described in detail in [5], where additional results and estimates on several hardware (ASIC/FPGA) implementations are provided. Other recent hardware implementations are reported in [13,14] and [15]. However, these results are hardly comparable to our work, because either they only report ASIC implementation results (cf. [13,14]) or they

are not fully autonomous implementations (cf. [15]). 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 the fully parallel design, if the compression function is pipelined.

The FPGA implementations presented in [15] show some results on the trade-offs between area and throughput, but unfortunately they do not implement the padding function (cf. Sec. 3) and hence a fair comparison to our results is not possible.

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. [16,9,17,18,19]) can be easily adapted to Grøstl. A good example are the ideas for a compact AES implementation described in [17]. Especially the iterative design of this implementation can be applied to Grøstl after some modifications. Another example are the AES S-box optimizations (cf. [9]).

Other 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. [19]). In contrast, Grøstl uses a Merkle-Darmgård construction (cf. [20,5]), hence the output is fed back into the processing of the next message block and thus the usefulness of pipelining for Grøstl is limited.

3 The Grøstl Hash Function

Following the Merkle-Darmgård 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 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 $f(h, m'_i) = P(h \oplus m'_i) \oplus Q(m'_i) \oplus h$. The value h is either an initial value,

depending on the desired hash length, if it is the first message block or the output of the previous computation of f.

The permutations are each composed of four different sub-transformations `AddRoundConstant`, `SubBytes`, `ShiftBytes` and `MixBytes`. These sub-transformations are sequentially computed for n rounds, where $n = 10$ for the 224 and 256 bit hashes and $n = 13$ 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. [6,7]). 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 x of f is post-processed by the output transformation Ω, which first computes $P(x) \oplus x$ and then truncates the result to the desired digest size.

4 Optimizations

All of the presented Grøstl implementations share a common structure (see Fig. 1), which will be described before delving into optimizations. The design is basically a straight forward implementation of the Grøstl algorithm, with some fairly easy general optimizations applied.

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. 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 tranformation is applied and the final hash value is placed in an output register. The output transformation reuses the instance of P in the compression function f.

4.1 General Architectural Optimizations

The first optimization idea is the reuse of the P-instance used by the compression function f in the output transformation Ω. From the specification of the Grøstl hash function, we have $\Omega(h) = P(h) \oplus h$ and $f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h$. Thus, we can achieve the reuse of P by changing the padding function to output an all-zero message block after the last message block and by ignoring the output of $Q(m)$.

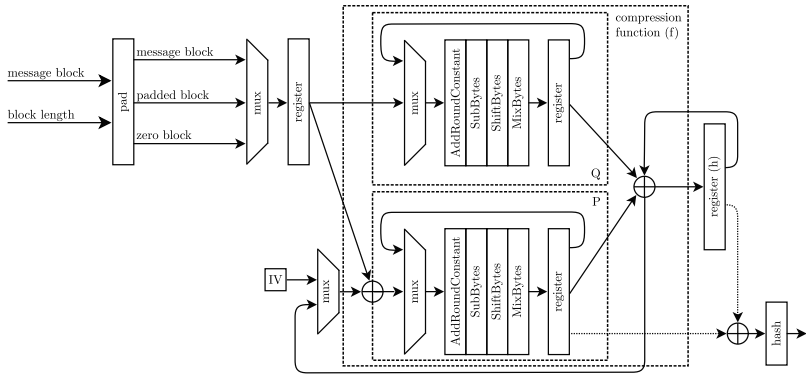


Fig. 1: The general design of the Grøstl implementations.

Changing Q to $Q_\Omega(m) = 0$, such that $P(h \oplus m) \oplus Q_\Omega(m) \oplus h = P(h) \oplus h$ for the computation of Ω does not reduce the area further, because both approaches require the same number of LUTs (cf. [21]).

Another general area optimization technique is to reduce the parallelization by serializing the execution of P and Q (see Fig. 2). This will half the number of S-boxes and `MixBytes` instances and thus reduce the necessary area at the cost of doubling the number of clock cycles to compute the compression function.

The performance can be boosted again, by introducing a pipeline step in the round computation, hence the clock frequency can be much higher, while maintaining the number of clock cycles for both permutations (cf. Fig. 2).

4.2 Reduction of the Datapath Width

Another way to further reduce the parallelism and hence the area is a reduction of the datapath width. The general idea is to reduce the computation of a complete round in eight smaller parts. Thus only one eighth of the original S-boxes and `MixBytes` calculations are required, at the expense of an eightfold increase of clock cycles for the computation of the compression function.

The approach is similar to the compact AES implementation proposed in [17]. One of the proposed optimizations is an area reduced implementation of the input and output memories needed for the Grøstl round. The first technique is based on distributed RAM, the second uses shift registers. Both can be implemented with LUTs. Adapting their idea to Grøstl, two memories of the necessary size hold the internal state of the Grøstl compression function. The first one implements the input register, the other one the output register. Input and output memories are swapped after each round.

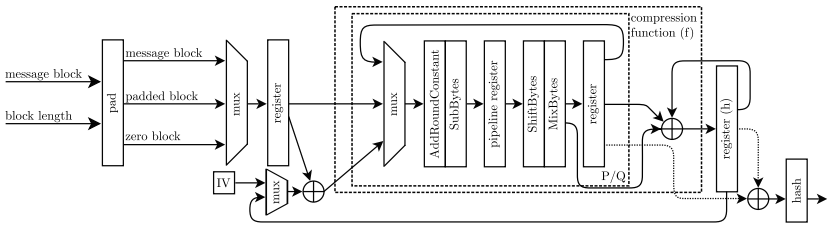


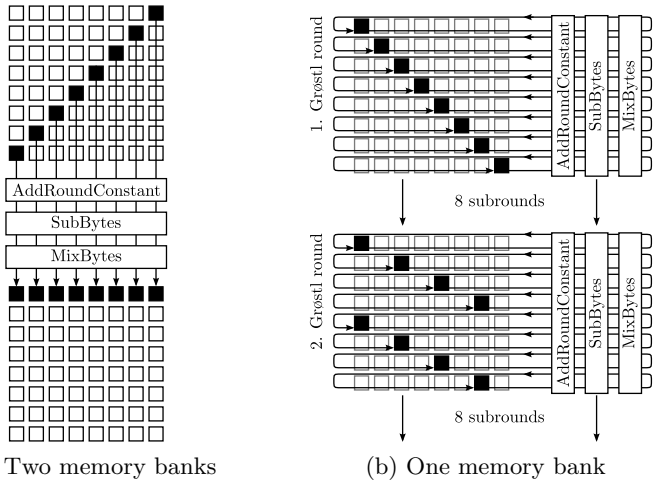
Fig. 2: The serialized Grøstl implementation.

For the input, it is necessary to tap the bytes out of the input register, according to the `ShiftBytes` transformation, thus the explicit `ShiftBytes` transformation is unnecessary. The tapping is achieved by using a counter for the sub-rounds and offsets for each matrix row. The output is written consecutively to the same address for each matrix row (see Fig. 3a).

We take this idea further by removing the second memory bank (see Fig. 3b). We benefit from a feature of dual-ported distributed RAM, which allows us to read and write to the same address in the same clock cycle using a pipelined RAM (cf. [22]). The new idea works as follows:

i) Calculate the current read/write addresses by adding the sub-round counter, the row offset and an additional round offset.
ii) Read the new inputs from these addresses.
iii) Write the current results to the same addresses.

The critical part in this procedure is the calculation of the addresses. In contrast to the previous case, we have to add additional offsets for each round of the compression function, to simulate the `ShiftBytes` sub-transformation. A very similar idea may be adapted for the storage of the output h of the compression function f and the subsequent feedback of h to the computation of the next message block. First we read the necessary part of the old value of h from the memory. Then we calculate the corresponding part of $P(h \oplus m_i) \oplus Q(m_i) \oplus h$ and write the result back to the same memory. This construction can be implemented easier than the previous. We have two reads, the first in the first round is done



(a) Two memory banks (b) One memory bank

Fig. 3: Different I/O register implementations for the Grøstl round.

according to `ShiftBytes` similar to the way it is done for the case with two memories. The second time, we read in the last round to calculate the new value. Here, we do not need a special treatment for the `ShiftBytes` sub-transformation.

We could further reduce the area, by using an 8 bit wide datapath, thus each round takes 64 clock cycles for each permutation P and Q. The main obstacle is the `MixBytes` sub-transformation, where the output of a single byte depends on 8 input bytes, which makes it more difficult to implement than the 64 bit wide datapath. However, this design has probably a extremely low throughput compared to the other implementations, hence it is not considered here.

The performance loss caused by the reduction of the datapath width could be mitigated by the introduction of more pipelining registers in the compression function, e.g. the 64 bit datapath could be implemented using a pipeline with depth 8 and therefore almost reach the original throughput. The area would not increase significantly, because the flip-flops implementing the pipeline registers could probably be placed in the same slice as the logic (cf. [21]).

4.3 Optimized S-Box

The area reduction of the S-box optimizes the underlying finite field arithmetic, which is used to calculate each value on-the-fly instead of the usage of a S-box lookup table. The basic idea is to change the representation of each finite field element to a computationally more efficient representation. This change of representation works, because it is well known, that all finite fields with the same cardinality are isomorphic (e.g. [23], 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 [9] and others (e.g. [10,11,12,24]) have explored this optimization technique quite in-depth for ASIC implementations. To our knowledge there is no such work specific for FPGAs. The main relevant difference between ASICs and FPGAs is the way how the logic is implemented. The previous work counts the number of elementary binary logic gates (e.g. XORs and ANDs in [9]). This is an adequate measurement for the area of ASIC implementations, but not necessarily if we target FPGAs.

This inadequacy is the result of the following ideas. There is an interesting result regarding the mapping of Boolean circuits to LUT-based

FPGAs. The minimal technological mapping without duplication of gates is known to be solvable in polynomial time. However, this mapping is not the global minimum for FPGA implementations, because duplication of logic gates may further decrease the area (cf. [25]). This result suggests, that some of the low-level optimizations proposed in [9] are counter-productive. Furthermore, every n-ary Boolean function ($n \in \{4, 5, 6\}$ depending on the LUT), can be implemented with the same cost, hence a differentiation between gate types is unnecessary. Therefore, a different choice of basis could be better for FPGA implementations.

Instead of just taking some of the previous results, we built a optimization framework, which optimizes the finite field arithmetic for LUT-based FPGAs. Following some basic assumptions of Canright ([9]) 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 $h(z) = z^2 + z + v$. These choices of irreducible polynomials result in 432 possible representations.

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 (cf. Fig. 4).

The optimizations applied by the optimization framework are similar to the high level optimizations described in [9]. However, the final results after the placement and routing are dependent on the optimizations applied by the Xilinx toolchain. This may be more controllable by extending the optimization framework to be able to generate `LUT4` instances (cf. [21]), instead of the current high level VHDL code generation.

For comparison we synthesized Grøstl with the S-box of of Canright (cf. [9]) and an optimized Boolean circuit, which is an even further opti-

	Digest	**Slices**	**Frequency (MHz)**	**Throughput (MBit/s)**
new	224/256	1672	38	243
see [9]	224/256	1693	36	230
see [24]	224/256	1684	31	198

Tab. 1: Grøstl implementation results with very compact S-box optimizations for Spartan-3 FPGAs.

mized version of Canright's result (cf. [24]). A fair comparison is achieved, by synthesizing all implementations with the Xilinx toolchain and the same optimization options. Tab. 1 illustrates the post place and route results of the 64 bit datapath P/Q-parallel Grøstl implementations using the new optimized S-box and the other optimized versions.

We can see, that the results are similar, but anyhow the new S-box implementation[1] slightly outperforms both other optimizations. The generated solutions were not examined in detail, hence one possible assumption is, that some of the low level optimizations applied by Canright are more difficult to optimize for the Xilinx toolchain. A similar assumption can be made for the decreased throughput of the implementation based on the result from [24], which is probably caused by an increased logic depth.

For further comparison, Fig. 4 shows all results for the 432 representations. We can see, that only a few representations result in a very small implementation, whereas most representations are rather mediocre and some are worse. Hence, it is important to choose the representation carefully. Furthermore, the plot includes our optimization of Canright's minimal representation and the optimizations presented in [9] and [24].

[1] Let $\alpha = x$ and $\beta = y$ be the roots of $x^2 + x + 1$ and $y^2 + y + u$, then $u = \alpha^2$ and $v = \alpha\beta + \alpha$. Two bases are normal, for $\mathbb{F}_{(2^2)^2}$ we have a polynomial basis.

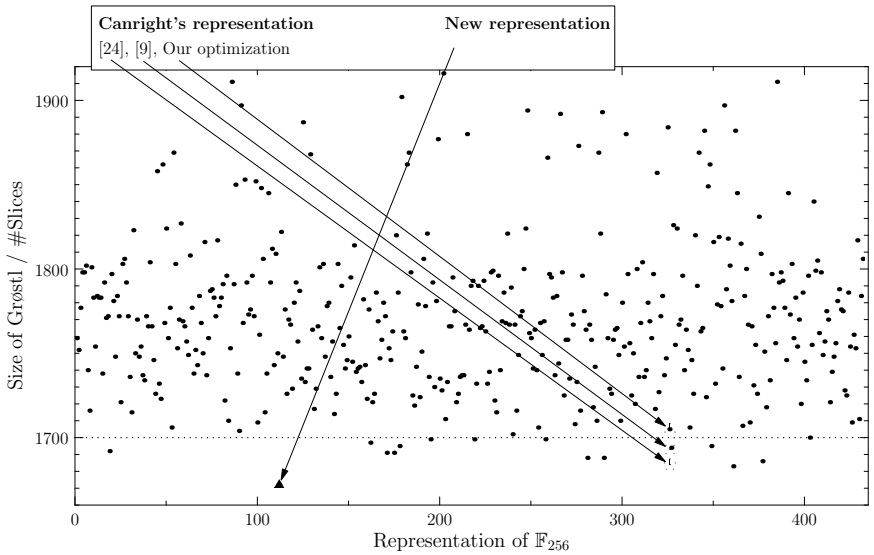


Fig. 4: Size of VHDL-based implementations for all 432 representations using the Xilinx toolchain.

5 Evaluation

We have implemented several different variants and synthesized each of them for a Spartan-3. We did not implement the design for Virtex-5 or other high-end FPGAs, because of the focus on low budget and compact implementations.

The main differences between the implemented variants (see Tab. 2) are the S-box implementations and the datapath width. The high-throughput variants (Tab. 2 (1-3)) use the maximum width datapath and implement the S-box as BRAM. The compact implementations (Tab. 2 (4-7)) have a reduced size datapath and use the optimized S-box (cf. Sec. 4.3).

Some general observations about the presented implementations can be made. A fully parallel implementation for 384/512 bit digests for the Spartan-3 cannot be achieved, because the FPGA does not have enough BRAM cells. Furthermore, the serialized versions are almost as fast as the parallel implementations. A similar effect was already observed in [14].

Comparing our fastest (Tab. 2 (2)) and smallest (Tab. 2 (7)) Spartan-3 implementations for 224/256 bit digests, the area reduces to 1276 slices which is about 22% of the high-throughput implementation. Hence, the compact implementation fits on a small Spartan-3 FPGA (XC3S200). At the same time the throughput drops to about 7%. This is a quite significant loss of performance compared to the achievable area reduction, but it is probably not critical for low budget implementations.

Tab. 3 shows the results of other FPGA implementations known to us. Most of our implementations are slower compared to these third party results, which was expected due to the focus on compact implementations. As previously mentioned, a detailed comparison with the results of [15] is not possible, because of the missing padding function. A comparison

	Digest	**P/Q**	**Slices**	**BRAM**	**MHz**	**MBit/s**
1024 bit datapath						
1	384/512	serial	8308	64	95	3474
512 bit datapath						
2	224/256	parallel	5693	64	54	2764
3	224/256	serial	4491	32	100	2560
64 bit datapath						
4	384/512	parallel	2463	0	36	164
5	384/512	serial	2110	0	63	144
6	224/256	parallel	1672	0	38	243
7	224/256	serial	**1276**	0	60	192

Tab. 2: Implementation results for Spartan-3 FPGAs.

	Reference	Digest	P/Q	Slices	BRAM	MHz	MBit/s
1024 bit datapath							
8	[5]	384/512	parallel	20233	n/a	80.7	**5901**
9	[15]	384/512	serial	6313	n/a	79.61	2910
512 bit datapath							
10	[5]	224/256	parallel	6582	n/a	86.7	4439
11	[15]	224/256	serial	3183	n/a	91.02	2330
64 bit datapath							
12	[5]	224/256	parallel	3000-4000	n/a	75-125	400
13	[5]	384/512	parallel	6000-8000	n/a	35-60	300

Tab. 3: Third party results and estimates for Virtex-2P (12, 13) and Spartan-3 FPGAs (8-11).

to the results of [5] reveals that the new high-throughput variants are significant smaller than the results reported in [5], but also have lower throughput. This problem is due to very high routing delays, which are more than 50% of the overall delay in the longest paths.

The compact implementations do not have a good comparison candidate. Comparing our compact versions with the estimates of [5], both are more compact than the estimate, but also slower. The slowdown is mainly caused by the optimized S-box which increases the logic depth.

6 Conclusion and Further Work

The present paper focuses on FPGA implementations of the SHA-3 candidate Grøstl. Several optimized variations were implemented and evaluated. Most optimizations were specifically designed to reduce the number of occupied FPGA slices. Overall the Grøstl hash function fits on small sized FPGAs like the Spartan-3 XC3S200. Further reduction of the area is possible, however, it seems unlikely that Grøstl may be implemented on any much smaller Spartan-3 or Spartan-2 FPGA.

The area reduction is possible by some optimizations we did not pursue, e.g. an 8 bit wide datapath. Other optimizations, like the pipelining of the reduced datapaths, may improve the performance of these compact implementations. Compared to a compact AES implementation (e.g. 222 slices on a Spartan-2 [17]), Grøstl will probably remain rather area consuming. After all none of the proposed implementations fits on the smallest Spartan-2 or Spartan-3 FPGAs. Hence, because our main target are automotive applications, where low budget and therefore the area requirements are important, our further work will continue to focus on additional area optimizations, rather than improvements of the throughput.

References

1. Kayser, R.F.: Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. In: Federal Register. Volume 72. National Institute of Standards and Technology (November 2007) 62212–62220
2. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: In Proceedings of Crypto. Volume 3621 of Lecture notes in computer science., Springer (2005) 17–36
3. Sanadhya, S., Sarkar, P.: New collision attacks against up to 24-step SHA-2. In: Progress in Cryptology-INDOCRYPT. Volume 5365 of Lecture notes in computer science., Springer (2008)
4. Isobe, T., Shibutani, K.: Preimage attacks on reduced tiger and SHA-2. In: Fast Software Encryption. Volume 5665 of Lecture notes in computer science., Springer (2009)
5. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST (2008)
6. Daemen, J., Rijmen, V.: AES Proposal: Rijndael. Submission to NIST (1999)
7. Rijmen, V., Daemen, J.: The Design of Rijndael. Springer (2002)
8. Rijmen, V.: Efficient implementation of the Rijndael S-Box. Technical report, Katholieke Universiteit Leuven (2000)
9. Canright, D.: A Very Compact S-Box for AES. In: Proceedings of 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag (2005) 441–455
10. Zhang, X., Parhi, K.K.: On the Optimum Constructions of Composite Field for the AES Algorithm. In: IEEE Transactions on Circuits and Systems. Volume 53. (2006) 1153–1157
11. Mentens, N., Batina, L., Preneel, B., Verbauwhede, I.: A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box. In: Topics in Cryptology - CT-RSA 2005. Volume 3376 of Lecture Notes in Computer Science., Springer-Verlag (2005) 232–333
12. Nikova, S., Rijmen, V., Schläffer, M.: Using Normal Bases for Compact Hardware Implementations of the AES S-Box. In: Security and Cryptography for Networks. Volume 5229 of Lecture Notes in Computer Science., Springer-Verlag (2008) 236–245
13. 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)
14. 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)
15. 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. Cryptology ePrint Archive, Report 2009/342 (2009)
16. Canright, D., Osvik, D.A.: A More Compact AES. Selected Areas in Cryptography: 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers (2009) 157–169
17. Chodowiec, P., Gaj, K.: Very compact FPGA implementation of the AES algorithm. In: Proceedings of 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag (2003) 319–333

18. Pramstaller, N., Mangard, S., Dominikus, S., Wolkerstorfer, J.: Efficient AES Implementations on ASICs and FPGAs. In: Advanced Encryption Standard – AES. Springer-Verlag (2005) 98–112
19. McLoone, M., McCanny, J.: 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) 65–76
20. Regenscheid, A., Perlner, R., jen Chang, S., Kelsey, J., Nandi, M., Paul, S.: Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition. Technical report, National Institute of Standards and Technology (2009)
21. Xilinx: Spartan-3 Generation FPGA User Guide. (2009)
22. Alfke, P.: Creative Uses of Block RAM. Xilinx. (2008)
23. Lidl, R., Niederreiter, H.: Finite Fields (Encyclopedia of Mathematics and its Applications). Cambridge University Press (1996)
24. Boyar, J., Peralta, R.: New logic minimization techniques with applications to cryptology. Cryptology ePrint Archive, Report 2009/191 (2009)
25. Cong, J., Ding, Y.: On Area/Depth Trade-Off in LUT-Based FPGA Technology Mapping. In: IEEE Transactions on VLSI Syystems. Volume 2. (1994) 137–148