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

COSADE 2013, Paris, March 7-8, 2013
Outline

1. Zooming onto KECCAK
2. Implementing KECCAK \textit{(how to cut a state)}
3. Power-attacking KECCAK
4. Power-protecting KECCAK
The sponge construction

- More general than a hash function: arbitrary-length output
- Calls a $b$-bit permutation $f$, with $b = r + c$
  - $r$ bits of rate
  - $c$ bits of capacity $\Rightarrow 2^{c/2}$ generic security \cite{Eurocrypt 2008} and even better when keyed \cite{SKEW 2011}
Keccak

- Instantiation of a *sponge function*
- the permutation **Keccak-f**
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
- Security-speed trade-offs using the same permutation, e.g.,
  - SHA-3 instance: $r = 1088$ and $c = 512$
    - permutation width: 1600
    - security strength 256: post-quantum sufficient
  - Lightweight instance: $r = 40$ and $c = 160$
    - permutation width: 200
    - security strength 80: same as SHA-1
Use KECCAK for regular hashing

- Electronic signatures, message integrity (GPG, X.509 ...)
- Data integrity (shaxsum ...)
- Data identifier (Git, Mercurial, online anti-virus, peer-2-peer ...)

Diagram:

- Padded message
- Hash
  - f
  - f
  - f
  - f
  - f
  - f
Use KECCAK for MACing

- As a message authentication code
- Simpler than HMAC [FIPS 198]
  - HMAC: special construction for MACing with SHA-1 and SHA-2
  - Required to plug a security hole in SHA-1 and SHA-2
  - No longer needed for KECCAK which is sound
Use **KECCAK** for (stream) encryption

- As a stream cipher
Single pass authenticated encryption

- Authentication and encryption in a single pass!
- Secure messaging \((SSL/TLS, SSH, IPSEC \ldots)\)
- Same primitive KECCAK-\(f\) but in a (slightly) different mode
  - Duplex construction \([SAC 2011]\)
  - Also for random generation with reseeding \((/dev/urandom \ldots)\)
The state: an array of $5 \times 5 \times 2^\ell$ bits

- $5 \times 5$ lanes, each containing $2^\ell$ bits (1, 2, 4, 8, 16, 32 or 64)
- $(5 \times 5)$-bit slices, $2^\ell$ of them
The state: an array of $5 \times 5 \times 2^\ell$ bits

- $5 \times 5$ lanes, each containing $2^\ell$ bits (1, 2, 4, 8, 16, 32 or 64)
- $(5 \times 5)$-bit slices, $2^\ell$ of them
The state: an array of $5 \times 5 \times 2^\ell$ bits

- $5 \times 5$ lanes, each containing $2^\ell$ bits (1, 2, 4, 8, 16, 32 or 64)
- $(5 \times 5)$-bit slices, $2^\ell$ of them
The state: an array of $5 \times 5 \times 2^\ell$ bits

- $5 \times 5$ lanes, each containing $2^\ell$ bits (1, 2, 4, 8, 16, 32 or 64)
- $(5 \times 5)$-bit slices, $2^\ell$ of them
The state: an array of $5 \times 5 \times 2^\ell$ bits

- $5 \times 5$ lanes, each containing $2^\ell$ bits (1, 2, 4, 8, 16, 32 or 64)
- $(5 \times 5)$-bit slices, $2^\ell$ of them
θ for linear diffusion

- Compute parity $c_{x,z}$ of each column
- Add to each cell parity of neighboring columns:

$$b_{x,y,z} = a_{x,y,z} \oplus c_{x-1,z} \oplus c_{x+1,z-1}$$
θ for linear diffusion

KECCAK-F[b](A) {
    forall i in 0…nr-1
        A = Round[b](A, RC[i])
    return A
}

Round[b](A,RC) {

    θ step
    D[x] = C[x-1] xor rot(C[x+1],1),forall x in 0…4
    A[x,y] = A[x,y] xor D[x],forall (x,y) in (0…4,0…4)

    ρ and π steps
    B[y,2*x+3*y] = rot(A[x,y], r[x,y]),forall (x,y) in (0…4,0…4)

    χ step
    A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]),forall (x,y) in (0…4,0…4)

    ι step
    A[0,0] = A[0,0] xor RC

    return A
}

http://keccak.noekeon.org/specs_summary.html
\( \rho \) for inter-slice dispersion

- We need diffusion between the slices ...
- \( \rho \): cyclic shifts of lanes with offsets

\[
i(i + 1)/2 \mod 2^\ell
\]
π for disturbing horizontal/vertical alignment

\[ a_{x,y} \leftarrow a_{x',y'} \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} \]
Zooming onto Keccak

\[ \text{KECCAK-F}[b](A) \{
    \text{forall } i \text{ in } 0\ldots n_r - 1
    \quad A = \text{Round}[b](A, RC[i])
    \text{return } A
\}

Round[b](A, RC) \{
    \theta \text{ step}
    C[x] = A[x,0] \text{xor } A[x,1] \text{xor } A[x,2] \text{xor } A[x,3] \text{xor } A[x,4], \text{forall } x \text{ in } 0\ldots 4
    D[x] = C[x-1] \text{xor rot}(C[x+1], 1), \text{forall } x \text{ in } 0\ldots 4
    A[x,y] = A[x,y] \text{xor } D[x], \text{forall } (x,y) \text{ in } (0\ldots 4, 0\ldots 4)

    \rho \text{ and } \pi \text{ steps}
    B[y,2x+3y] = \text{rot}(A[x,y], r[x,y]), \text{forall } (x,y) \text{ in } (0\ldots 4, 0\ldots 4)

    \chi \text{ step}
    A[x,y] = B[x,y] \text{xor } ((\text{not } B[x+1,y]) \text{ and } B[x+2,y]), \text{forall } (x,y) \text{ in } (0\ldots 4, 0\ldots 4)

    \iota \text{ step}
    A[0,0] = A[0,0] \text{xor } RC

    \text{return } A
\}

http://keccak.noekeon.org/specs_summary.html
\( \chi \) for non-linearity

- “Flip bit if neighbors exhibit 01 pattern”
- Operates independently and in parallel on 5-bit rows
- Algebraic degree 2, inverse has degree 3
Zooming onto Keccak

χ for non-linearity

\[
\text{KECCAK-F[b]}(A) \{ \\
\text{forall } i \text{ in } 0…n_r-1 \\
\quad A = \text{Round[b]}(A, RC[i]) \\
\text{return } A \\
\}
\]

\[
\text{Round[b]}(A,RC) \{ \\
\text{θ step} \\
C[x] = A[x,0] \text{ xor } A[x,1] \text{ xor } A[x,2] \text{ xor } A[x,3] \text{ xor } A[x,4], \text{ forall } x \text{ in } 0…4 \\
D[x] = C[x-1] \text{ xor rot}(C[x+1], 1), \text{ forall } x \text{ in } 0…4 \\
A[x,y] = A[x,y] \text{ xor } D[x], \text{ forall } (x,y) \text{ in } (0…4,0…4) \\
\text{ρ and π steps} \\
B[y,2*x+3*y] = \text{rot}(A[x,y], r[x,y]), \text{ forall } (x,y) \text{ in } (0…4,0…4) \\
\text{χ step} \\
A[x,y] = B[x,y] \text{ xor ((not } B[x+1,y]) \text{ and } B[x+2,y]), \text{ forall } (x,y) \text{ in } (0…4,0…4) \\
\text{ι step} \\
A[0,0] = A[0,0] \text{ xor } RC \\
\text{return } A \\
\}
\]

http://keccak.noekeon.org/specs_summary.html
\( \iota \) for breaking the symmetry

- XOR of round-dependent constant to lane in origin.
- Without \( \iota \), the round mapping would be symmetric.
  - Invariant to translation in the z-direction.
- Without \( \iota \), all rounds would be the same.
  - Susceptibility to slide attacks.
  - Defective cycle structure.
- Without \( \iota \), we get simple fixed points (000 and 111).
### Zooming onto Keccak

For breaking the symmetry

**Keccak-F[b](A)**

```plaintext
forall i in 0…n_r-1
    A = Round[b](A, RC[i])
return A
```

**Round[b](A,RC)**

**θ step**

D[x] = C[x-1] xor rot(C[x+1],1),
A[x, y] = A[x, y] xor D[x],

**ρ and π steps**

B[y, 2*x+3*y] = rot(A[x, y], r[x, y]),

**χ step**

A[x, y] = B[x, y] xor ((not B[x+1, y]) and B[x+2, y]),

**ι step**

A[0, 0] = A[0, 0] xor RC

return A
```

http://keccak.noekeon.org/specs_summary.html
Outline

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

Not cutting it: straightforward hardware architecture

- Logic for one round + register for the state
  - very short critical path ⇒ high throughput
- Multiple rounds can be computed in a single clock cycle
  - 2, 3, 4 or 6 rounds in one shot
Implementing KECCAK (how to cut a state)

Lanes: straightforward software implementation

- Lanes fit in $2^\ell$-bit registers
  - 64-bit lanes for KECCAK-f[1600]
  - 8-bit lanes for KECCAK-f[200]

- Very basic operations required:
  - $\theta$ XOR and 1-bit rotations
  - $\rho$ rotations
  - $\pi$ just reading the correct words
  - $\chi$ XOR, AND, NOT
  - $\iota$ just a XOR
Implementing KECCAK (how to cut a state)

Lanes: straightforward software implementation

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

<table>
<thead>
<tr>
<th>C/b</th>
<th>Algo</th>
<th>Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.79</td>
<td>keccakc256treed2</td>
<td>128</td>
</tr>
<tr>
<td>4.98</td>
<td>md5</td>
<td>64</td>
</tr>
<tr>
<td>5.89</td>
<td>keccakc512treed2</td>
<td>256</td>
</tr>
<tr>
<td>6.09</td>
<td>sha1</td>
<td>80</td>
</tr>
<tr>
<td>8.25</td>
<td>keccakc256</td>
<td>128</td>
</tr>
<tr>
<td>10.02</td>
<td>keccakc512</td>
<td>256</td>
</tr>
<tr>
<td>13.73</td>
<td>sha512</td>
<td>256</td>
</tr>
<tr>
<td>21.66</td>
<td>sha256</td>
<td>128</td>
</tr>
</tbody>
</table>

[eBASH, hydra6, http://bench.cr.yp.to/]
Bit interleaving

- Ex.: map 64-bit lane to 32-bit words
  - $\rho$ 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]
Implementing KECCAK (how to cut a state)

Interleaved lanes for 32-bit implementations

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

<table>
<thead>
<tr>
<th>C/b</th>
<th>RAM</th>
<th>Algo</th>
<th>Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>300</td>
<td>sha256</td>
<td>128</td>
</tr>
<tr>
<td>76</td>
<td>260</td>
<td>keccakc256*</td>
<td>128</td>
</tr>
<tr>
<td>94</td>
<td>260</td>
<td>keccakc512</td>
<td>256</td>
</tr>
<tr>
<td>173</td>
<td>916</td>
<td>sha512</td>
<td>256</td>
</tr>
</tbody>
</table>


*estimated for c = 256
Implementing KECCAK (how to cut a state)

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, \( \rho \) 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
  - $\chi$, $\theta$, $\pi$ and $\iota$ on blocks of slices
  - $\rho$ 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
  - $\chi, \theta, \pi$ and $\iota$ on blocks of slices
  - $\rho$ 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/]
Re-schedule the execution
- $\chi$, $\theta$, $\pi$ and $\iota$ on blocks of slices
- $\rho$ 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
  - $\chi$, $\theta$, $\pi$ and $\iota$ on blocks of slices
  - $\rho$ 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/]
Implementing KECCAK (how to cut a state)

Cutting the state in lanes or in slices?

- Both solutions are efficient, results for Virtex 5

<table>
<thead>
<tr>
<th>Architecture</th>
<th>T.put Mbit/s</th>
<th>Freq. MHz</th>
<th>Slices (+RAM)</th>
<th>Latency clocks</th>
<th>Efficiency Mbit/s/slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane-wise [1]</td>
<td>52</td>
<td>265</td>
<td>448</td>
<td>5160</td>
<td>0.12</td>
</tr>
<tr>
<td>Lane-wise [2]</td>
<td>501</td>
<td>520</td>
<td>151 (+3)</td>
<td>1062</td>
<td>3.32</td>
</tr>
<tr>
<td>Slice-wise [3]</td>
<td>813</td>
<td>159</td>
<td>372</td>
<td>200</td>
<td>2.19</td>
</tr>
</tbody>
</table>

[1] Keccak Team, KECCAK implementation overview
[3] Jungk, Apfelbeck, ReConFig 2011 (scaled to $r = 1024$)
[4] GMU ATHENa (scaled to $r = 1024$)
Outline

1. Zooming onto KECCAK
2. Implementing KECCAK (*how to cut a state*)
3. Power-attacking KECCAK
4. 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 \( \lambda \) followed by non-linear part \( \chi \)
- \( \lambda = \pi \circ \rho \circ \theta \): mixing followed by bit transposition
- \( \chi \): 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: $d = a_0 + b_0 + (b_1 + 1)b_2$
Take the case $M = 0$

- We call $K$ the input of $\chi$-block if $M = 0$
- $K$ will be our target
DPA applied to an unprotected implementation

- We call the effect of $M$ at input of $\chi$: $\mu$
- $\mu = \lambda(M||0^c)$
- Linearity of $\lambda$: $B = K + \lambda(M||0^c)$
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 $\mu_1$ and $\mu_2$ 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 $\mu_1$ and $\mu_2$
  - 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. Zooming onto KECCAK
2. Implementing KECCAK \textit{(how to cut a state)}
3. Power-attacking KECCAK
4. Power-protecting KECCAK
Secret sharing

- Countermeasure at algorithmic level:
  - Split variables in *random* shares: $x = a \oplus b \oplus \ldots$
  - 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:
  
  \[
  a_i \leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2} \\
  b_i \leftarrow b_i + (b_{i+1} + 1)b_{i+2} + b_{i+1}a_{i+2}
  \]

- 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:
  
  \[
  a \leftarrow \lambda(a) \\
  b \leftarrow \lambda(b)
  \]
Making it faster!

\( \chi \) becomes:

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

\[
b_i \leftarrow b_i
\]

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

\( \lambda = \pi \circ \rho \circ \theta \) becomes:

\[
a \leftarrow \lambda(a) + R
\]

\[
b \leftarrow b
\]
Software: two-share masking (faster)

- Making it faster!
- $\chi$ 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 $1 +$ algebraic degree
  - 3 shares are needed for $\chi$

- 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 + algebraic degree
  3 shares are needed for $\chi$

- 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 + algebraic degree
  - 3 shares are needed for $\chi$

- **Glitches as second-order effect**
  - A glitch can leak about two shares, say, $a + b$
  - Another part can leak $c$
  - $\implies$ as if two shares only!
Three-share masking for $\chi$

- Implementing $\chi$ 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
Power-protecting Keccak

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

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?