Shrinking KECCAK Hardware Implementations

Bernhard Jungk[1], Marc Stöttinger[2,3] and Matthias Harter[4]

[1] easycore GmbH, Erlangen, Germany
[2] PACE Temasek Laboratories
[3] Division of Mathematical Sciences, SPMS, Nanyang Technological University, Singapore
[4] Hochschule RheinMain, University of Applied Sciences, Wiesbaden Rüsselsheim, Germany

Abstract. The SHA-3 competition ended in late 2012 by announcing KECCAK as the winning algorithm. During the contest, several criteria were evaluated for hardware implementations, foremost the resource consumption, the throughput and the tradeoff between both criteria. Unfortunately, until recently, there were very few performance studies for lightweight and midrange implementations. This paper extends the work on efficient KECCAK implementations to lightweight applications with KECCAK versions using a reduced state size. Furthermore, we present extended and improved versions of previous design space explorations and implementation results. For instance, the throughput-area ratio has been improved by 10-20 % compared to earlier investigations.

1 Introduction

In 2012 the National Institute of Technology (NIST) selected the KECCAK algorithm as the hash function to be standardized as SHA-3, after a five year long competition cf. [1–3]. During and after the contest, various implementations of KECCAK for hardware platforms have been proposed and evaluated. However, very few results for non-standard implementations were published.

In this contribution, we add a study of more variants of KECCAK to improve the understanding of the area and throughput impact of different parameters for the KECCAK hash function. In particular, the impact of the state size and the capacity and hence also of the rate is investigated. First, we describe a improved version of the design strategy proposed in [4]. Second, we extend the evaluation of KECCAK implementations to smaller state sizes. This additional analysis consists of a short review of the theoretical security of the chosen parameter sets and a performance evaluation of each set. Third, we generated a lot more FPGA results and added a few selected ASIC results for the same architectures.

The remainder of this paper is organised as follows. First, we provide an overview of previous work in Section. 2. After that, we give a rationale for the parameters for smaller state sizes of Keccak in Section 3. Then, we describe our implementations in Section 4, and compare them to previous KECCAK FPGA (Section 5) and ASIC designs (Section 6) , before we conclude our paper with a summary and future work in Section 7.

2 Previous Work

Many hardware designs for FPGAs have been published during the SHA-3 competition. To the best of our knowledge, only three generally different architecture types of the KECCAK algorithm have been implemented and published in the literature so far, while the KECCAK designers proposed more possibilities to process the state in hardware, cf. [5].

Most designs compute the compression function using a fully parallel data path to reduce the number of clock cycles to a minimum. These implementations focus on maximising the throughput, reaching up to 13 GBit/s (cf. [6–8]). Variants of this architecture type are pipelined or process more than one data stream in parallel.

For lightweight implementations, a lane-oriented architecture is favoured in most of the proposed designs, cf. [9–11]. The only alternatively implemented design strategy for area-efficient implementations of KECCAK is a slice-oriented architecture as proposed in [12], cf. Fig. 1. This implementation strategy was also followed by the recent and more extensive study in [4]. Table 2 in Sect. 4 provides an overview of these implementations in terms of resource consumption and throughput.

For ASICs, there are also a lot of reported results. The most relevant results reported in the literature are [13] and [14]. Kavun et al. investigated lightweight implementations of KECCAK with the full and reduced states [13]. In [14], the area of the full KECCAK-$f[1600]$ was pushed further to consume even less area.

3 Keccak Parameters

KECCAK is a sponge function. This means, it can be parameterized by the state size b, the rate r and the capacity c. The three parameters are interdependend, i.e. $b = c + r$ and thus, changing one parameter changes at least one other parameter. A forth important parameter is the size n of the message digest. The parameters b and r determine the performance of KECCAK, whereas c and n are important security parameters.

The KECCAK variant which was proposed as a SHA-3 candidate is the variant with $b = 1600$ bits of internal state. As we will see in the evaluation in Section 5, it is already possible to implement this KECCAK-$f[b = 1600]$ variant with a reasonable amount of area on modern FPGAs. However, several applications emphasise lower costs and thus area over throughput and security. Therefore, the security requirements may be lowered for these applications to achieve less implementation cost. Therefore, we will also analyse variants of KECCAK-$f[b]$ with $b \in \{200, 400, 800\}$. The investigated variants still have reasonable security for many applications.

The evaluation of these variants is split in two parts. First, we discuss the sponge parameters, which we chose to evaluate and which determine the security of the resulting hash function. Second, the performance evaluation will be given in Section 4.

3.1 Security Parameters

There are three major security claims presented by Bertoni et al. for the sponge construction and KECCAK, cf. [15, 16], which can be used to derive meaningful parameter sets. The most interesting security properties of hash functions are the collision, the preimage and the second preimage resistance. The exact claimed security bounds in terms of hash digest length n and capacity c are:

- Collision resistance: $O(min(2^{n/2}, 2^{c/2}))$
- Preimage resistance: $O(min(2^{n}, 2^{c/2}))$
- Second preimage resistance: $O(min(2^{n}, 2^{c/2}))$

Note, that the security assumed for the preimage resistance is claimed to be higher by some publications, because no generic attack is known which is as good as the theoretical bound, cf. [17].

The security parameters used in the present evaluation are partially derived from the Photon specification adapted to KECCAK, cf. [17], with the exception of the smallest Photon variant. Additionally, higher

Table 1: KECCAK parameters.

Message digest	State Size	Capacity	Rate	Rounds
128	200	128	72	18
160	200	160	40	18
128	400	128	272	20
128	400	256	144	20
160	400	160	240	20
160	400	320	80	20
224	400	224	176	20
256	400	256	144	20
256	800	256	544	22
256	800	512	288	22
256	1600	512	1088	24

security versions are evaluated for the message digest sizes, where the state size allows a theoretically optimal preimage resistance. All evaluated parameters are presented in Table 1. Using the same Photon parameter sets has the main advantage, that it is easy to replace one hash function with the other and it will be easier to compare the performance of both algorithms. Similar, but slightly different parameters are used by the Spongent hash function [18]. For the ASIC designs, only a subset of these options is investigated for now, because of the long time needed to evaluate all possibilities.

4 Generic Implementation

We implemented a generic slice-based architecture, based on the same reasoning as [4]. The general slice-based architecture is identical to the previously published architecture. For FPGAs, an additional optimization using manual instantiation of `LUT6_2` primitives further reduced the area consumption. Additionally, the generic VHDL code was synthesized for a few selected parameter sets for ASICs.

4.1 Basic Architecture

The generic architecture can be parametrised to change the data path width, the state size and the size of the message digest to evaluate all parameter sets depicted in Table 1 for all possible variants of the basic architecture, beginning with the computation of only one slice per clock cycle up to 2^l slices, i.e. the data path width $d \in \{25 \cdot 2^k | 1 \leq k \leq l\}$ can be selected to trade throughput for area savings.

The proposed architecture stays the same for all of the evaluated versions, as depicted in Fig. 2. It uses the rescheduled round function proposed in [12] and the hardware interface from [7,19], which we tailored to be parametrisable down to $w_{\text{in}} = w_{\text{out}} = 16$ bit width for the larger state sizes and $w_{\text{in}} = w_{\text{out}} = 8$ bit for the smallest state size ($b = 200$). For a possible version with $b = 100$, the interface is more difficult to adapt, because for e.g. $n = c = 80$, the rate $r = 20$ is not dividable by 8 and thus, some changes to the implementation of the interface is needed. The basic data flow in the architecture is as follows:

1. The input message block is absorbed into the state RAM using the input FIFO.
2. After the absorption of a message block, the round function is used to compute the KECCAK permutation on the state RAM. Repeat step (1) and (2) for all message blocks.

3. Afterwards, the output of the state is transferred to the user of the hash function using the output FIFO in the squeezing phase to generate the message digest.

4.2 Absorption

According to the general data flow, the absorption phase is separated from the processing of the round function. For a compact implementation, this is necessary, because otherwise, a complete message block would have to be stored before the actual absorption. The reason for this property is, that the input is usually supplied in a lane-oriented way, which does not work well with the slice-oriented designs and the input has to be either reordered before the absorption or it has to be separated from the processing. A third possibility is to reorder the message block before the transfer to the implementation, however, this puts additional requirements on the user of the hardware implementation.

In our architecture, we focused on a compact design. Thus, each input data of 8, 16 or 32 bits is split into $^{d}\!/_{25}$ bit chunks. Each chunk has to absorbed for the correct lane and thus, appropriate read and write addresses have to be calculated. For example if $d = 25$ and the interface is 16 bit wide, then 16 clock cycles are necessary for the absorption of one data transfer, because every bit is absorbed in a separate clock cycle. This procedure removes the need for additional memory as it was used in [12]. However, it adds $^{25 \cdot r}\!/_{d}$ additional clock cycles.

4.3 Round Function

The input data for the functional modules implementing ι, χ (cf. Fig. 3b), and θ (cf. Fig. 3a) is read from the state memory module and also written back to it after only one clock cycle. The round constants for the ι function are loaded from a ROM with the (sub) round counter as read address. The χ permutation can be easily computed for each slice in one clock cycle. The θ function is split into two parts. The first computes for all parallel computed slices the sums necessary to compute the final output of θ. The second part computes the final outputs. For the slice with the highest z in this clock cycle, the sums are temporarily stored for the computation in the next clock cycle.

However, the slice with $z = 0$ depends on the data from the slice with $z = 2^l - 1$. This data dependency is solved by a special case which stores the intermediate values after χ for the slice with

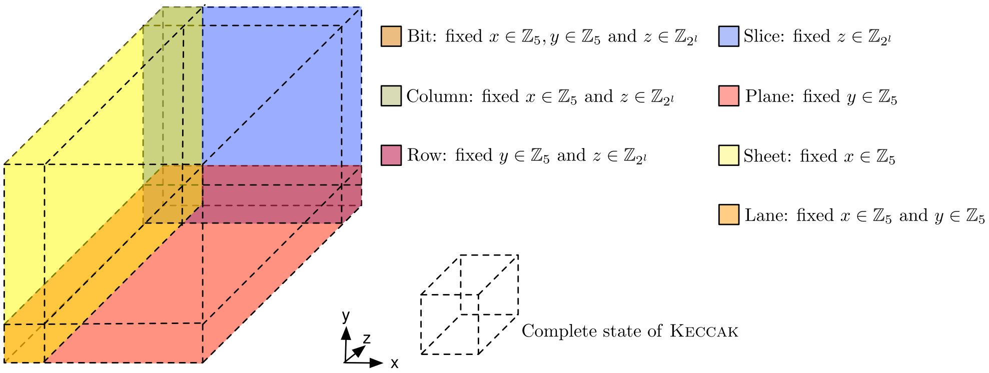

Bit: fixed $x \in \mathbb{Z}_5, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_{2^l}$
Slice: fixed $z \in \mathbb{Z}_{2^l}$
Column: fixed $x \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_{2^l}$
Plane: fixed $y \in \mathbb{Z}_5$
Row: fixed $y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_{2^l}$
Sheet: fixed $x \in \mathbb{Z}_5$
Lane: fixed $x \in \mathbb{Z}_5$ and $y \in \mathbb{Z}_5$
y
z
x
Complete state of KECCAK

Fig. 1: Different storage organizations of the KECCAK state

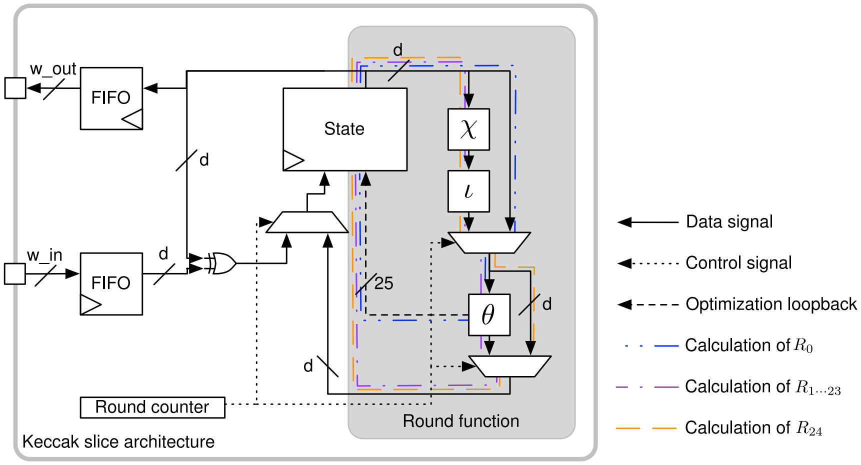
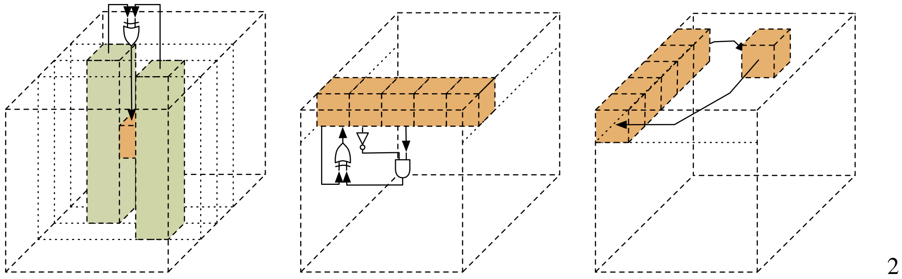


Fig. 2: Generic architecture of the proposed slice-based implementation.

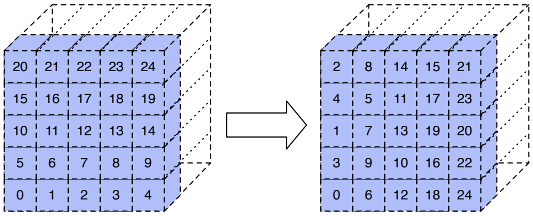
$z = 0$. The final results for this slice are computed in the same clock cycle in which the results for $z = 2^l - 1$ are calculated. Hence, in the last sub-step of θ, $d/25 + 1$ slices are processed, which hides the latency from the first round, in which only $d/25 - 1$ final outputs of θ are calculated.

The other KECCAK sub-functions π (cf. Fig. 3d) and ρ (cf. Fig. 3c) are realised by using appropriate read and write addresses. The π transformation only shuffles the bits between lanes and thus, can be implemented using a fixed reordering of the bits in a slice.

The ρ transformation is implemented by splitting the RAM into a lane-wise organisation, where each lane is again divided into two RAMs with a total width of $d/25$ bits. If there are two such RAMs RAM_0



(a) Dependencies of θ (b) Dependencies of χ (c) Dependencies of ρ



20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4

2	8	14	15	21
4	5	11	17	23
1	7	13	19	20
3	9	10	16	22
0	6	12	18	24

(d) Dependencies of π

Fig. 3: Visualisation the data dependencies of the different KECCAK functions

and RAM_1 for a lane, and the rotation is by q bits, then for the first round the data for clock cycle i is read from $\mathrm{RAM}_0(i)$ and $\mathrm{RAM}_1(i)$ and afterwards written back to $\mathrm{RAM}_1(i)$ and $\mathrm{RAM}_0(i+1)$. Since this procedure rotates the data only by $q \mod {}^{d}/_{25}$ bits, a round counter is used to manipulate the read addresses for consecutive round(s) to facilitate the rest of the rotation, which is a multiple of ${}^{d}/_{25}$ bits and thus, can be realized by a moving read offset per lane.

The multiplexers before and after the θ function are used to realise the rescheduling of the sub-function sequence of the original KECCAK.

4.4 Squeezing

The squeezing alternates between generating output bits and further computations of the KECCAK permutation. Similar to the message absorption, it is necessary to reorder the bits, which is achieved in a almost identical way. The only difference is, that instead of reading and writing to the state memory, the state is only read and the bits necessary for the message digest output are written to the output FIFO, until w_{out} bits are written to the FIFO and the transfer using the outgoing interface is initiated. This process is repeated ${}^{r}/_{w_{\text{out}}}$ times.

4.5 Throughput Calculation

The throughput for large messages $\mathcal{T}_{\text{large}}$ of each design is calculated as follows, where f is the clock frequency, n_r is the number of rounds, b is the state size, r is the rate and d is the data path width:

$$\mathcal{T}_{\text{large}} = \frac{r \cdot f}{\left((n_r+1)\cdot \frac{b}{d}\right) + \left(\frac{r\cdot 25}{d}\right)}. \tag{1}$$

For shorter messages this approximation is invalid because of the squeezing phase. Therefore the throughput $\mathcal{T}_{\text{general}}$ is approximated by the following formula, where n is the size of the message digest and m is the number of message blocks:

$$\mathcal{T}_{\text{general}} = \frac{m \cdot r \cdot f}{m \cdot \left[\left((n_r+1)\cdot \frac{b}{d}\right) + \left(\frac{r\cdot 25}{d}\right)\right] + \left(\lfloor \frac{n}{r} \rfloor (n_r+1)\cdot \frac{b}{d}\right) + n}. \tag{2}$$

Since the first term of the denominator $m \cdot \left[\left((n_r+1)\cdot \frac{b}{d}\right) + \left(\frac{r\cdot 25}{d}\right)\right]$ grows, whereas the other terms are constant, Eq. 1 follows from Eq. 2 for large m.

5 FPGA evaluation

In this section the FPGA evaluation is described. This consists first on a description of the `LUT6_2` optimisation, the process of hardware synthesis and finally the discussion of the post place and route results. The discussion is limited to Virtex-5 FPGAs.

5.1 Optimisation using LUT instances

For Virtex-5 and newer FPGAs, it is possible to optimise the design by manually instantiating LUT primitives. For the present work, we combined the χ and ι functions together with the multiplexer before θ. The direct instantiation helps to reduce the slice count compared to the older results by Jungk et al., cf. [4]. This idea works nicely, because for each output bit of χ only 3 input bits are necessary, ι only works on one or zero bits per row and the multiplexer selects only between computation of χ and ι or route-through. Therefore, we can package the computation of four output bits into two `LUT6_2` instances with four data bit inputs. The multiplexer bit and the bit with $x = 0$ is assigned to a single `LUT6_2` instance, which additionally computes ι. Overall, we need only three `LUT6_2` instances per row.

5.2 Synthesis Settings

In order to evaluate our implementations, we used a systematic approach similar to the ATHENa framework, cf. [20]. We synthesised our design for all $d \in \{25 \cdot 2^k | 1 \leq k \leq l\}$. For each setting, we used Xilinx ISE 14.5 to synthesise the designs with various optimisation settings. The settings with the best post-place and route results were then used in a second optimisation step to increase the throughput by tightening the timing constraints. In addition to our previous results in [4], we detailed the analysis of KECCAK-f[1600] and KECCAK-f[800] for less constrained platforms. For the analysis of lightweight variants, we extended the analysis of KECCAK-f[400] and furthermore add an investigation of KECCAK-f[200].

The best post-place and route results of the proposed slice architecture are shown in Fig. 4 and Fig. 5, visualising the design trade-offs for different parameter sets. The exploration space for all investigated KECCAK constructions, KECCAK-f[1600], KECCAK-f[800], KECCAK-f[400] and KECCAK-f[200] are plotted on a logarithmic axis, showing the area consumption in slices over the performance denoted in MBit/s. Each proposed design, as well as the previously introduced FPGA-based designs from the literature, are individually marked. We also highlight certain bounds of throughput-area ratio, which is a commonly used metric to compare different designs in terms of their efficiency. We additionally provide the resource consumption and throughput of the designs in plain numbers in Tab. 2 and Tab. 3, respectively.

5.3 Higher Security Implementations

The implementations of the KECCAK-f[1600] and KECCAK-f[800] designs, are addressed in this section. First, the implementations of our proposed KECCAK-f[1600] design as well as other design from the

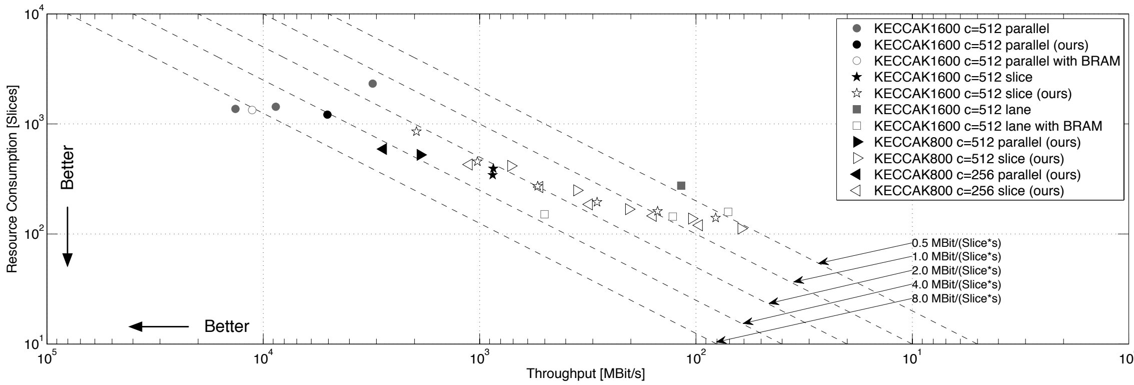


Fig. 4: Throughput-Area tradeoff for various implementations of KECCAK-f[1600] and KECCAK-f[800].

Table 2: Implementation results for Keccak-$f[1600]$ and Keccak-$f[800]$

Design							Resource		Performance		Metric
Variant	State structure	Data path (d) [Bit]	Capacity (c) [Bit]	Rate (r) [Bit]	Architecture	Platform	Slices	BRAM/ DSP	Frequency (f) [Mhz]	Throughput ($\mathcal{T}$) [Mbit/s]	[Mbit] / [s*slice]
Keccak-$f[1600]$	Slice	25	512	1088	This paper	Virtex-5	**140**	0/0	200	81	0.58
		50	512	1088	This paper	Virtex-5	161	0/0	186	151	0.93
		100	512	1088	This paper	Virtex-5	195	0/0	177	287	1.47
		200	512	1088	This paper	Virtex-5	272	0/0	166	539	1.98
		200	512	1088	[12][1]	Virtex-5	393	0/0	159	864	2.20
		200	512	1088	[21]	Virtex-5	344	0/0	-	870	2.53
		400	512	1088	This paper	Virtex-5	455	0/0	158	1024	2.25
		800	512	1088	This paper	Virtex-5	854	0/0	151	1959	2.29
	Lane	64	512	1088	[9]	Virtex-6	144	0/0	250	128	0.89
		64	512	1088	[11]	Virtex-5	151	3/0	520	501	3.32
		64	512	1088	[10]	Virtex-5	159	1/0	248	71	0.45
		64	512	1088	[10]	Virtex-5	275	0/0	260	117	0.43
	Parallel	1600	512	1088	This paper	Virtex-5	1,215	0/0	195	5,054	4.16
		1600	512	1088	[7]	Virtex-5	1,338	1/0	248	11,252	8.41
		1600	512	1088	[7]	Virtex-5	1,369	0/0	297	13,452	**9.83**
		1600	512	1088	[8]	Virtex-5	1,433	0/0	205	8,747	6.10
		1600	512	1088	[6][1]	Virtex-5	2,326	0/201	306	3,120	1.34
Keccak-$f[800]$	Slice	25	256	544	This paper	Virtex-5	120	0/0	227	96	0.80
		25	512	288	This paper	Virtex-5	**112**	0/0	220	62	0.55
		50	256	544	This paper	Virtex-5	146	0/0	186	158	1.08
		50	512	288	This paper	Virtex-5	138	0/0	168	105	0.76
		100	256	544	This paper	Virtex-5	186	0/0	163	312	1.68
		100	512	288	This paper	Virtex-5	168	0/0	162	205	1.22
		200	256	544	This paper	Virtex-5	267	0/0	155	528	1.98
		200	512	288	This paper	Virtex-5	249	0/0	158	355	1.43
		400	256	544	This paper	Virtex-5	428	0/0	165	1,120	2.62
		400	512	288	This paper	Virtex-5	416	0/0	159	717	1.72
		800	256	544	This paper	Virtex-5	591	0/0	205	2,785	**4.71**
	Parallel	800	512	288	This paper	Virtex-5	524	0/0	209	1,880	3.59

[1] With padder unit for performing the padding in the design.

literature are discussed, then we will investigate the implementation results of the Keccak-$f[800]$ design with different capacities.

One can see that the designs with the throughput of the proposed slice-oriented architecture of Keccak-$f[1600]$ scales almost linear by increasing d, cf. Tab. 2. However, the throughput-area ratio ($^{\text{MBit}}/_{\text{Slices}\cdot\text{s}}$) does not scale proportionally to d, cf. Fig. 4. This effect is probably caused by a static resource consumption offset, which is due to the 1600 bit large state memory of Keccak and the control logic, which both have a resource consumption, that is less dependent on the data path width d. Hence, the designs with $d = \{200, 400, 800\}$ have a roughly similar throughput-area efficiency, because the static resource consumption is a less dominant part of these designs. However, for the smaller designs, the efficiency drops considerably.

The presented slice-oriented architecture outperforms the previously introduced designs of [12, 21] in terms of resource utilisation at the same data path width of $d = 200$. Our optimised variant with $d = 25$

is the smallest design reported so far, even if compared to the designs using BRAM, e.g. [10, 11]. For example, the result reported in [11] is faster than our smallest design, but uses 3 BRAMs, which adds a considerable overhead, that is not covered by the slice count. On the left side of Fig. 4 one can find published parallel Keccak implementation results, cf. [6–8]. Our parallel design, based on the proposed architecture is one of the slowest parallel design, mostly because of the message absorption overhead. However, it is also the smallest of these designs.

For the Keccak-$f[800]$ implementations, we extended the analysis and also include designs with different capacities. The capacity does not only influence the security of the Keccak algorithm, it also influences the resource consumption and hence, the efficiency of the architecture, cf. Tab. 2. The more the secure the design gets, the smaller is the implementation on the one hand, but on the other hand the throughput also gets less. Both effects are caused by the smaller rate. If the capacity increases, the rate decreases, thus, the performance decreases. On the other hand, the control logic shrinks slightly, if the rate decreases, because of smaller counters for the message absorption and squeezing phases. Hence, the resource consumption is less for the variant with the higher capacity.

5.4 Lightweight Implementations

Lightweight hash functions are used in RFID components and embedded devices for authentication purposes. The large state of Keccak-$f[1600]$ is a very high hurdle for its usage in these applications. Therefore, we also investigated the properties of our proposed architecture for the Keccak-$f[400]$ and Keccak-$f[200]$ variants, which are lightweight versions of the SHA-3 winner. The results of the Keccak-$f[400]$ and Keccak-$f[200]$ designs are similar to the higher security implementations in terms of its scalability, but are in general considerably smaller than the heavyweight siblings, see Table 3 and Fig. 5.

The proposed slice-oriented architecture seems to be also quite scalable for lightweight applications of the Keccak algorithm. The overall throughput is determined be the data path width d and the rate r. The parameter d determines the number of clock cycles and has also an influence on the clock frequency.

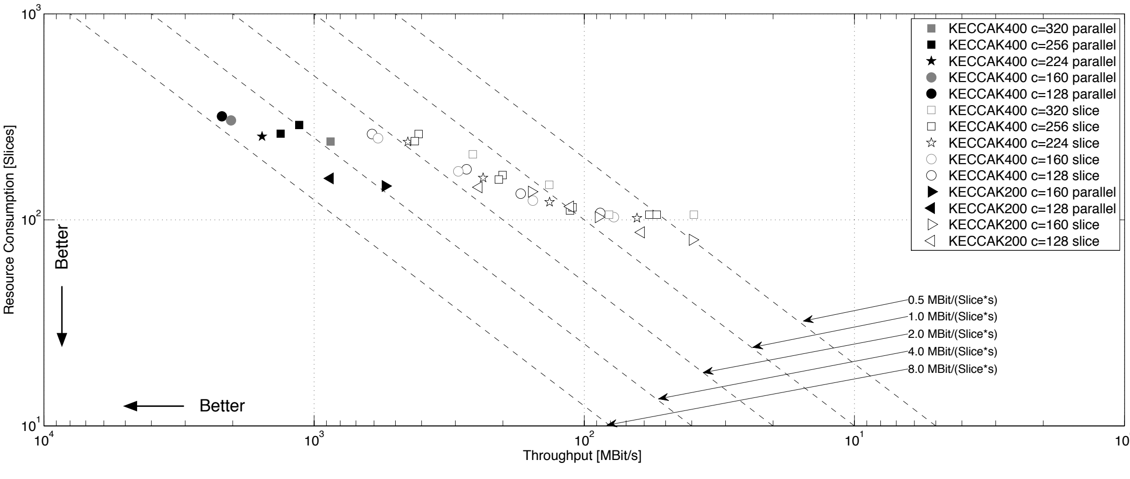


Fig. 5: Throughput-Area tradeoff for various implementations of Keccak-$f[400]$ and Keccak-$f[200]$.

Table 3: Implementation results for KECCAK-$f[400]$ and KECCAK-$f[200]$ on Virtex-5

Design						Resource		Performance		Metric
Variant	State structure	Data path (d) [Bit]	Message Digest (n) [Bit]	Capacity (c) [Bit]	Rate (r) [Bit]	Slices	BRAM/ DSP	Frequency (f) [Mhz]	Throughput[1] ($\mathcal{T}$) [Mbit/s]	[Mbit] [s*slice]
KECCAK-$f[400]$	Slice	25	128	256	144	106	0/0	191	57	0.54
		25	128	128	272	108	0/0	195	87	0.81
		25	160	160	240	103	0/0	186	78	0.75
		25	160	320	80	106	0/0	204	39	0.37
		25	224	224	176	**102**	0/0	186	64	0.63
		25	256	256	144	106	0/0	180	54	0.51
		50	128	256	144	111	0/0	188	113	1.02
		50	128	128	272	134	0/0	192	172	1.28
		50	160	160	240	124	0/0	186	155	1.25
		50	160	320	80	106	0/0	211	81	0.76
		50	224	224	176	122	0/0	195	134	1.10
		50	256	256	144	115	0/0	184	110	0.96
		100	128	256	144	165	0/0	167	201	1.22
		100	128	128	272	176	0/0	152	273	1.55
		100	160	160	240	172	0/0	176	293	1.70
		100	160	320	80	148	0/0	175	135	0.91
		100	224	224	176	160	0/0	172	237	1.48
		100	256	256	144	157	0/0	173	207	1.32
		200	128	256	144	261	0/0	171	410	1.57
		200	128	128	272	261	0/0	171	610	2.34
		200	160	160	240	249	0/0	174	580	2.33
		200	160	320	80	208	0/0	168	259	1.24
		200	224	224	176	239	0/0	163	450	1.88
		200	256	256	144	241	0/0	177	424	1.76
	Parallel	400	128	256	144	289	0/0	236	1135	3.93
		400	128	128	272	318	0/0	306	2194	**6.90**
		400	160	160	240	304	0/0	304	2029	6.68
		400	160	320	80	240	0/0	283	869	3.62
		400	224	224	176	254	0/0	283	1558	6.13
		400	256	256	144	262	0/0	277	1330	5.07
KECCAK-$f[200]$	Slice	25	128	128	72	87	0/0	191	62	0.71
		25	160	160	40	**80**	0/0	191	40	0.50
		50	128	128	72	116	0/0	175	113	0.97
		50	160	160	40	103	0/0	213	89	0.86
		100	128	128	72	144	0/0	191	246	1.71
		100	160	160	40	137	0/0	188	157	1.14
	Parallel	200	128	128	72	159	0/0	339	872	**5.48**
		200	160	160	40	146	0/0	327	545	3.73

However, variations of the clock frequency for versions with the same value for d seem to be of no statistical relevance, because the variations due to different settings for the Xilinx tool chain is usually higher. The rate r determines the number of input bits processed per clock cycle. Please note that for the evaluation, we focused on hashing long messages, thus for shorter messages additional squeezing steps are required for the designs with the property $r < n$, cf. Eq. 2.

6 ASIC evaluation

In addition to the extensive evaluation of FPGA results, we chose to generate ASIC results for three implementations (cf. Tab. 4). Since we were mainly interested in low area results for ASICs, we chose the following three variants:

- Keccak-[200] with $n = 128, c = 128, r = 72$, and $d = 25$.
- Keccak-[400] with $n = 160, c = 160, r = 240$, and $d = 25$.
- Keccak-[400] with $n = 256, c = 256, r = 144$, and $d = 25$.

The optimization phase for these implementations was first conducted to optimize for throughput. Therefore, the gate count is not comparable to earlier implementations, because of the large clock tree. However, it is still interesting to note that the area reduction from $b = 400$ to $b = 200$ is considerable (around 38-41%). A second optimization round for area optimization was also performed, which resulted in the second set of results for each technology in Tab. 4.

The results show, that the implementation of the architecture is less efficient for ASICs, despite being very suitable for FPGAs. However, comparing our FPGA results to earlier ASIC performance reports for Keccak it can be assumed, that it is possible to build an implementation that is a lot smaller for both choices of b, because the smallest reported ASIC implementation for the full Keccak requires only about 5.54 kGE. Thus, it seems likely, that such area reduced versions of Keccak with a smaller state size have the potential to rival hash functions like Photon, which were specially designed for lightweight applications. For reference, we included several Photon results in Tab. 4.

Table 4: ASIC implementation results for Keccak-$f[400]$ and Keccak-$f[200]$.

Variant	Capacity (c) [Bit]	Rate (r) [Bit]	Architecture	Process	Area kGE	Frequency [Mhz]	Throughput [MBit/s]	Throughput @ 100 kHz [MBit/s]
Photon-256/32/32	256	32	[17]	UMC 65nm	2.17	N/A	N/A	0.003
Photon-160/36/36	160	36	[17]	UMC 65nm	1.39	N/A	N/A	0.0027
Photon-128/16/16	128	16	[17]	UMC 65nm	1.12	N/A	N/A	0.0016
Keccak-$f[1600]$	512	1088	[14]	UMC 130nm	5.52	N/A	N/A	0.0044
Keccak-$f[400]$	256	144	This paper	UMC 65nm	13.84	860	258	0.03
			This paper	UMC 65nm	8.69	477	143	0.03
			This paper	AMS 350nm	7.51	178	53.4	0.03
			This paper	AMS 350nm	6.64	104	31.2	0.03
			[13]	130nm	5.09	N/A	N/A	0.0144
	160	240	This paper	UMC 65nm	13.93	842	350	0.04
			This paper	UMC 65nm	8.80	457	190	0.04
			This paper	AMS 350nm	7.60	176	73.3	0.04
			This paper	AMS 350nm	6.68	109	45.4	0.04
Keccak-$f[200]$	128	72	This paper	UMC 65nm	8.63	908	291	0.032
			This paper	UMC 65nm	5.09	482	154	0.032
			This paper	AMS 350nm	4.50	178	57.2	0.032
			This paper	AMS 350nm	3.87	113	36.3	0.032
			[13]	130nm	2.52	N/A	N/A	0.008

7 Conclusion

In this contribution, we extended our previous analysis of the SHA-3 contest winner KECCAK in [4]. We re-used the previously developed KECCAK architecture to extend the study of implementations for other KECCAK-$f[b]$ versions, with $b \in \{800, 400, 200\}$, and consequently also to different security parameters and interesting usage scenarios.

In this extended design space exploration, it turns out that the slice-organised storage representation also seems to be a very efficient one for lightweight versions of KECCAK, i.e. KECCAK-$f[200]$, as well as midrange versions and implementations. This slice-oriented architecture of the proposed implementation scales almost linearly in terms of the design's throughput by varying the data path width, while keeping all other parameters the same. Thus, scaling KECCAK is almost straightforward, depending on the security and performance requirements. However, the reported ASIC results are worse, which shows, that the architecture or at least the implementation previously optimized for FPGAs is probably less suitable for ASICs.

Overall, for FPGAs, the proposed architecture is able to close the gap between the previously published high-throughput designs and the lightweight implementations by adjusting the parameter d of the generic data path, the capacity c and the state size b. Furthermore, we presented the so far smallest implementation of KECCAK-$f[1600]$, KECCAK-$f[800]$, KECCAK-$f[400]$, and KECCAK-$f[200]$ in terms of slices using a Virtex-5 FPGA without using BRAM primitives.

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. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak SHA-3 Submission. Submission to NIST (Round 3) (2011)
3. Chang, S., Perlner, R., Burr, W.E., Turan, M.S., Kelsey, J.M., Paul, S., Bassham, L.E.: Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition (2012)
4. Jungk, B., Stöttinger, M.: Among Slow Dwarfs and Fast Giants: A Systematic Design Space Exploration of KECCAK. In: 8th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), IEEE (2013) 1–8
5. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: 1001 Ways to Implement Keccak. Third SHA-3 candidate conference (2012)
6. Provelengios, G., Kitsos, P., Sklavos, N., Koulamas, C.: FPGA-based Design Approaches of Keccak Hash Function. In: 15th Euromicro Conference on Digital System Design, IEEE (2012) 648–653
7. Shahid, R., Sharif, M.U., Rogawski, M., Gaj, K.: Use of Embedded FPGA Resources in Implementations of 14 Round 2 SHA-3 Candidates. In Tessier, R., ed.: International Conference on Field-Programmable Technology, IEEE (2011) 1–9
8. Knezevic, M., Kobayashi, K., Ikegami, J., Matsuo, S., Satoh, A., Koçabas, Ü., Fan, J., Katashita, T., Sugawara, T., Sakiyama, K., Verbauwhede, I., Ohta, K., Homma, N., Aoki, T.: Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA-3 Candidates. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **20**(5) (2012) 827–840
9. Kerckhof, S., Durvaux, F., Veyrat-Charvillon, N., Regazzoni, F., de Dormale, G.M., Standaert, F.X.: Compact FPGA Implementations of the Five SHA-3 Finalists. In: 10th Smart Card Research and Advanced Application Conference. Volume 7079 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2011) 217–233
10. Kaps, J.P., Yalla, P., Surapathi, K.K., Habib, B., Vadlamudi, S., Gurung, S., Pham, J.: Lightweight Implementations of SHA-3 Finalists on FPGAs. Submission to NIST (Round 3) (2011)
11. San, I., At, N.: Compact Keccak Hardware Architecture for Data Integrity and Authentication on FPGAs. Information Security Journal: A Global Perspective **21**(5) (2012) 231–242
12. Jungk, B.: Evaluation of Compact FPGA Implementations for All SHA-3 Finalists. Third SHA-3 Candidate Conference (2012)
13. Kavun, E.B., Yalcin, T.: A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications. In: Radio frequency identification: security and privacy issues. Springer (2010) 258–269

14. Pessl, P., Hutter, M.: Pushing the Limits of SHA-3 Hardware Implementations to Fit on RFID. In: Cryptographic Hardware and Embedded Systems - CHES '13. Springer (2013) 126–141
15. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak Reference. Online publication (2011)
16. Bertoni, G., Daemen, J., Peeters, M., van Assche, G.: Cryptographic sponge functions (2011)
17. Guo, J., Peyrin, T., Poschmann, A.: The PHOTON Family of Lightweight Hash Functions. In: Advances in Cryptology CRYPTO 2011. Volume 6841 of Lecture Notes in Computer Science. Springer-Verlag (2011) 222–239
18. Bogdanov, A., Knezevic, M., Leander, G., Toz, D., Varici, K., Verbauwhede, I.: SPONGENT: The Design Space of Lightweight Cryptographic Hashing. IEEE Transactions on Computers **62**(10) (2013) 2041–2053
19. Gaj, K., Homsirikamol, E., Rogawski, M.: Fair and comprehensive methodology for comparing hardware performance of fourteen round two sha-3 candidates using fpgas. In Mangard, S., Standaert, F.X., eds.: CHES. Volume 6225 of Lecture Notes in Computer Science., Springer (2010) 264–278
20. Gaj, K., Kaps, J.P., Amirineni, V., Rogawski, M., Homsirikamol, E., Brewster, B.Y.: ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs. In: 20th International Conference on Field Programmable Logic and Applications, IEEE (2010) 414–421
21. Gaj, K., Homsirikamol, E., Rogawski, M., Shahid, R., Sharif, M.U.: Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs. Cryptology ePrint Archive, Report 2012/368 (2012)