

Implementation aspects of Keccak

Guido BERTONI¹, Joan DAEMEN¹,
Michaël PEETERS², Gilles VAN ASSCHE¹

Parts jointly with
Nicolas DEBANDE³, Thanh-Ha LE³, Ronny VAN KEER¹

¹STMicroelectronics ²NXP Semiconductors

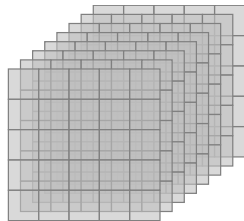
³Morpho

KECCAK & SHA-3 Day
Université Libre de Bruxelles
March 27, 2013

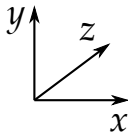
Outline

- 1 Implementing KECCAK (*how to cut a state*)
- 2 Power-attacking KECCAK
- 3 Power-protecting KECCAK

The state: an array of $5 \times 5 \times 2^\ell$ bits

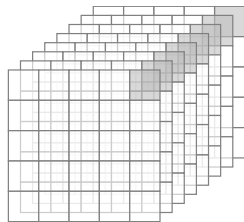


state

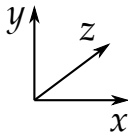


- 5×5 lanes, each containing 2^ℓ bits (1, 2, 4, 8, 16, 32 or 64)
- (5×5) -bit slices, 2^ℓ of them

The state: an array of $5 \times 5 \times 2^\ell$ bits

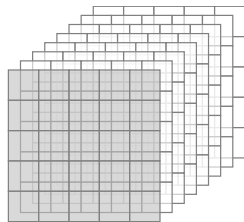


lane

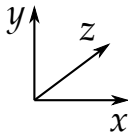


- 5×5 lanes, each containing 2^ℓ bits (1, 2, 4, 8, 16, 32 or 64)
- (5×5) -bit slices, 2^ℓ of them

The state: an array of $5 \times 5 \times 2^\ell$ bits

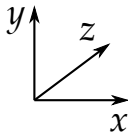
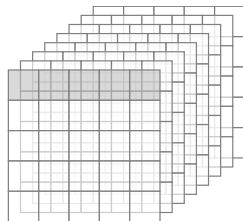


slice



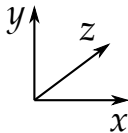
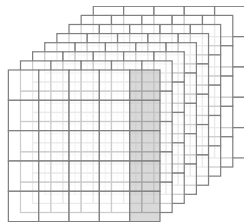
- 5×5 lanes, each containing 2^ℓ bits (1, 2, 4, 8, 16, 32 or 64)
- (5×5) -bit slices, 2^ℓ of them

The state: an array of $5 \times 5 \times 2^\ell$ bits



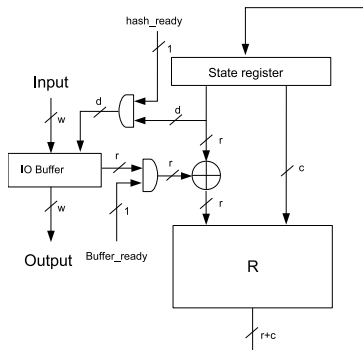
- 5×5 lanes, each containing 2^ℓ bits (1, 2, 4, 8, 16, 32 or 64)
- (5×5) -bit slices, 2^ℓ of them

The state: an array of $5 \times 5 \times 2^\ell$ bits



- 5×5 lanes, each containing 2^ℓ bits (1, 2, 4, 8, 16, 32 or 64)
- (5×5) -bit slices, 2^ℓ of them

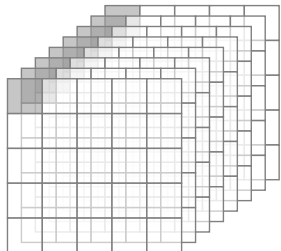
Not cutting it: straightforward hardware architecture



- Logic for one round + register for the state
 - very short critical path \Rightarrow high throughput
- Multiple rounds can be computed in a single clock cycle
 - 2, 3, 4 or 6 rounds in one shot

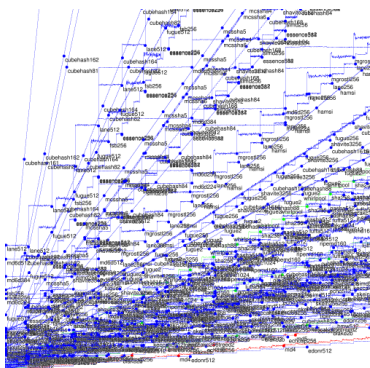
Lanes: straightforward software implementation

- Lanes fit in 2^ℓ -bit registers
 - 64-bit lanes for KECCAK-f[1600]
 - 8-bit lanes for KECCAK-f[200]
- Very basic operations required:
 - θ XOR and 1-bit rotations
 - ρ rotations
 - π just reading the correct words
 - χ XOR, AND, NOT
 - ι just a XOR



Lanes: straightforward software implementation

- Faster than SHA-2 on all modern PC
- KECCAKTREE faster than MD5 on some platforms

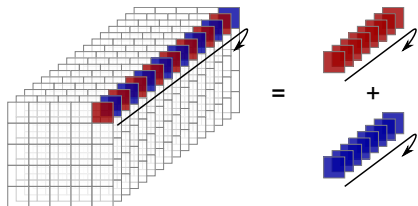


C/b	Algo	Strength
4.79	keccakc256treed2	128
4.98	md5 broken!	64
5.89	keccakc512treed2	256
6.09	sha1 broken!	80
8.25	keccakc256	128
10.02	keccakc512	256
13.73	sha512	256
21.66	sha256	128

[eBASH, hydra6, <http://bench.cr.yp.to/>]

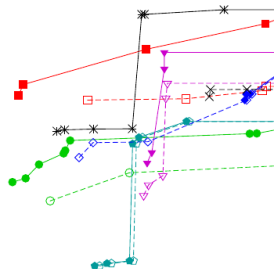
Bit interleaving

- Ex.: map 64-bit lane to 32-bit words
 - ρ seems the critical step
 - **Even** bits in one word
 - Odd** bits in a second word
 - $\text{ROT}_{64} \leftrightarrow 2 \times \text{ROT}_{32}$
- Can be generalized
 - to 16- and 8-bit words
- Can be combined
 - with lane/slice-wise architectures
 - with most other techniques



[KECCAK impl. overview, Section 2.1]

Interleaved lanes for 32-bit implementations



- Speed between SHA-256 and SHA-512
- Lower RAM usage

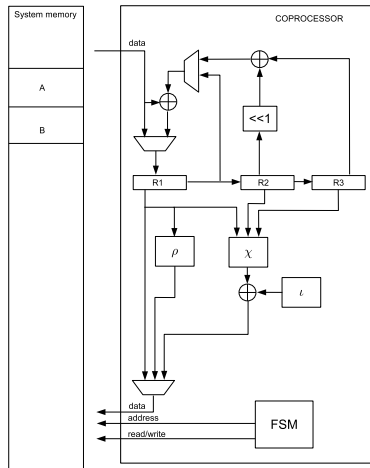
C/b	RAM	Algo	Strength
41	300	sha256	128
76	260	keccakc256*	128
94	260	keccakc512	256
173	916	sha512	256

[XBX, ARM Cortex-M3, <http://xbx.das-labor.org/>]

*estimated for $c = 256$

Lane-wise hardware architecture

- Basic processing unit + RAM
- Improvements over our co-processor:
 - 5 registers and barrel rotator [Kerckhof et al. CARDIS 2011]
 - 4-stage pipeline, ρ in 2 cycles, instruction-based parallel execution [San and At, ISJ 2012]
- Permutation latency in clock cycles:
 - From 5160, to 2137, down to 1062

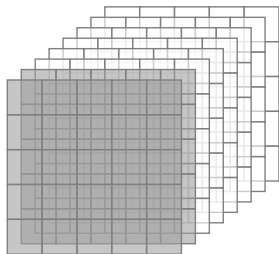


Slice-wise hardware architecture

- Re-schedule the execution
 - χ , θ , π and ι on blocks of slices
 - ρ by addressing

[Jungk et al, ReConFig 2011]
- Suitable for compact FPGA or ASIC
- Performance-area trade-offs
 - Possible to select number of processed slices from 1 up to 32

[VHDL on <http://keccak.noekeon.org/>]

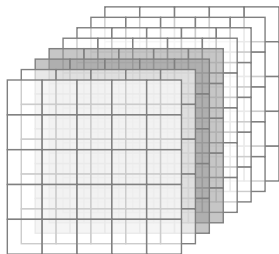


Slice-wise hardware architecture

- Re-schedule the execution
 - χ , θ , π and ι on blocks of slices
 - ρ by addressing

[Jungk et al, ReConFig 2011]
- Suitable for compact FPGA or ASIC
- Performance-area trade-offs
 - Possible to select number of processed slices from 1 up to 32

[VHDL on <http://keccak.noekeon.org/>]

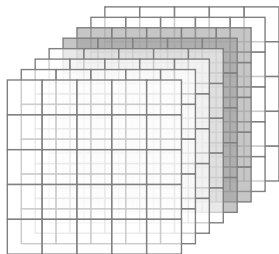


Slice-wise hardware architecture

- Re-schedule the execution
 - χ , θ , π and ι on blocks of slices
 - ρ by addressing

[Jungk et al, ReConFig 2011]
- Suitable for compact FPGA or ASIC
- Performance-area trade-offs
 - Possible to select number of processed slices from 1 up to 32

[VHDL on <http://keccak.noekeon.org/>]

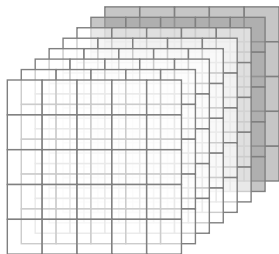


Slice-wise hardware architecture

- Re-schedule the execution
 - χ , θ , π and ι on blocks of slices
 - ρ by addressing

[Jungk et al, ReConFig 2011]
- Suitable for compact FPGA or ASIC
- Performance-area trade-offs
 - Possible to select number of processed slices from 1 up to 32

[VHDL on <http://keccak.noekeon.org/>]



Cutting the state in lanes or in slices?

- Both solutions are efficient, results for Virtex 5

Architecture	T.put Mbit/s	Freq. MHz	Slices (+RAM)	Latency clocks	Efficiency Mbit/s/slice
Lane-wise [1]	52	265	448	5160	0.12
Lane-wise [2]	501	520	151 (+3)	1062	3.32
Slice-wise [3]	813	159	372	200	2.19
High-Speed [4]	12789	305	1384	24	9.24

[1] Keccak Team, KECCAK implementation overview

[2] San, At, ISJ 2012

[3] Jungk, Apfelbeck, ReConFig 2011 (scaled to $r = 1024$)

[4] GMU ATHENa (scaled to $r = 1024$)

Outline

- 1 Implementing KECCAK (*how to cut a state*)
- 2 Power-attacking KECCAK
- 3 Power-protecting KECCAK

A model of the power consumption

Consumption at any time instance can be modeled as

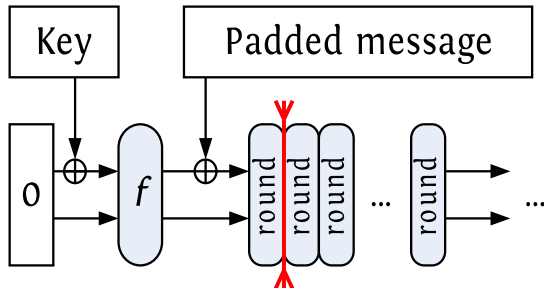
$$P = \sum_i T_i[d_i]$$

- d_i : Boolean variables that express *activity*
 - bit 1 in a given register or gate output at some stage
 - flipping of a specific register or gate output at some stage
- $T_i[0]$ and $T_i[1]$: stochastic variables

Simplified model

$$P = \alpha + \sum_i (-1)^{d_i}$$

DPA on a keyed sponge function



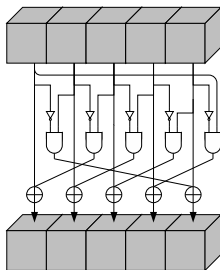
- 1 Attack the first round after absorbing known input bits
- 2 Compute backward by inverting the permutation

The KECCAK- f round function in a DPA perspective

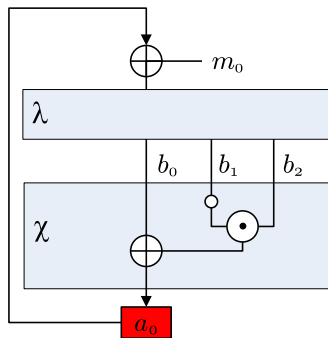
$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

- Linear part λ followed by non-linear part χ
- $\lambda = \pi \circ \rho \circ \theta$: mixing followed by bit transposition
- χ : simple mapping operating on rows:

$$b_i \leftarrow b_i + (b_{i+1} + 1)b_{i+2}$$

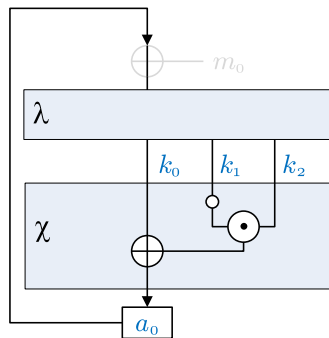


DPA applied to an unprotected implementation



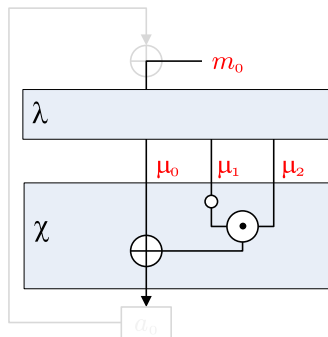
- Leakage exploited: switching consumption of **register bit 0**
- Value switches from a_0 to $b_0 + (b_1 + 1)b_2$
- Activity equation: $d = a_0 + b_0 + (b_1 + 1)b_2$

DPA applied to an unprotected implementation



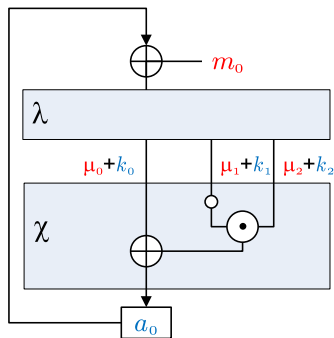
- Take the case $M = 0$
- We call K the input of χ -block if $M = 0$
- K will be our target

DPA applied to an unprotected implementation



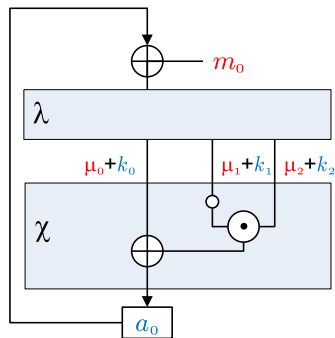
- We call the effect of M at input of χ : μ
- $\mu = \lambda(M||0^c)$
- Linearity of λ : $B = K + \lambda(M||0^c)$

DPA applied to an unprotected implementation



- $d = a_0 + k_0 + (k_1 + 1)(k_2) + \mu_0 + (\mu_1 + 1)\mu_2 + k_1\mu_2 + k_2\mu_1$
- Fact: value of $q = a_0 + k_0 + (k_1 + 1)k_2$ is same for all traces
- Let M_0 : traces with $d = q$ and M_1 : $d = q + 1$

DPA applied to an unprotected implementation



- Selection: $s(M, K^*) = \mu_0 + (\mu_1 + 1)\mu_2 + k_1^*\mu_2 + k_2^*\mu_1$
- Values of μ_1 and μ_2 computed from M
- Hypothesis has two bits only: k_1^* and k_2^*

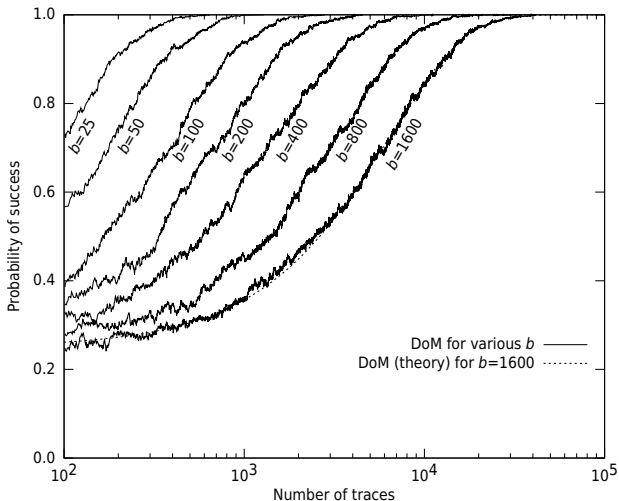
DPA applied to an unprotected implementation

- Correct hypothesis K
 - traces in M_0 : $d = q$
 - traces in M_1 : $d = q + 1$
- Incorrect hypothesis $K^* = K + \Delta$
 - trace in M_0 : $d = q + \mu_1\delta_2 + \mu_2\delta_1$
 - trace in M_1 : $d = q + \mu_1\delta_2 + \mu_2\delta_1 + 1$
- Remember: $\mu = \lambda(M||0^c)$
 - random inputs M lead to random μ_1 and μ_2
 - Incorrect hypothesis: d uncorrelated with $\{M_0, M_1\}$

Result of experiments

■ Analytical prediction of success probability possible

[Bertoni, Daemen, Debande, Le, Peeters, Van Assche, HASP 2012]



Outline

- 1 Implementing KECCAK (*how to cut a state*)
- 2 Power-attacking KECCAK
- 3 Power-protecting KECCAK**

Secret sharing

- Countermeasure at algorithmic level:
 - Split variables in *random* shares: $x = a \oplus b \oplus \dots$
 - Keep computed variables *independent* from *native* variables
 - Protection against n -th order DPA: at least $n + 1$ shares

Software: two-share masking

- $\chi : x_i \leftarrow x_i + (x_{i+1} + 1)x_{i+2}$ becomes:

$$\begin{aligned} a_i &\leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2} \\ b_i &\leftarrow b_i + (b_{i+1} + 1)b_{i+2} + b_{i+1}a_{i+2} \end{aligned}$$

- Independence from native variables, if:
 - we compute left-to-right
 - we avoid leakage in register or bus transitions
- $\lambda = \pi \circ \rho \circ \theta$ becomes:

$$\begin{aligned} a &\leftarrow \lambda(a) \\ b &\leftarrow \lambda(b) \end{aligned}$$

Software: two-share masking (faster)

- Making it **faster!**

- χ becomes:

$$\begin{aligned} a_i &\leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2} + (b_{i+1} + 1)b_{i+2} + b_{i+1}a_{i+2} \\ b_i &\leftarrow b_i \end{aligned}$$

- Precompute $R = b + \lambda(b)$

- $\lambda = \pi \circ \rho \circ \theta$ becomes:

$$\begin{aligned} a &\leftarrow \lambda(a) + R \\ b &\leftarrow b \end{aligned}$$

Software: two-share masking (faster)

- Making it **faster!**
- χ becomes:

$$a_i \leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2} + (b_{i+1} + 1)b_{i+2} + b_{i+1}a_{i+2}$$

- Precompute $R = b + \lambda(b)$
- $\lambda = \pi \circ \rho \circ \theta$ becomes:

$$a \leftarrow \lambda(a) + R$$

Hardware: two shares are not enough

- Unknown order in combinatorial logic!

$$a_i \leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2}$$

Using a threshold secret-sharing scheme

- Idea: **incomplete** computations only
 - Each circuit does not leak anything
[Nikova, Rijmen, Schläffer 2008]
- Number of shares: at least $1 + \text{algebraic degree}$
3 shares are needed for χ
- Glitches as second-order effect
 - A glitch can leak about two shares, say, $a + b$
 - Another part can leak c
 - \Rightarrow as if two shares only!

Using a threshold secret-sharing scheme

- Idea: **incomplete** computations only
 - Each circuit does not leak anything
[Nikova, Rijmen, Schläffer 2008]
- Number of shares: at least $1 + \text{algebraic degree}$
3 shares are needed for χ
- Glitches as second-order effect
 - A glitch can leak about two shares, say, $a + b$
 - Another part can leak c
 - \Rightarrow as if two shares only!

Using a threshold secret-sharing scheme

- Idea: **incomplete** computations only
 - Each circuit does not leak anything
[Nikova, Rijmen, Schläffer 2008]
- Number of shares: at least $1 + \text{algebraic degree}$
3 shares are needed for χ
- Glitches as second-order effect
 - A glitch can leak about two shares, say, $a + b$
 - Another part can leak c
 - \Rightarrow as if two shares only!

Three-share masking for χ

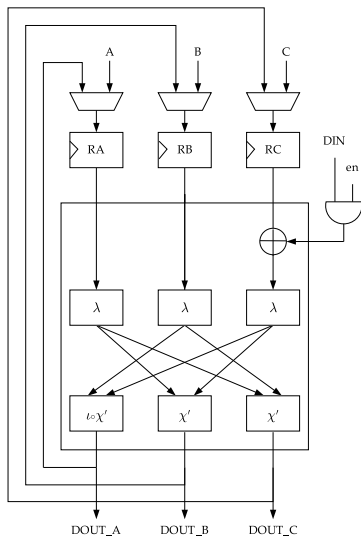
- Implementing χ in three shares:

$$a_i \leftarrow b_i + (b_{i+1} + 1)b_{i+2} + b_{i+1}c_{i+2} + c_{i+1}b_{i+2}$$

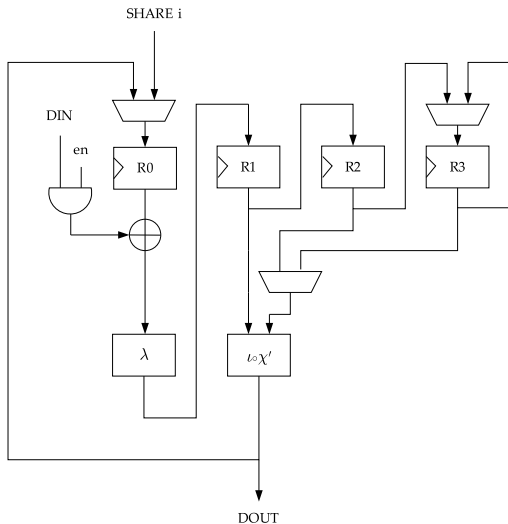
$$b_i \leftarrow c_i + (c_{i+1} + 1)c_{i+2} + c_{i+1}a_{i+2} + a_{i+1}c_{i+2}$$

$$c_i \leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2} + b_{i+1}a_{i+2}$$

One-cycle round architecture



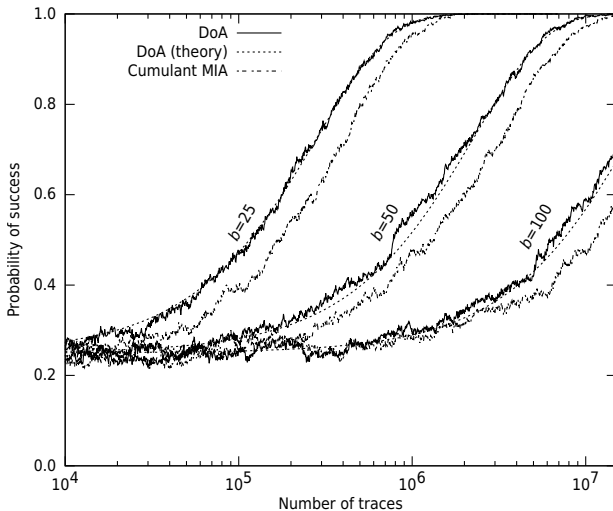
Three-cycle round architecture



Parallel vs sequential leakage

■ Generalization of results for protected implementation

[Bertoni, Daemen, Debande, Le, Peeters, Van Assche, HASP 2012]



Some references (1/2)

Main references

- Van Keer and KT, *Keccak implementation overview*
- Debande, Le and KT, *PA of HW impl. protected with secret sharing*, HASP 2012 + ePrint 2013/067

- *Note on side-channel attacks and their countermeasures*, NIST hash forum 2009
- *Building power analysis resistant implementations of KECCAK*, SHA-3 2010
- Software implementations and benchmarks
 - Bernstein and Lange, *eBASH*
 - Wenzel-Benner and Gräf, *XBX*
 - Balasch et al., *CARDIS* 2012

Optimized implementations available at
<http://keccak.noekeon.org/>

Some references (2/2)

- Hardware benchmarks and implementations on FPGA
 - Kerckhof et al., CARDIS 2011
 - Jungk and Apfelbeck, ReConFig 2011
 - San and At, ISJ 2012
 - Gaj et al.; Mahboob et al.; Kaps et al.; SHA-3 2012
- Hardware benchmarks and implementations on ASIC
 - Henzen et al., CHES 2010
 - Tillich et al., SHA-3 2010
 - Guo et al., DATE 2012
 - Gurkaynak et al.; Kavun et al.; SHA-3 2012

VHDL code available at
<http://keccak.noekeon.org/>

Questions?

