Tuturial on KECCAK

Guido BERTONI[1] Joan DAEMEN[1]
Michaël PEETERS[2] Gilles VAN ASSCHE[1] Ronny VAN KEER[1]

[1]STMicroelectronics

[2]NXP Semiconductors

CHES 2014

Outline

Objective of the tutorial

- An overview of the Sponge construction and derivations
- How the KECCAK permutation has been designed
- How to implement KECCAK in HW, SW and with SCA protection
- How the SHA-3 standard is evolving

Symmetric cryptographic functions

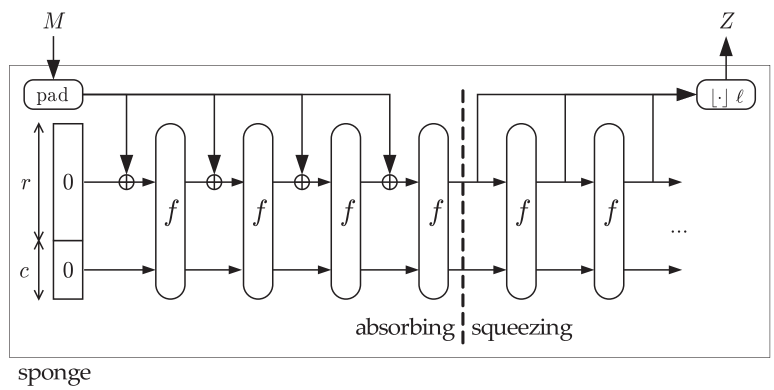
- Encryption
- Hashing
- Message authentication code
- Authenticated encryption
- Key derivation function
- Mask generation function
- PRNG

Outline

Contribution to permutation base crypto

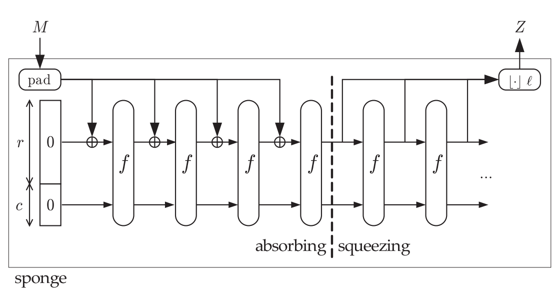
- Sponge [Ecrypt workshop 2007 and Eurocrypt 2008]
- Duplex [CHES and SHA-3 workshop 2010]
- Donkey Sponge and Monkey Duplex [DIAC 2012]
- HADDOC and MMB [SHA-3 2014]

The sponge construction



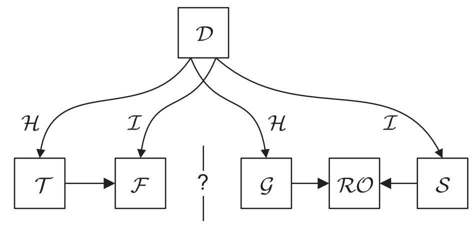
- More general than a hash function: arbitrary-length output
- Calls a b-bit permutation f, with $b = r + c$
- r bits of *rate*
- c bits of *capacity* (security parameter)

Generic security of the sponge construction



- RO-differentiating advantage $\leq N^2 / 2^{c+1}$
  - N is number of calls to f
  - opens up wide range of applications
- Bound assumes f is random permutation
  - It covers generic attacks
  - ...but not attacks that exploit specific properties of f

Generic security: indifferentiability [Maurer et al. (2004)]

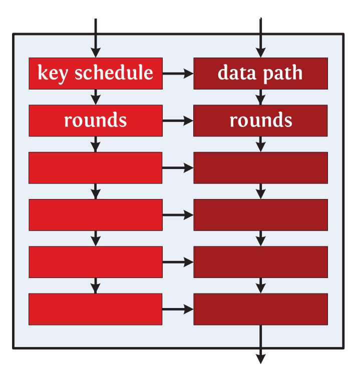


- Applied to hash functions in [Coron et al. (2005)]
  - distinguishing $\mathcal{T}$ from ideal function ($\mathcal{RO}$)
  - models adversary access to inner function f at left
  - f interface $\mathcal{I}$, covered by a *simulator* $\mathcal{S}$ at right
- Definition of differentiating advantage:

$$\Pr(\text{success}|\mathcal{D}) = \frac{1}{2} + \frac{1}{2}\text{Adv}(\mathcal{D})$$

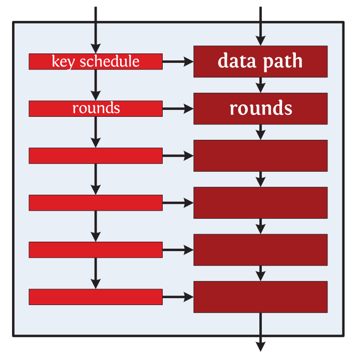
Block cipher versus permutation

- No diffusion from data path to key (and tweak) schedule
- Sometimes **lightweight** key schedule
- Let's remove these artificial barriers...
- That's a permutation!



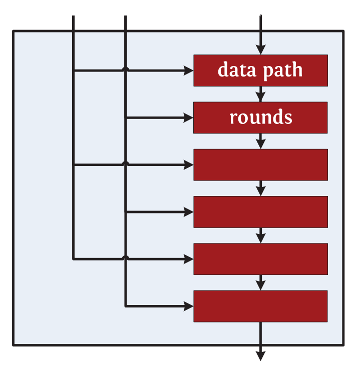
Block cipher versus permutation

- No diffusion from data path to key (and tweak) schedule
- Sometimes **lightweight** key schedule
- Let's remove these artificial barriers...
- That's a permutation!



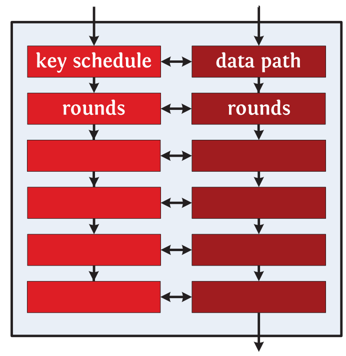
Block cipher versus permutation

- No diffusion from data path to key (and tweak) schedule
- Sometimes **lightweight** key schedule
- Let's remove these artificial barriers...
- That's a permutation!



Block cipher versus permutation

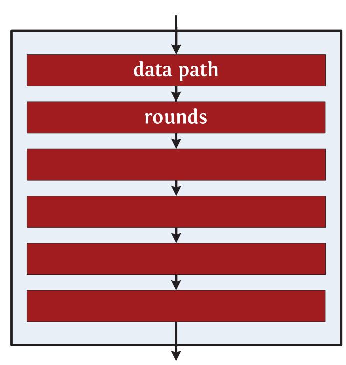
- No diffusion from data path to key (and tweak) schedule
- Sometimes **lightweight** key schedule
- Let's remove these artificial barriers...
- That's a permutation!



Permutation based crypto | Definition

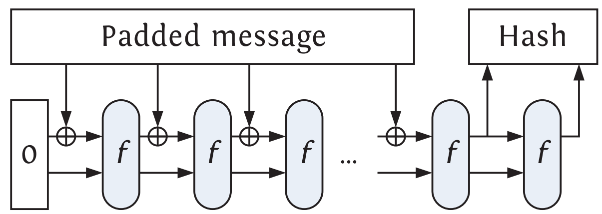
Block cipher versus permutation

- No diffusion from data path to key (and tweak) schedule
- Sometimes **lightweight** key schedule
- Let's remove these artificial barriers...
- That's a permutation!



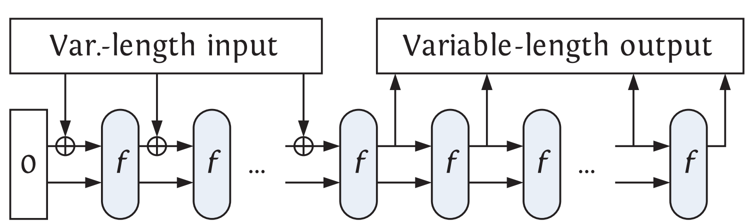
10 / 146

How to use a sponge function?



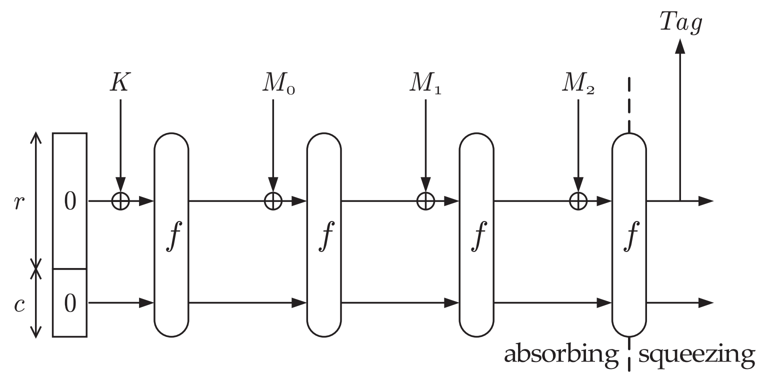
- For regular hashing

How to use a sponge function?



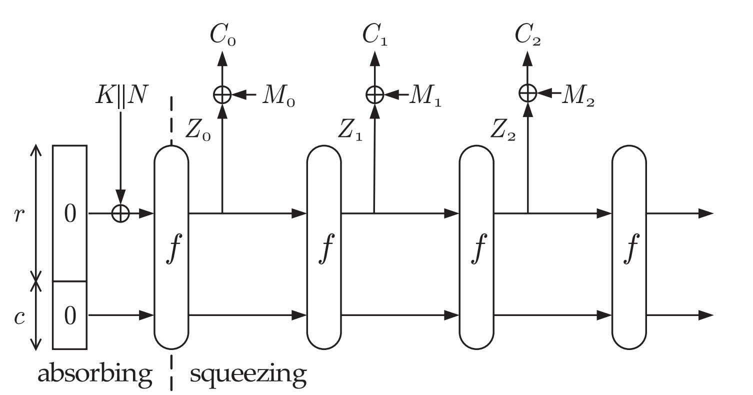
- As a mask generating function [PKCS#1, IEEE Std 1363a]

How to use a sponge function?



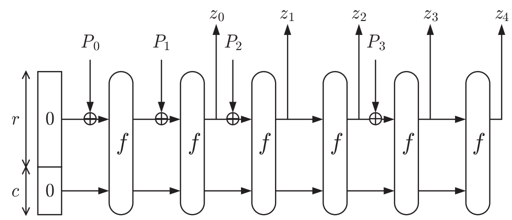
- As a message authentication code

How to use a sponge function?



- As a stream cipher

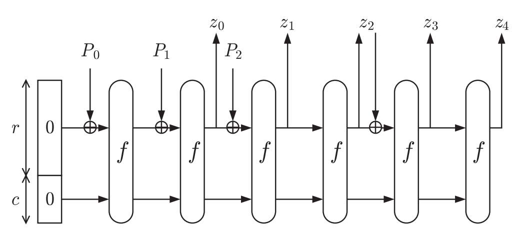
Sponge-based PRNG: the idea CHES2010



- **Feed** seeding (and reseeding) material P_i
- **Fetch** pseudo-random strings z_i
- Features:
  - f invertible $\Rightarrow$ no entropy loss
  - Forward secrecy: chop state by feeding back z_i

Permutation based crypto | Beyond Sponge: the Duplex construction

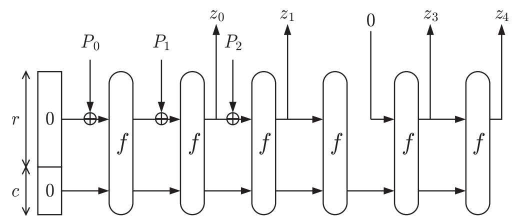
Sponge-based PRNG: the idea CHES2010



- **Feed** seeding (and reseeding) material P_i
- **Fetch** pseudo-random strings z_i
- Features:
  - f invertible $\Rightarrow$ no entropy loss
  - Forward secrecy: chop state by feeding back z_i

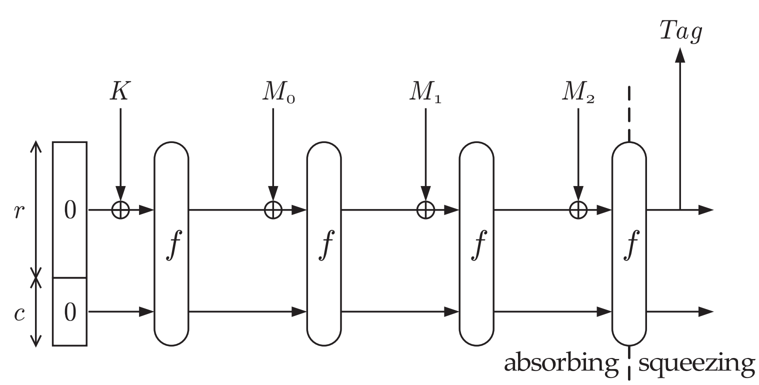
15 / 146

Sponge-based PRNG: the idea CHES2010

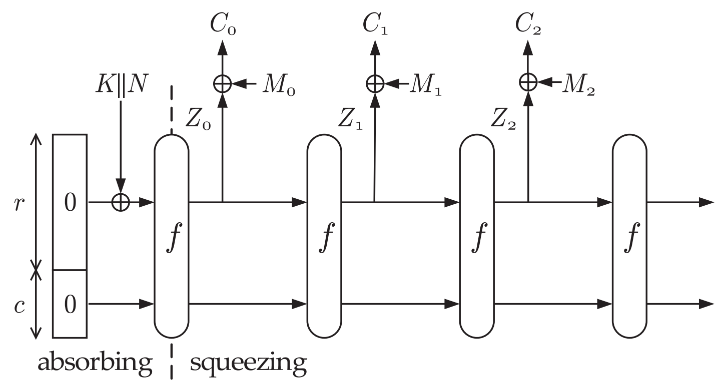


- **Feed** seeding (and reseeding) material P_i
- **Fetch** pseudo-random strings z_i
- Features:
  - f invertible $\Rightarrow$ no entropy loss
  - Forward secrecy: chop state by feeding back z_i

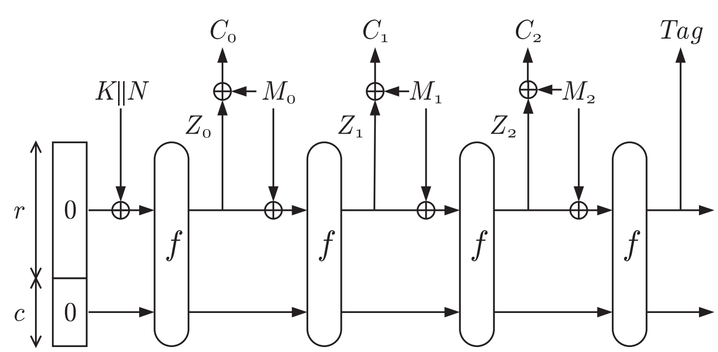
MAC generation with a sponge



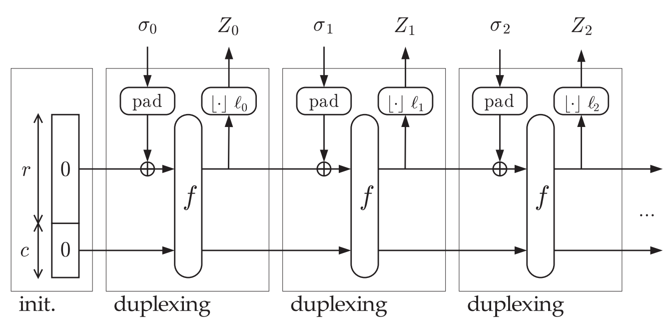
Encryption with a sponge



Both encryption and MAC?



The duplex construction



- Can be proven equivalent to Sponge
- Applications include:
  - Authenticated encryption
  - Reseedable pseudorandom sequence generator

Keyed Sponge

- When sponge is used with a key it is possible to have a better security compared to $2^{c/2}$
- The security goes with 2^{c-a} where a is a function of the number of observation available to the attacker [SKEW 2011 + work in progress]
- Recently Jovanovic et al. published "Beyond $2^{c/2}$ Security in Sponge-Based Authenticated Encryption Modes" [eprint373]

Sponge functions exists!

KECCAK	Bertoni, Daemen, Peeters, Van Assche	SHA-3 2008	25, 50, 100, 200 400, 800, 1600
Quark	Aumasson, Henzen, Meier, Naya-Plasencia	CHES 2010	136, 176 256, 384
Photon	Guo, Peyrin, Poschmann	Crypto 2011	100, 144, 196, 256, 288
Spongent	Bogdanov, Knezevic, Leander, Toz, Varici, Verbauwhede	CHES 2011	88, 136, 176 248, 320

And more in SHA-3 and CAESAR

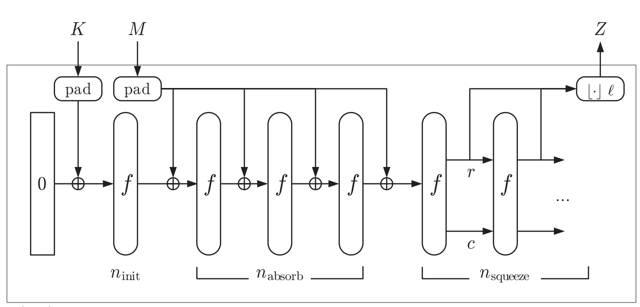
The lightweight taste

- Quark, Photon, Spongent: *lightweight hash functions*
- Easy to see why. Let us target security strength $2^{c/2}$
  - Davies-Meyer block cipher based hash (“narrow pipe”)
    - chaining value (block size): $n \geq c$
    - input block size (*key* length): typically $k \geq n$
    - feedforward (block size): n
    - total state $\geq 3c$
  - Sponge (“huge state”)
    - permutation width: $c + r$
    - r can be made arbitrarily small, e.g. 1 byte
    - total state $\geq c + 8$

Playing with the Sponge

- Sponge and Duplex, as presented right now, are rigid
  - fixed permutation
  - fixed rate (and capacity)
- Some optimizations are possible for improving performances
- in the different phases of the processing:
  - tuning number of rounds of the permutation
  - tuning rate (and capacity)

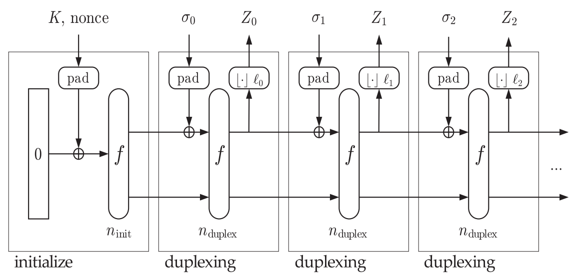
The donkeySponge MAC



donkey sponge

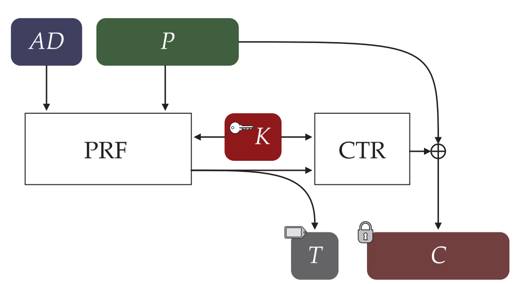
- Inspired by the Pelican MAC [DR, Pelican, 2005]:
- Usage of full state width b during absorbing
- Reduced number of rounds during init and absorbing

MonkeyDuplex



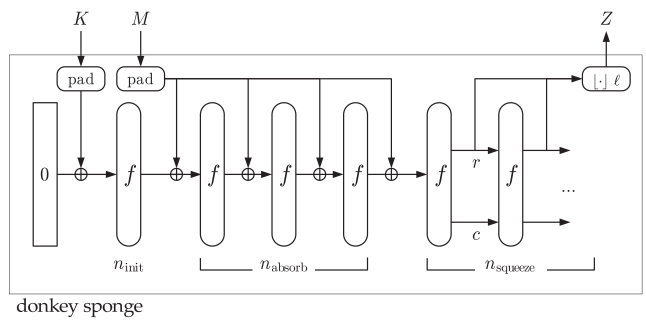
- For authenticated encryption and keystream generation [DIAC2012]
- Initialization: key, nonce and strong permutation
- Reduced number of rounds in duplex calls
- Warning: care should be taken in n_{duplex}

Haddoc: the concept



- Permutation-based variant of SIV [Rogaway, Shrimpton 2006]
- Nonceless
- Leakage limited to:
  - length of messages
  - identical messages (AD, P) give identical cryptograms (C, T)

Inside HADDOC: the DONKEYSPONGE PRF



- Absorbing phase exploits state secrecy [DR, Pelican, 2005]:
  - usage of full state width b
  - $n_{\text{init}} = 2$: make all state bits depend on the key
  - $n_{\text{absorb}} = 6$: limit max DP to prevent state collisions
- Squeezing phase: $n_{\text{squeeze}} = 12, c = 256$

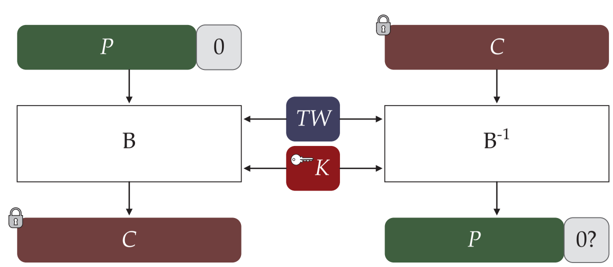
Building blocks of HADDOC

- PRF: DONKEYSPONGE with
  - input K: key
  - input M: injective coding of (AD, P)
  - output $T = \lfloor Z \rfloor_{256}$
- CTR: sponge in counter mode
  - single-block in- and outputs
  - $Z_i = \text{sponge}(K||T||i)$
  - $C_i = M_i \oplus Z_i$
  - $n_r = 12$
- Permutations
  - KECCAK-$p[1600, n_r]$
  - KECCAK-$p[800, n_r]$

Haddoc features

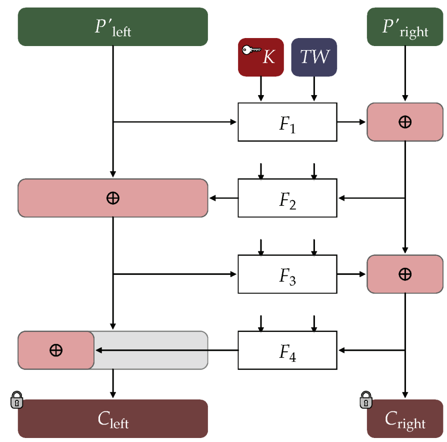
- Processing:
  - long messages: about 70 % of SHAKE128
  - short messages: 26 rounds
  - if *P* is absent we get a MAC function:
    - long messages: about 21 % of SHAKE128
    - short messages: 14 rounds
- Advantages
  - decryption: random access
  - encryption: PRF parallelizable
- Disadvantages
  - encryption strictly two-pass
  - message expansion by 2*n*-bit tag for *n*-bit security

Mr. Monster Burrito: the concept



- Robust AE [Rogaway, ACNS 2014]
  - inspired by AEZ [Hoang, Krovetz, Rogaway, Shrimpton 2014]
  - wide tweakable block cipher
  - variable key-, tweak- and blocksize: $|K|$, $|TW|$ and $|B|$
- *Best possible forgery resistance for given message expansion*

Inside MR. MONSTER BURRITO



- Based on [Naor Reingold 1997], thanks [DJB, Tenerife 2013]
  - F_2 and F_3: PRF
  - F_1 and F_4: constraint is max DP $< 2^{-256}$

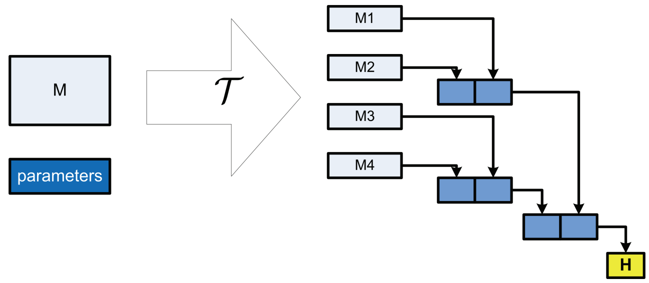
Building blocks of MR. MONSTER BURRITO

- Asymmetric Feistel: right part is single block
- F_i: DONKEYSPONGE instances as in HADDOC PRF
- F_i input:
  - K: key
  - M: injective coding of $(|B|, TW, S_{left/right}, i)$
- F_i output length:
  - F_1, F_3 and F_4: single-block
  - F_2: same length as left part
- Permutations
  - KECCAK-$p[1600, n_r]$
  - KECCAK-$p[800, n_r]$

MR. MONSTER BURRITO features

- Processing
  - block length above rate: close to 100% of SHAKE128
  - short block length: 56 rounds
- Advantages
  - minimum data expansion for given anti-forgery level
  - can even exploit redundancy in plaintext
- Disadvantages: *heavyweight crypto*
  - four-pass
  - inefficient for small block lengths

What is a tree hashing mode?



- Parameterized recipe to hash messages M by a number of calls to f
- Inner function f: compression function, hash function or extendable output function (XOF)
- Nodes Z_i: input strings to f composed of
  - message bits: taken from M
  - chaining bits: taken from $f(Z_j)$
  - frame bits: fully determined by $|M|$ and parameters

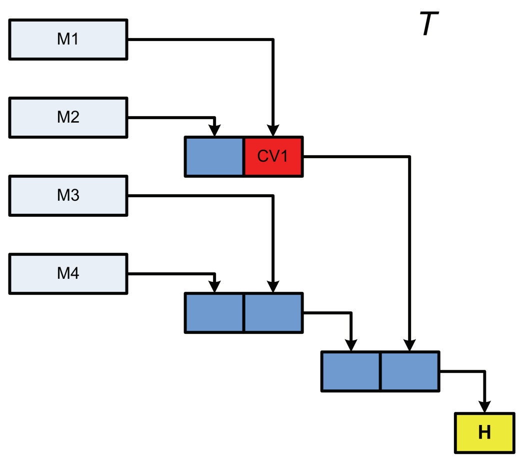
Functionality

- hash recomputation when modifying small part of file
- Merkle signature scheme
- peer-to-peer
  - networks like Gnutella
  - file sharing like BitTorrent
  - cryptocurrency like Bitcoin
  - distributed data store like Tahoe-LAFS
- ...

What we aspire to: random oracle $\mathcal{RO}$

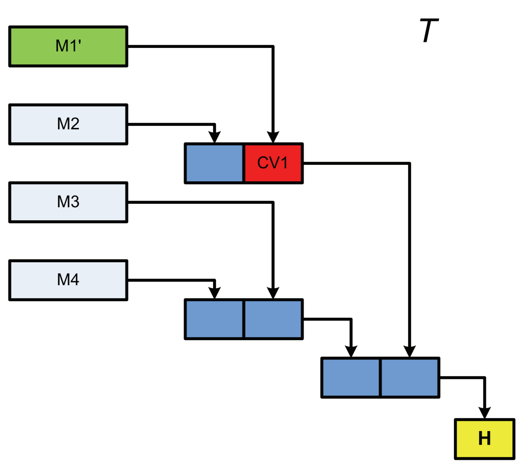
- A random oracle [Bellare-Rogaway 1993] maps:
  - message of arbitrary length
  - to an infinite output string
- Supports queries of following type: (M, ℓ)
  - M: message
  - ℓ: requested number of output bits
- Response H
  - ℓ independently and identically distributed bits
  - self-consistent: equal M give matching outputs
- Any deviation from this behaviour is considered bad news

f-collisions in inner nodes



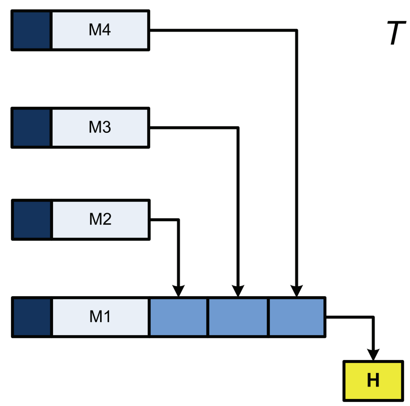
Permutation based crypto Tree hashing

f-collisions in inner nodes



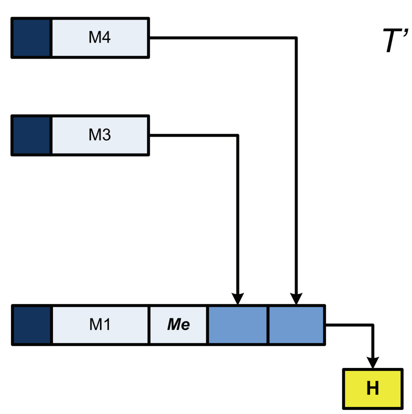
37 / 146

Collision in *H* without inner collision



Permutation based crypto Tree hashing

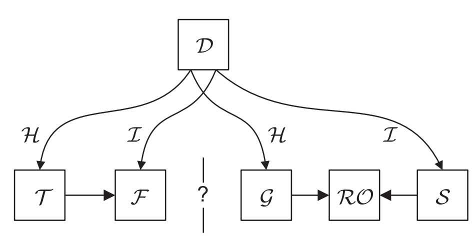
Collision in *H* without inner collision



Dealing with the problems

- Inner collisions
  - for $|CV| = n$, success probability $N^2 2^{-(n+1)}$ after N attempts
  - inevitably leads to inner collision after about $2^{n/2}$ attempts
- Other problems are avoided if following conditions are met:
  - final-node domain separation
  - tree-decodability: can *decode* (partial) hash trees
  - message-completeness: can reconstruct message from hash tree
- We call these the *three conditions for sound tree hashing*
- The best we can hope for is security strength $n/2$

Generic security: indifferentiability [Maurer et al. (2004)]



- Applied to hash functions in [Coron et al. (2005)]
  - distinguishing $\mathcal{T}$ from ideal function ($\mathcal{RO}$)
  - models adversary access to inner function f at left
  - f interface $\mathcal{I}$, covered by a *simulator* $\mathcal{S}$ at right
- Definition of differentiating advantage:

$$\Pr(\text{success}|\mathcal{D}) = \frac{1}{2} + \frac{1}{2}\text{Adv}(\mathcal{D})$$

Soundness of a tree hashing mode

Theorem

For any tree hashing mode $\mathcal{T}$ satisfying the three soundness conditions*:*

$$A \leq \frac{N^2}{2^{n+1}}$$

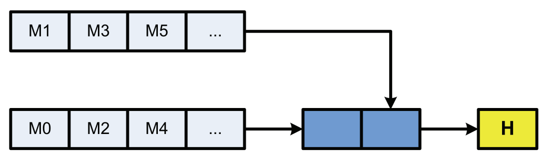
A: differentiating advantage of $\mathcal{T}$ from random oracle
N: number of calls of adversary to f
n: length of chaining values
*[Keccak team, ePrint 2009/210 — **last updated 2014**]*

- tight bound: success probability of generating inner collisions
- assumes f is ideal: security against generic attacks

SAKURA and tree hashing

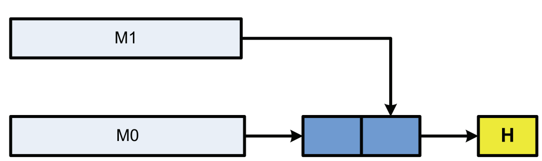
- Defining tree hash modes for all future use cases is infeasible
  - depth and degree of tree as a function of $|M|$?
  - or binary tree for saving intermediate hash results?
  - best choice strongly depends on specific requirement
- Define a tree hash *coding* instead: SAKURA
  - a way to code message blocks and chaining values in nodes
  - SAKURA-coding ensures 3 conditions for sound tree hashing
  - extends to all modes with SAKURA-compliant coding *together*
  - supported features:
    - any tree topology
    - message block interleaving
    - *kangaroo hopping*

Message block interleaving



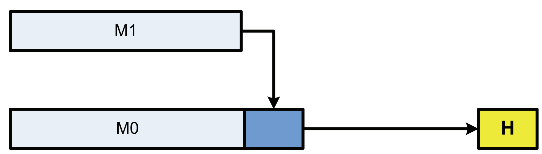
- Distribute the message data over parallel nodes as it arrives
- Multi-level, e.g.,
  - interleaved 64-bit pieces for SIMD
  - 1MB chunks for independent processes

Kangaroo hopping



- Append chaining value to other message block
- Reduces:
  - overhead of data to be processed by f
  - number of f evaluations

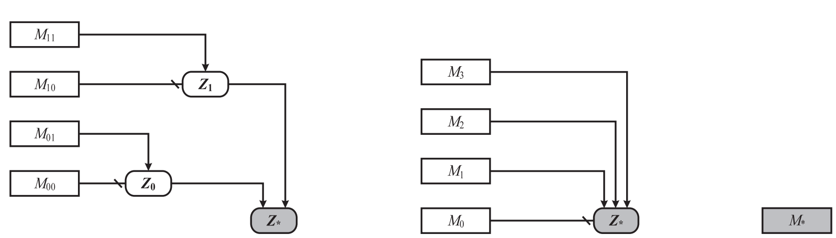
Kangaroo hopping



- Append chaining value to other message block
- Reduces:
  - overhead of data to be processed by f
  - number of f evaluations

SAKURA hops and nodes

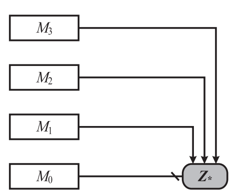
- Hops: hierarchy and interleaving
  - any tree topology
  - message hops: leaves
  - chaining hops: sequence of CVs and block interleaving info
- Nodes: inputs to f
  - final vs inner nodes
  - node contains 1 hop, followed by 0 to n chaining hops
- Examples:



SAKURA examples

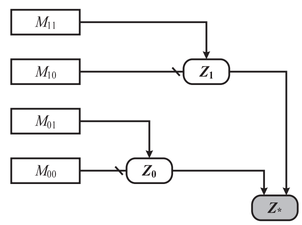
Node index	Encoding
*	M11

SAKURA examples



Node index	Encoding
2	$M_3 1\ 10^*\ 0$
1	$M_2 1\ 10^*\ 0$
0	$M_1 1\ 10^*\ 0$
*	$M_0 1\ 10^*\ CV_0\ CV_1\ CV_2$ 0x03 0x01 $\{I_*\}$0 1

SAKURA examples



Node index	Encoding
10	$M_{11}1\ 10^*\ 0$
1	$M_{10}1\ 10^*\ CV_{10}$ 0x01 0x01 $\{I_1\}0\ 10^*\ 0$
00	$M_{01}1\ 10^*\ 0$
0	$M_{00}1\ 10^*\ CV_{00}$ 0x01 0x01 $\{I_0\}0\ 10^*\ 0$
$*$	$CV_0\ CV_1$ 0x02 0x01 $\{I_*\}0\ 1$

Recap

- In this section we have seen how to:
  - Build symmetric key primitive based on permutation
  - How the Sponge construction allows to trade security and speed
  - Permutation-based nonce-less authenticated encryption
  - Sakura: flexible coding for tree hashing

Outline

KECCAK

- Instantiation of a *sponge function*
- Using the permutation KECCAK-f
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
    … from toy over lightweight to high-speed …
- Multi-rate padding 10*1
- SHA-3 instance
  - permutation width: 1600
  - from $c = 256$ to $c = 1024$
- Lightweight instance: $r = 40$ and $c = 160$
  - permutation width: 200
  - security strength 80: same as (initially expected from) SHA-1

See [The KECCAK reference] for more details

KECCAK

- Instantiation of a *sponge function*
- Using the permutation KECCAK-f
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
    ... from toy over lightweight to high-speed ...
- Multi-rate padding 10^*1
- SHA-3 instance
  - permutation width: 1600
  - from $c = 256$ to $c = 1024$
- Lightweight instance: $r = 40$ and $c = 160$
  - permutation width: 200
  - security strength 80: same as (initially expected from) SHA-1

See [The KECCAK reference] for more details

KECCAK

- Instantiation of a *sponge function*
- Using the permutation KECCAK-f
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
    ... from toy over lightweight to high-speed ...
- Multi-rate padding 10^*1
- SHA-3 instance
  - permutation width: 1600
  - from $c = 256$ to $c = 1024$
- Lightweight instance: $r = 40$ and $c = 160$
  - permutation width: 200
  - security strength 80: same as (initially expected from) SHA-1

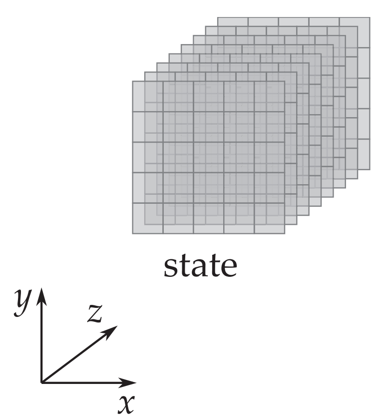
See [The KECCAK reference] for more details

KECCAK

- Instantiation of a *sponge function*
- Using the permutation KECCAK-f
  - 7 permutations: $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
    ... from toy over lightweight to high-speed ...
- Multi-rate padding 10^*1
- SHA-3 instance
  - permutation width: 1600
  - from $c = 256$ to $c = 1024$
- Lightweight instance: $r = 40$ and $c = 160$
  - permutation width: 200
  - security strength 80: same as (initially expected from) SHA-1

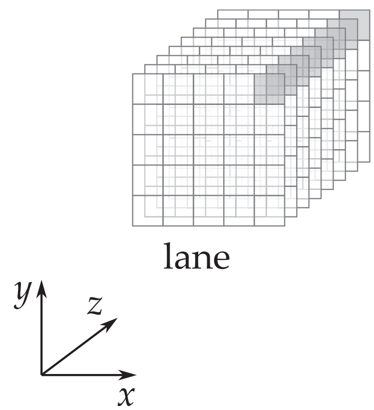
See [The KECCAK reference] for more details

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

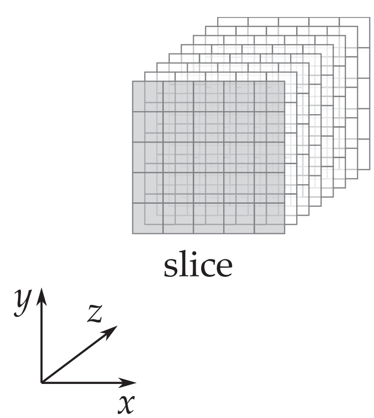
The state: an array of $5 \times 5 \times 2^\ell$ bits



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

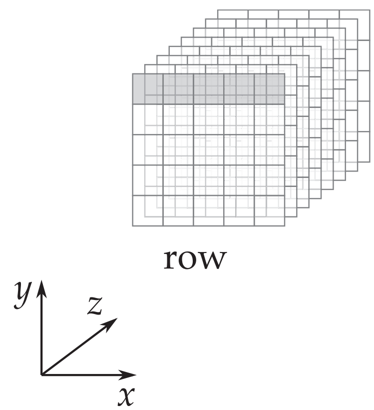
Keccak

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

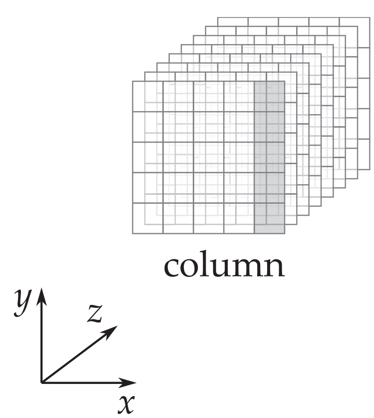
The state: an array of $5 \times 5 \times 2^\ell$ bits



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

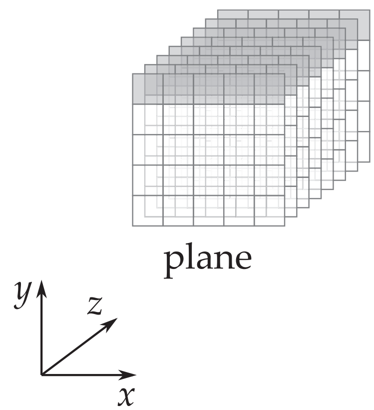
Keccak

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



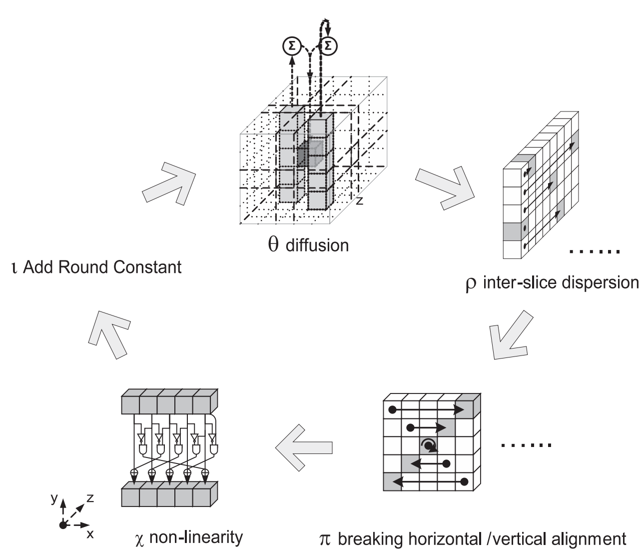
- 5×5 lanes, each containing 2^ℓ bits (1, 2, 4, 8, 16, 32 or 64)
- (5×5)-bit slices, 2^ℓ of them

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

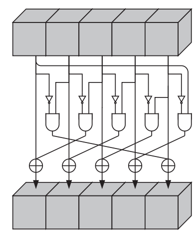


- 5×5 lanes, each containing 2^ℓ bits (1, 2, 4, 8, 16, 32 or 64)
- (5×5)-bit slices, 2^ℓ of them

The step mappings of KECCAK-*f*

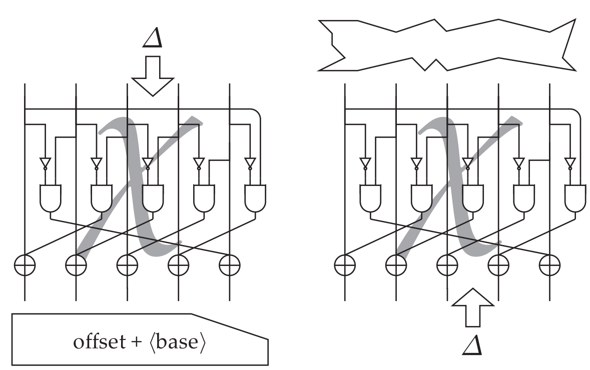


χ, the nonlinear mapping in KECCAK-f



- “Flip bit if neighbors exhibit 01 pattern”
- Operates independently and in parallel on 5-bit rows
- Cheap: small number of operations per bit
- Algebraic degree 2, inverse has degree 3
- LC/DC propagation properties easy to describe and analyze

Propagating differences through χ



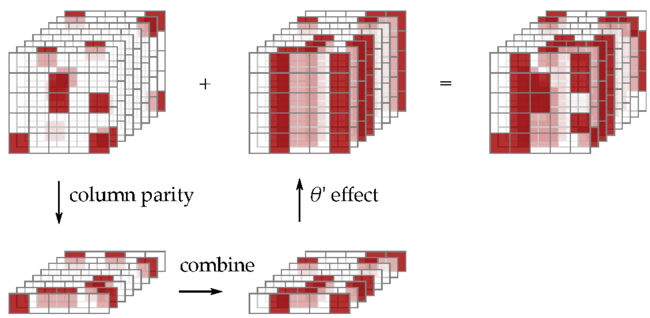
- The propagation weight…
  - … is equal to $-\log_2$(fraction of pairs);
  - … is determined by input difference only;
  - … is the size of the affine base;
  - … is the number of affine conditions.

θ', a first attempt at mixing bits

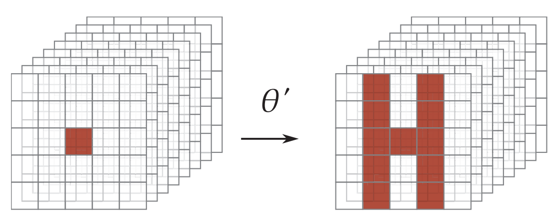
- Compute parity $c_{x,z}$ of each column
- Add to each cell parity of neighboring columns:

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

- Cheap: two XORs per bit

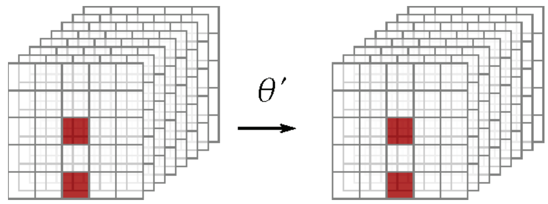


Diffusion of θ'



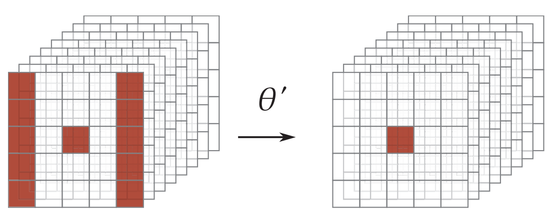
$$1 + \left(1 + y + y^2 + y^3 + y^4\right)\left(x + x^4\right)$$
$$\left(\bmod \left\langle 1 + x^5, 1 + y^5, 1 + z^w \right\rangle\right)$$

Diffusion of θ' (kernel)



$$1 + \left(1 + y + y^2 + y^3 + y^4\right)\left(x + x^4\right)$$
$$\left(\bmod \left\langle 1 + x^5, 1 + y^5, 1 + z^w \right\rangle\right)$$

Diffusion of the inverse of θ'



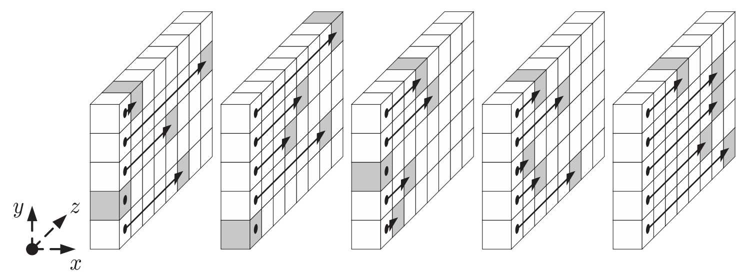
$$1 + \left(1 + y + y^2 + y^3 + y^4\right)\left(x^2 + x^3\right)$$
$$\left(\bmod \left\langle 1 + x^5, 1 + y^5, 1 + z^w \right\rangle\right)$$

ρ for inter-slice dispersion

- We need diffusion between the slices …
- ρ: cyclic shifts of lanes with offsets

$$i(i+1)/2 \bmod 2^{\ell}, \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^{i-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

- Offsets cycle through all values below 2^{ℓ}

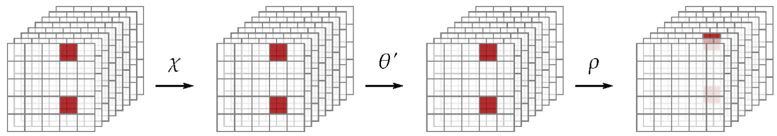


ι to break symmetry

- XOR of round-dependent constant to lane in origin
- Without ι, the round mapping would be symmetric
  - invariant to translation in the z-direction
  - susceptible to *rotational* cryptanalysis
- Without ι, all rounds would be the same
  - susceptibility to *slide* attacks
  - defective cycle structure
- Without ι, we get simple fixed points (000 and 111)

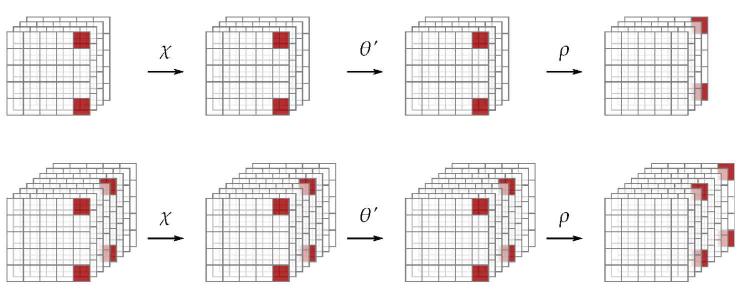
A first attempt at KECCAK-f

- Round function: $\mathrm{R} = \iota \circ \rho \circ \theta' \circ \chi$
- Problem: low-weight periodic trails by chaining:



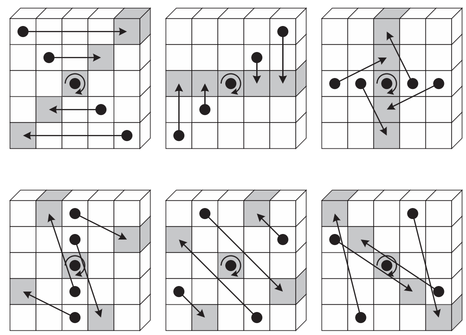
  - χ: propagates unchanged with weight 4
  - θ': propagates unchanged, because all column parities are 0
  - ρ: in general moves active bits to different slices ...
    ...but not always

The Matryoshka property



- Patterns in Q' are z-periodic versions of patterns in Q
- Weight of trail Q' is twice that of trail Q (or 2^n times in general)

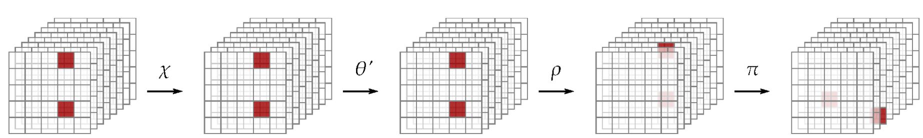
π for disturbing horizontal/vertical alignment



$$a_{x,y} \leftarrow a_{x',y'} \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

A second attempt at KECCAK-f

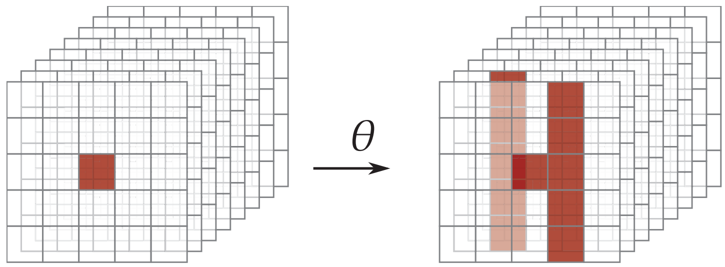
- Round function: $R = \iota \circ \pi \circ \rho \circ \theta' \circ \chi$
- Solves problem encountered before:



- π moves bits in same column to different columns!

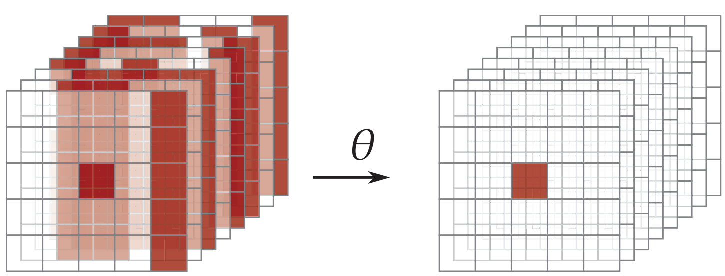
Almost there, still a final tweak ...

Tweaking θ' to θ



$$1 + \left(1 + y + y^2 + y^3 + y^4\right)\left(x + x^4 z\right)$$
$$\left(\text{mod } \left\langle 1 + x^5, 1 + y^5, 1 + z^w \right\rangle\right)$$

Inverse of θ



$$1 + \left(1 + y + y^2 + y^3 + y^4\right) \mathbf{Q},$$
$$\text{with } \mathbf{Q} = 1 + \left(1 + x + x^4 z\right)^{-1} \text{ mod } \left\langle 1 + x^5, 1 + z^w \right\rangle$$

- **Q** is dense, so:
  - Diffusion from single-bit output to input very high
  - Increases resistance against LC/DC and algebraic attacks

Keccak-*f* summary

- Round function:

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

- Number of rounds: $12 + 2\ell$
  - Keccak-$f[25]$ has 12 rounds
  - Keccak-$f[1600]$ has 24 rounds
- Keccak-$f[b]$ vs Keccak-$p[b, n_r]$ [FIPS 202 draft, 2014]

Recap

- In this section we have seen:
  - The internal design of Keccak-*f*
  - The 5 step mappings
  - The structure of the state

Outline

CAESAR

Competition for Authenticated Encryption: Security, Applicability, and Robustness

- M0, 2014.03.15: submissions.
- M10: round-2 candidates.
- M21: round-3 candidates.
- M33: finalists.
- M45: portfolio.

See http://competitions.cr.yp.to/caesar.html

Overview

- Inspired by KECCAK and DUPLEX
- KEYAK targeting high performances
  - Using reduced-round KECCAK-f[1600] or KECCAK-f[800]
  - Optionally parallelizable
- KETJE targeting lightweight
  - Using reduced-round KECCAK-f[400] or KECCAK-f[200]

CAESAR

Overview

- Inspired by KECCAK and DUPLEX
- KEYAK targeting high performances
  - Using reduced-round KECCAK-f[1600] or KECCAK-f[800]
  - Optionally parallelizable
- KETJE targeting lightweight
  - Using reduced-round KECCAK-f[400] or KECCAK-f[200]

72 / 146

Overview

- Inspired by KECCAK and DUPLEX
- KEYAK targeting high performances
  - Using reduced-round KECCAK-$f[1600]$ or KECCAK-$f[800]$
  - Optionally parallelizable
- KETJE targeting lightweight
  - Using reduced-round KECCAK-$f[400]$ or KECCAK-$f[200]$

Two approaches

KEYAK:	KETJE:
DUPLEXWRAP	MONKEYWRAP
A (strong) permutation	A (thin) round function
– fixed #rounds	– #rounds in phases
Block-oriented	Stream-oriented
Cryptanalysis	Cryptanalysis
– permutation-level	– round function + construction

Two approaches

KEYAK:

- DUPLEXWRAP
- A (strong) permutation
  - fixed #rounds
- Block-oriented
- Cryptanalysis
  - permutation-level

KETJE:

- MONKEYWRAP
- A (thin) round function
  - #rounds in phases
- Stream-oriented
- Cryptanalysis
  - round function + construction

Key Pack

The purpose of the key pack is to have a uniform way of encoding a secret key as prefix of a string input.

$$\text{keypack}(K, l) = \text{enc}_8(l/8)||K||\text{pad10}^*[l-8](|K|),$$

That is, the key pack consists of

- a first byte indicating its whole length in bytes, followed by
- the key itself, followed by
- simple padding.

For instance, the 32-bit key $K =$ 0x01 0x23 0x45 0x67 yields

$$\text{keypack}(K, 64) = \text{0x08 0x01 0x23 0x45 0x67 0x01 0x00}^2.$$

[Keyak spec.]

KEYAK goals

- Nonce-based AE function
- 128-bit security (incl. multi-target)
- Sequence of header-body pairs
  - keeping the state during the session
- Optionally parallelizable
- Using reduced-round KECCAK-$f[1600]$ or KECCAK-$f[800]$, to allow
  - implementation re-use
  - cryptanalysis re-use
  - reasonable side-channel protections

(... and because we like it ...)

CAESAR

KEYAK goals

- Nonce-based AE function
- 128-bit security (incl. multi-target)
- Sequence of header-body pairs
  - keeping the state during the session
- Optionally parallelizable
- Using reduced-round KECCAK-$f[1600]$ or KECCAK-$f[800]$, to allow
  - implementation re-use
  - cryptanalysis re-use
  - reasonable side-channel protections

(... and because we like it ...)

75 / 146

KEYAK goals

- Nonce-based AE function
- 128-bit security (incl. multi-target)
- Sequence of header-body pairs
  - keeping the state during the session
- Optionally parallelizable
- Using reduced-round KECCAK-f[1600] or KECCAK-f[800], to allow
  - implementation re-use
  - cryptanalysis re-use
  - reasonable side-channel protections

(... and because we like it ...)

KEYAK goals

- Nonce-based AE function
- 128-bit security (incl. multi-target)
- Sequence of header-body pairs
  - keeping the state during the session
- Optionally parallelizable
- Using reduced-round KECCAK-f[1600] or KECCAK-f[800], to allow
  - implementation re-use
  - cryptanalysis re-use
  - reasonable side-channel protections

(... and because we like it ...)

KEYAK goals

- Nonce-based AE function
- 128-bit security (incl. multi-target)
- Sequence of header-body pairs
  - keeping the state during the session
- Optionally parallelizable
- Using reduced-round KECCAK-f[1600] or KECCAK-f[800], to allow
  - implementation re-use
  - cryptanalysis re-use
  - reasonable side-channel protections

(... and because we like it ...)

KEYAK goals

- Nonce-based AE function
- 128-bit security (incl. multi-target)
- Sequence of header-body pairs
  - keeping the state during the session
- Optionally parallelizable
- Using reduced-round KECCAK-f[1600] or KECCAK-f[800], to allow
  - implementation re-use
  - cryptanalysis re-use
  - reasonable side-channel protections

(... and because we like it ...)

KEYAK goals

- Nonce-based AE function
- 128-bit security (incl. multi-target)
- Sequence of header-body pairs
  - keeping the state during the session
- Optionally parallelizable
- Using reduced-round KECCAK-f[1600] or KECCAK-f[800], to allow
  - implementation re-use
  - cryptanalysis re-use
  - reasonable side-channel protections

(... and because we like it ...)

CAESAR

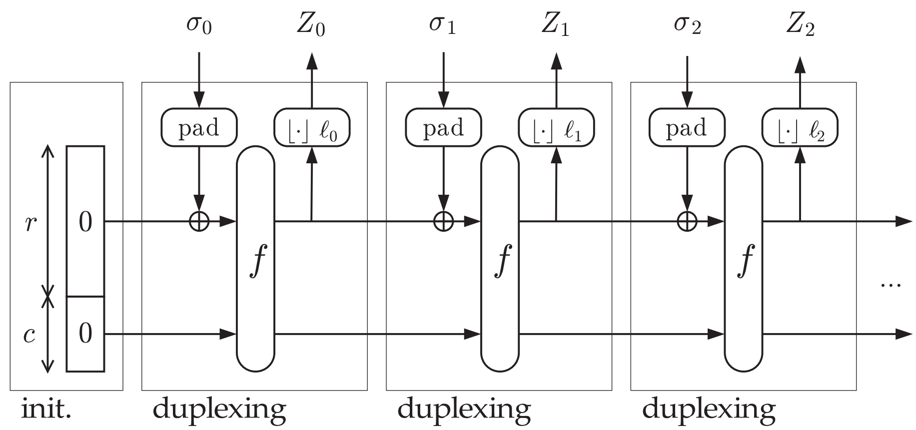
KEYAK goals

- Nonce-based AE function
- 128-bit security (incl. multi-target)
- Sequence of header-body pairs
  - keeping the state during the session
- Optionally parallelizable
- Using reduced-round KECCAK-f[1600] or KECCAK-f[800], to allow
  - implementation re-use
  - cryptanalysis re-use
  - reasonable side-channel protections

(... and because we like it ...)

75 / 146

Duplex layer

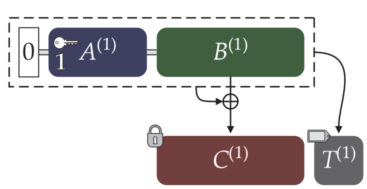


KECCAK-$p[1600, n_r = 12]$ or KECCAK-$p[800, n_r = 12]$

DUPLEXWRAP layer

DUPLEXWRAP

- is a nonce-based authenticated encryption mode;
- works on sequences of header-body pairs.



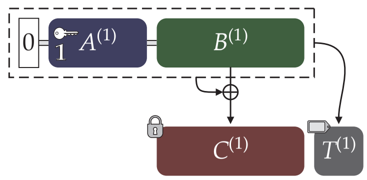
$A^{(1)}$ contains the key and must be unique, e.g.,

- $A^{(1)}$ contains a session key used only once;
- $A^{(1)}$ contains a key and a nonce.

In general: $A^{(1)} = \text{key}||\text{nonce}||\text{associated data}$.

DuplexWrap layer

DuplexWrap
- is a nonce-based authenticated encryption mode;
- works on sequences of header-body pairs.



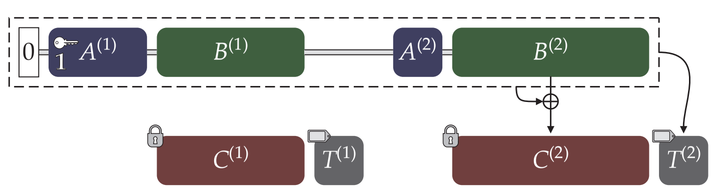
$A^{(1)}$ contains the key and must be unique, e.g.,
- $A^{(1)}$ contains a session key used only once;
- $A^{(1)}$ contains a key and a nonce.

In general: $A^{(1)} = \text{key}||\text{nonce}||\text{associated data}$.

CAESAR

DuplexWrap layer

DuplexWrap
- is a nonce-based authenticated encryption mode;
- works on sequences of header-body pairs.



$A^{(1)}$ contains the key and must be unique, e.g.,
- $A^{(1)}$ contains a session key used only once;
- $A^{(1)}$ contains a key and a nonce.

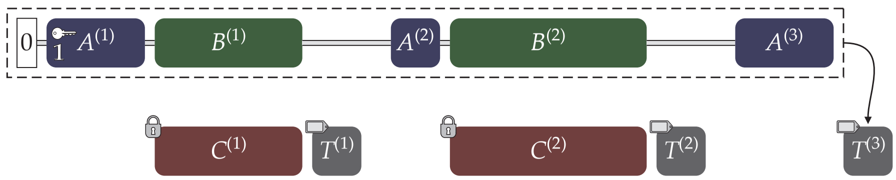
In general: $A^{(1)} = \text{key}||\text{nonce}||\text{associated data}$.

77 / 146

DuplexWrap layer

DuplexWrap

- is a nonce-based authenticated encryption mode;
- works on sequences of header-body pairs.

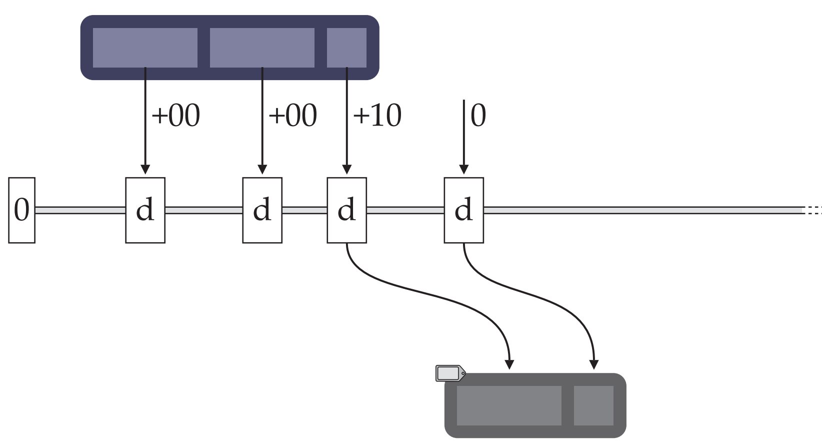


$A^{(1)}$ contains the key and must be unique, e.g.,

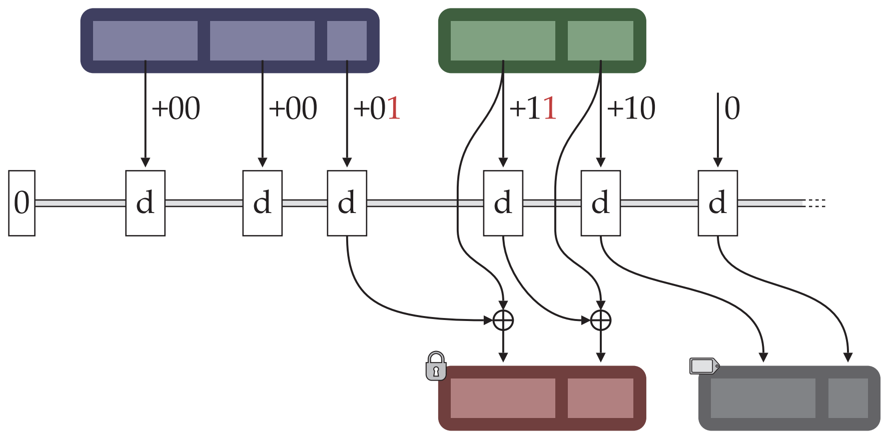
- $A^{(1)}$ contains a session key used only once;
- $A^{(1)}$ contains a key and a nonce.

In general: $A^{(1)}$ = key||nonce||associated data.

Inside DuplexWrap



Inside DuplexWrap



Keyak instances and efficiency

Name	Width b	Parallelism P
Ocean Keyak	1600	4
Sea Keyak	1600	2
Lake Keyak	1600	1
River Keyak	800	1

- Processing for Lake Keyak
  - long messages: about 50 % of SHAKE128
  - short messages: 24 rounds
- Working memory footprint
  - reasonable on high- and middle-end platforms
  - not ideal on constrained platforms

Security of Keyak

Generic security of Keyak thanks to a combination of results:

- Sound tree hashing modes [IJIS 2013] for parallelized modes
- Keyed sponge indistinguishability [SKEW 2011 + work in progress]
- SpongeWrap generic security [SAC 2011], adapted to DuplexWrap

Safety margin against shortcut attacks:

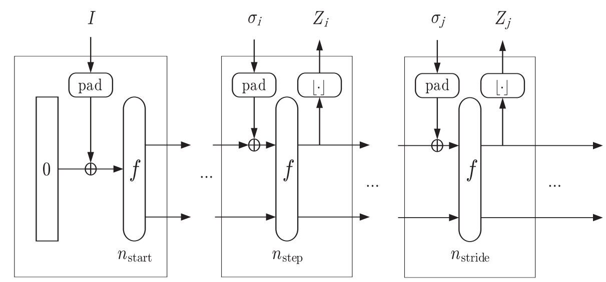
- Practical attacks up to 6 rounds [Dinur et al. SHA-3 2014]
- Academic attacks up to 9 rounds [Dinur et al. SHA-3 2014]

Ketje goals

- Nonce-based AE function
- 96-bit or 128-bit security (incl. multi-target)
- Sequence of header-body pairs
  - keeping the state during the session
- Small footprint
- Target niche: secure channel protocol on secure chips
  - banking card, ID, (U)SIM, secure element, FIDO, etc.
  - secure chip has strictly incrementing counter
- Using reduced-round Keccak-$f[400]$ or Keccak-$f[200]$, to allow
  - implementation re-use
  - cryptanalysis re-use
  - reasonable side-channel protections

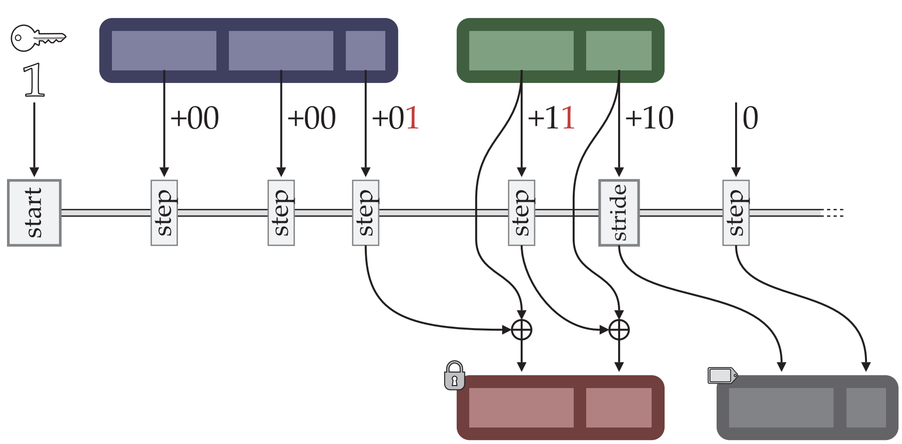
(... and because we like it ...)

Inside KETJE: the MONKEYDUPLEX layer



- $n_{start} = 12$ rounds should provide strong instance separation
- $n_{step} = 1$, $r = 2b/25$ should avoid single-instance state retrieval
- $n_{stride} = 6$ rounds should avoid a forgery with one instance

Inside MONKEYWRAP



Ketje instances and lightweight features

feature		Ketje Jr	Ketje Sr
state size		25 bytes	50 bytes
block size		2 bytes	4 bytes
processing		**computational cost**	
initialization	per session	12 rounds	12 rounds
wrapping	per block	1 round	1 round
8-byte tag comp.	per message	9 rounds	7 rounds

Current developments

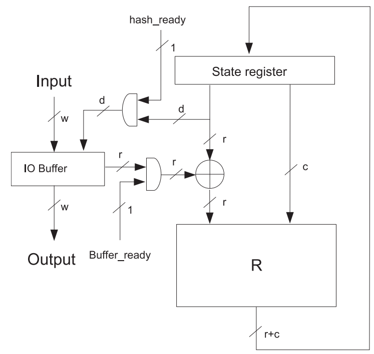
- Optimized software implementations
  - Gross estimations can be derived from Keccak
  - Lake Keyak expected twice faster than SHAKE128
  - There might be interesting improvement with new AVX512 (VPTERNLOG, rotations and 32 registers)
- Hardware implementations

Recap

- In this section we have seen:
  - The two proposal for CAESAR KETJE and KEYAK

Outline

Straightforward hardware architecture



- Logic for one round + register for the state
  - very short critical path $\Rightarrow$ high throughput

Multiple round per clock cycle

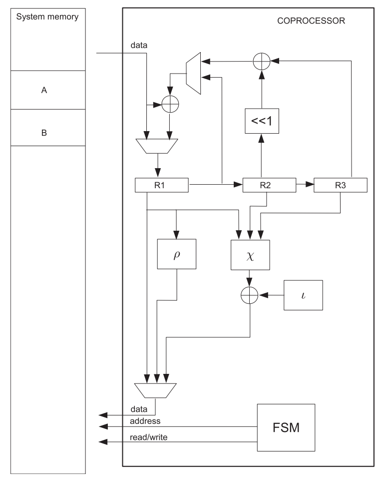
Num of round	Size	Critical Path	Frequency	Throughput
$n = 1$	48 kgates	1.9 ns	526 MHz	29.45 Gbit/s
$n = 2$	67 kgates	3.0 ns	333 MHz	37.29 Gbit/s
$n = 3$	86 kgates	4.1 ns	244 MHZ	40.99 Gbit/s
$n = 4$	105 kgates	5.2 ns	192 MHz	43.00 Gbit/s
$n = 6$	143 kgates	6.3 ns	135 MHZ	45.36 Gbit/s

- Multiple rounds can be computed in a single clock cycle
  - 2, 3, 4 or 6 rounds in one shot
  - but you have to feed the beast....
  - input throughput up to of 336 bits per clock cycle

Data related to STM 130 nm and rate = 1344

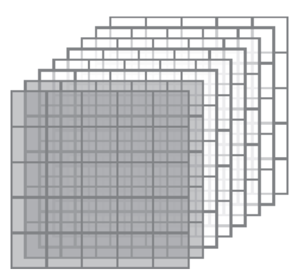
Lane-wise hardware architecture

- 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
- Area is in the order of 10kgate including RAM



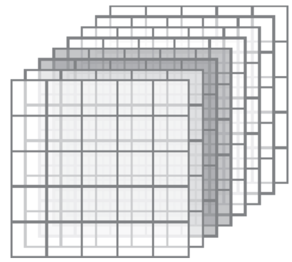
Slice-wise hardware architecture

- 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

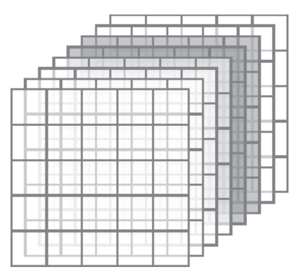
- Re-schedule the execution
  - χ, θ, π and ι on blocks of slices
  - ρ by addressing
    [Jungk et al, ReConFig 2011]
- Suitable for compact FPGA or ASIC
- Performance-area trade-offs
  - Possible to select number of processed slices from 1 up to 32
    [VHDL on http://keccak.noekeon.org/]



Implementations | Hardware implementations

Slice-wise hardware architecture

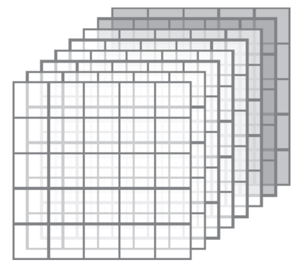
- Re-schedule the execution
  - χ, θ, π and ι on blocks of slices
  - ρ by addressing
    [Jungk et al, ReConFig 2011]
- Suitable for compact FPGA or ASIC
- Performance-area trade-offs
  - Possible to select number of processed slices from 1 up to 32
    [VHDL on http://keccak.noekeon.org/]



91 / 146

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/]

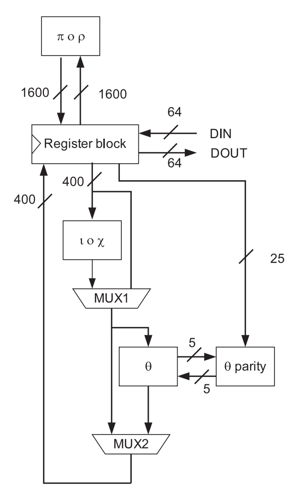


Mid-range core

- Example: state divided in 4 blocks
  - compute or absorb/squeeze
  - $\pi\rho$ done in one shot as wiring

Architecture	Size	Throughput (at 500MHz)
High-speed core	48.0 KGE	27.9 Gbit/s
Mid-range $N_b = 2$	28.3 KGE	7.4 Gbit/s
Mid-range $N_b = 4$	22.3 KGE	4.7 Gbit/s

Techno: STM 130nm



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]	68	265	448	5160	0.12
Lane-wise [2]	657	520	151 (+3)	1062	3.32
Slice-wise [3]	1067	159	372	200	2.19
High-Speed [4]	16786	305	1384	24	9.24

[1] Keccak Team, KECCAK implementation overview
[2] San, At, ISJ 2012
[3] Jungk, Apfelbeck, ReConFig 2011
[4] GMU ATHENa

All scaled to $r = 1344$

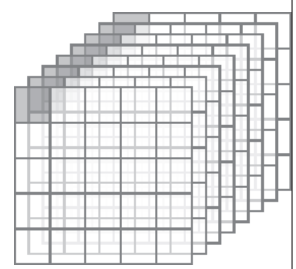
Further low area

Peter Pessl and Michael Hutter ”Pushing the Limits of SHA-3 Hardware Implementations to Fit on RFID” CHES2013

- Idea: store the state in a RAM, organization of data as a mix of bit interleaving and slice oriented
- latency, depends on the size of the RAM:
  - 16 bit word: 15k clock cycles
  - 8 bit word: 22k clock cycles
- Area goes down to 5.5 to 5.9kgate including RAM

Lanes: straightforward software implementation

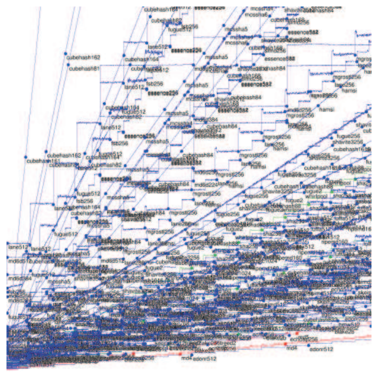
- Lanes fit in 2^ℓ-bit registers
  - 64-bit lanes for KECCAK-$f[1600]$
  - 8-bit lanes for KECCAK-$f[200]$
- Very basic operations required:
  - θ XOR and 1-bit rotations
  - ρ rotations
  - π just reading the correct words
  - χ XOR, AND, NOT
  - ι just a XOR



Optimizations

- The lane complementing transform
  - χ requires 5 XORs, 5 AND and 5 NOT
  - The number of NOT can be reduced to 1, storing lanes in complemented form
- Redundant state representation: includes the column parities
- Plane-per-plane processing
- Evaluate two or more permutations in parallel on a single core via SIMD
- Most of the code is generated by KeccakTools

Some benchmarks



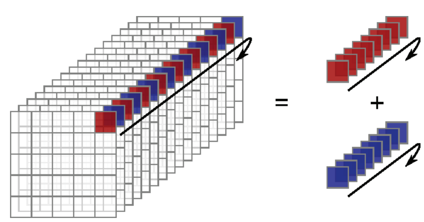
- Competitive with SHA-2 on all modern PC
- KECCAKTREE faster than MD5 on some platforms

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
  - $\mathrm{ROT}_{64} \leftrightarrow 2 \times \mathrm{ROT}_{32}$
- 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 $c = 256$

Extending the scope of software implementations?

In **KeccakReferenceAndOptimized.zip**, there are

- implementations for hashing only
- implementations of Keccak-f[1600] only

So what about extending this set to

- other applications
- parallelized modes
- Ketje and Keyak
- Keccak-f[800/400/200], Keccak-p[1600, $n_r = 12$], etc.
  - ... and other permutations ... ?

Extending the scope of software implementations?

In KeccakReferenceAndOptimized.zip, there are

- implementations for hashing only
- implementations of KECCAK-$f[1600]$ only

So what about extending this set to

- other applications
- parallelized modes
- KETJE and KEYAK
- KECCAK-$f[800/400/200]$, KECCAK-$p[1600, n_r = 12]$, etc.
  - … and other permutations … ?

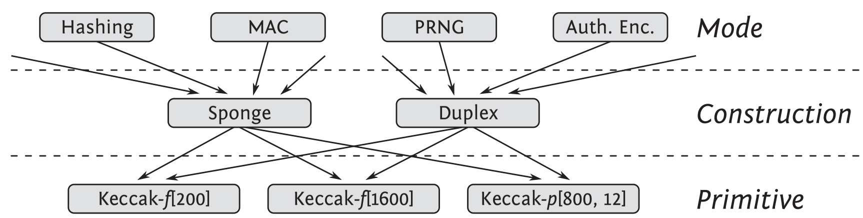
A heterogenous set of software implementations

In KeccakReferenceAndOptimized.zip, there are

- implementations for various architectures
- with **different** structures
- with hard-coded or flexible capacity
- with or without an input queue

avr8, avr8asm-compact, avr8asm-fast, compact, compact8, inplace, inplace32BI-armgcc-ARMv6M/v7A/v7M, opt32, opt64, reference, reference32BI, xop, simple, simple32BI, simd64, simd128, x86-64, x86-64-shld, Keccakc512-crypto_hash-inplace-armgcc-ARMv7A-NEON.s, …

Goals of a layered approach

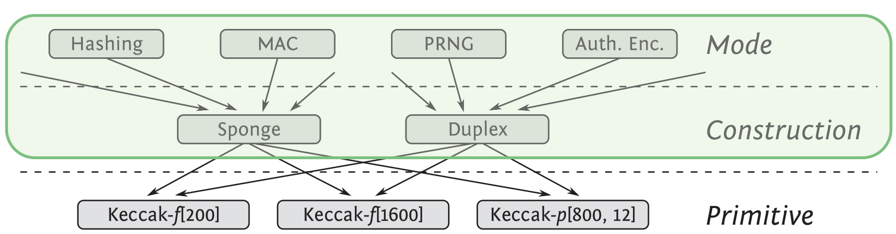


Generic
- focus on **user**
  - as easy to use as possible
  - e.g., message queue, etc.
- <u>one</u> implementation
  - pointers and arithmetic

Specific
- focus on **developer**
  - limited scope to optimize
  - bugs caught early
- tailored implementation<u>s</u>
  - permutation
  - bulk data processing

Implementations | Inside the package

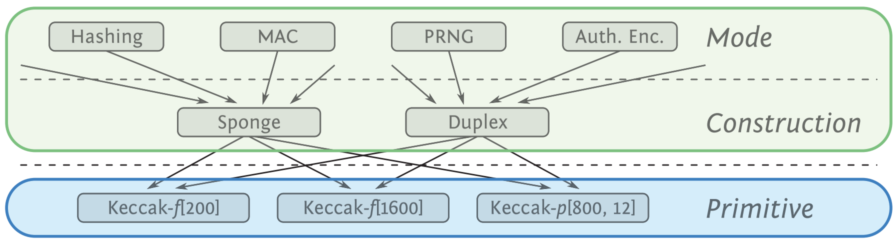
Goals of a layered approach



Generic
- focus on **user**
  - as easy to use as possible
  - e.g., message queue, etc.
- <u>one</u> implementation
  - pointers and arithmetic

Specific
- focus on **developer**
  - limited scope to optimize
  - bugs caught early
- tailored implementation<u>s</u>
  - permutation
  - bulk data processing

Goals of a layered approach



Generic

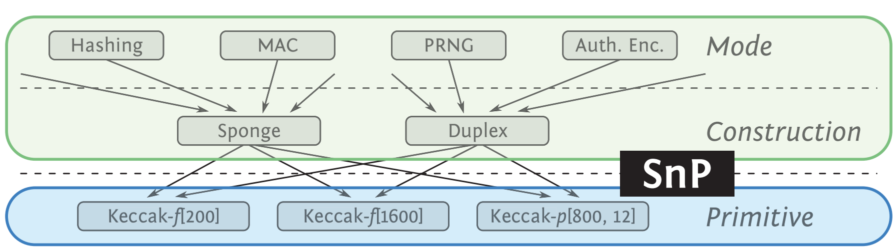
- focus on **user**
  - as easy to use as possible
  - e.g., message queue, etc.
- one implementation
  - pointers and arithmetic

Specific

- focus on **developer**
  - limited scope to optimize
  - bugs caught early
- tailored implementations
  - permutation
  - bulk data processing

Implementations | Inside the package

Goals of a layered approach



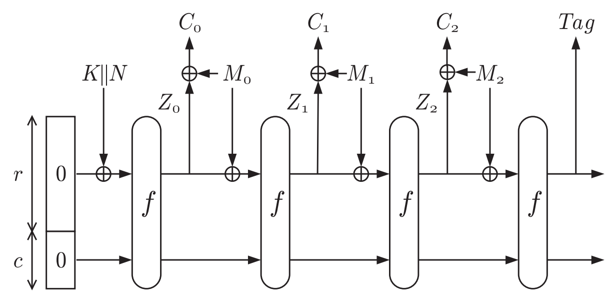
Generic

- focus on **user**
  - as easy to use as possible
  - e.g., message queue, etc.
- one implementation
  - pointers and arithmetic

Specific

- focus on **developer**
  - limited scope to optimize
  - bugs caught early
- tailored implementations
  - permutation
  - bulk data processing

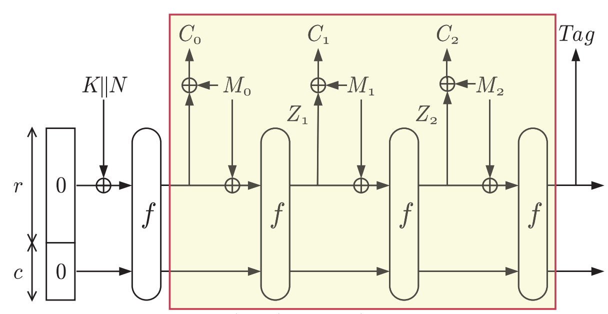
SnP (= State and Permutation)



Absorbing and squeezing

- initialize the state to zero
- apply the permutation f
- XOR/overwrite bytes into the state
- extract bytes from the state
  - and optionally XOR them

SnP FBWL (= Full Blocks Whole Lane)

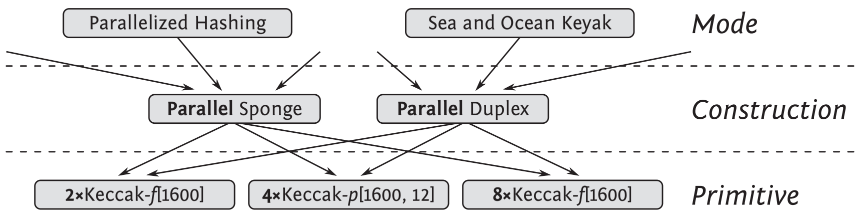


Absorbing and squeezing

Specialized repeated application of some operations
(optional)

SnP_FBWL_Absorb/Squeeze/Wrap/Unwrap

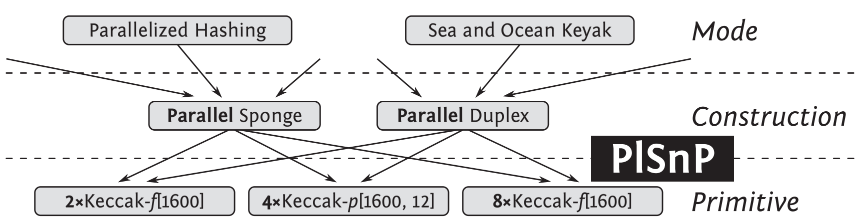
Parallel processing



- Some modes exploit parallelism
- To exploit this, we need:
  - sponge functions and duplex objects running in parallel
  - permutation applied on several states in parallel

Implementations | Inside the package

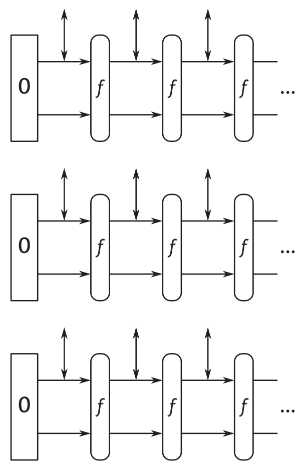
Parallel processing



- Some modes exploit parallelism
- To exploit this, we need:
  - sponge functions and duplex objects running in parallel
  - permutation applied on several states in parallel

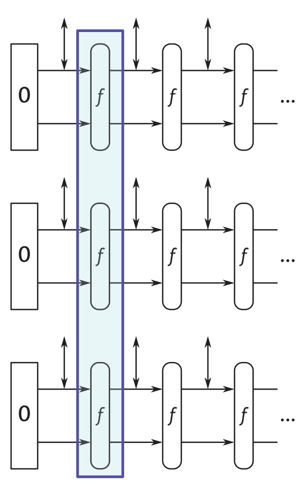
105 / 146

PlSnP (= Parallel States and Permutations)



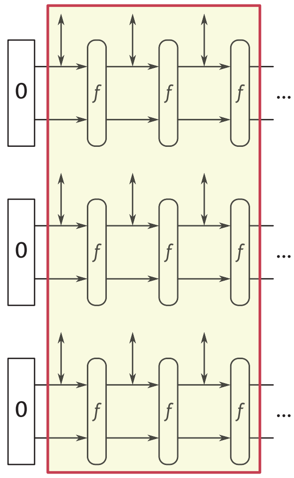
- SnP on individual instances
- Some SnP functions parallelized
  - Parallel application of f
- **PlSnP FBWL** for repeated operations

PlSnP (= Parallel States and Permutations)



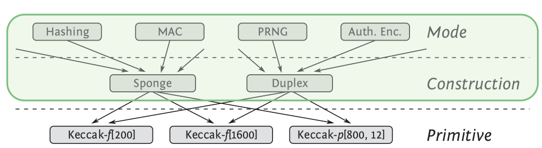
- SnP on individual instances
- Some SnP functions parallelized
  - Parallel application of f
- **PlSnP FBWL** for repeated operations

PlSnP (= Parallel States and Permutations)



- SnP on individual instances
- Some SnP functions parallelized
  - Parallel application of f
- **PlSnP FBWL** for repeated operations

Constructions and modes



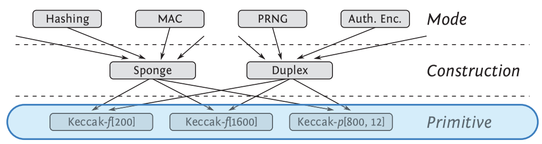
Currently in the KCP

- SHA-3 hashing and XOFs
- River and Lake Keyak
- Ketje
- Anything using sponge or duplex directly

Nice to have

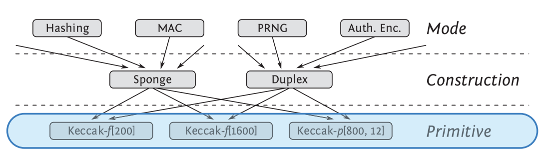
- Pseudo-random bit sequence generator
- Overwrite sponge

Primitives



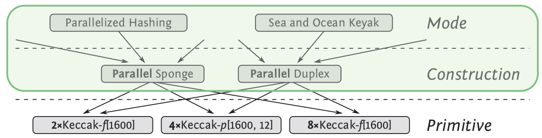
KECCAK-f[200 to 1600], KECCAK-p[200 to 1600, n_r]
- Reference implementations
- Optimized impl. in C of KECCAK-f[1600] and -p[1600, $n_r = 12$]
  - using 64-bit words or 32-bit words (bit interleaving)
  - compact, in place, unrolled, lane complemented, etc.
- Assembly optimized for
  - x86_64 (KECCAK-f[1600] and KECCAK-p[1600, $n_r = 12$] only)
  - ARMv6M, ARMv7M, ARMv7A, NEON
  - AVR8

Primitives



On the to-do list
- Some implementations still to be migrated from KeccakReferenceAndOptimized.zip
- Optimized in C for 800-bit width and smaller
- ARMv8, (your favorite platform here)

Parallel constructions and modes



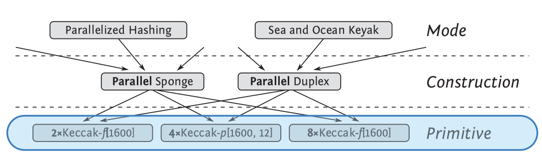
Currently in the KCP

- Sea and Ocean Keyak
- Anything using parallel duplex objects directly

On the to-do list

- Parallel sponge functions
- Parallelized hashing

Parallelized primitives



Currently in the KCP

- Serial fallback to SnP
- $2 \times$ Keccak-$f[1600] / p[1600, n_r = 12]$ on ARMv7M+NEON

Many things on the to-do list

- $2 \times$ Keccak-$f[1600] / p[1600, n_r = 12]$ using SSE, XOP or AVX (...WIP...)
- $4 \times$ Keccak-$f[1600] / p[1600, n_r = 12]$ using AVX2 or AVX512
- $8 \times$ Keccak-$f[1600] / p[1600, n_r = 12]$ using AVX512
- ARMv8 NEON, (your favorite SIMD instruction set here)

Recap

- In this section we have seen:
  - Hardware and software implementations techniques
  - The KeccakCodePackage evolution
    - on github https://github.com/gvanas/KeccakCodePackage

Outline

Secure implementations

Keyed modes may require protected implementations

- Keccak offers protection against
  - timing or cache-miss attacks
    no table look-ups
  - side channels (DPA)
    efficient secret sharing thanks to degree-2 round function

A model of the power consumption

Consumption at any time instance can be modeled as

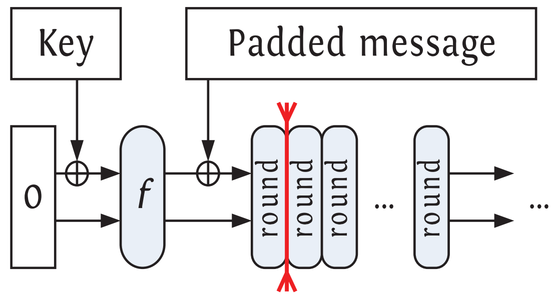
$$P = \sum_i T_i[d_i]$$

- 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



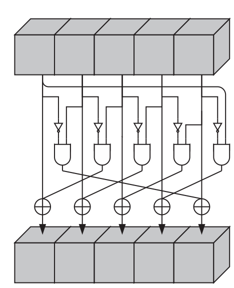
1. Attack the first round after absorbing known input bits
2. Compute backward by inverting the permutation

The KECCAK-f round function in a DPA perspective

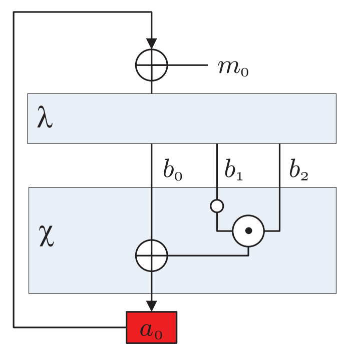
$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

- 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:

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

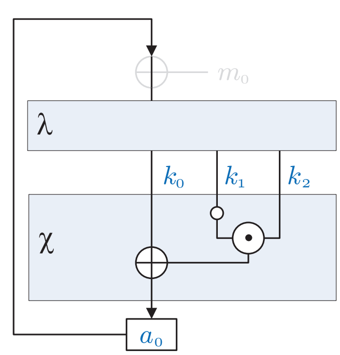


DPA applied to an unprotected implementation



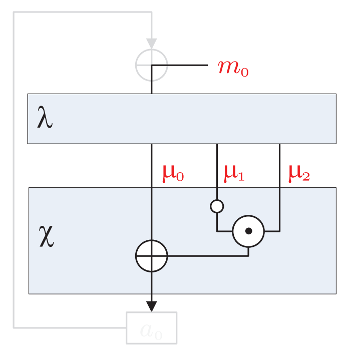
- Leakage exploited: switching consumption of register bit 0
- Value switches from a_0 to $b_0 + (b_1 + 1)b_2$
- Activity equation: $d = a_0 + b_0 + (b_1 + 1)b_2$

DPA applied to an unprotected implementation



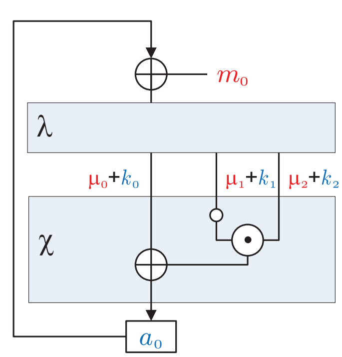
- Take the case $M = 0$
- We call K the input of χ-block if $M = 0$
- K will be our target

DPA applied to an unprotected implementation



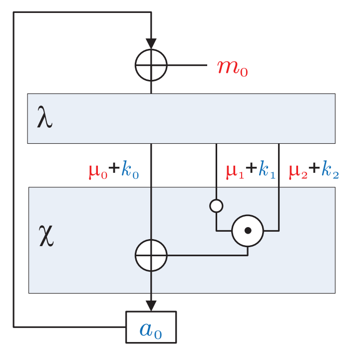
- We call the effect of M at input of χ: μ
- $\mu = \lambda(M||0^c)$
- Linearity of λ: $B = K + \lambda(M||0^c)$

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



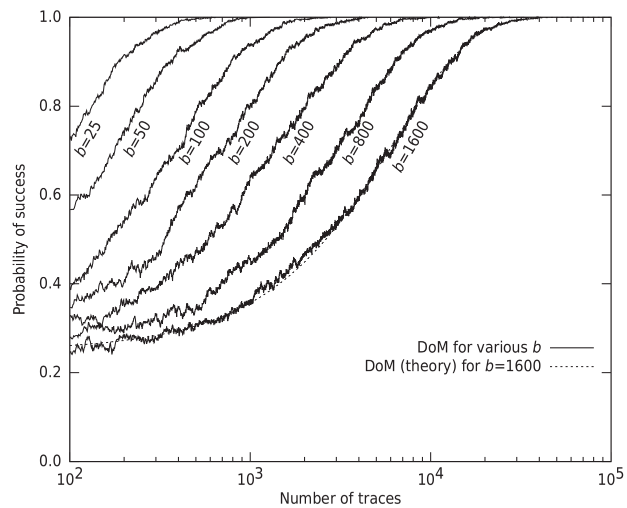
- Selection: $s(M, K^*) = \mu_0 + (\mu_1 + 1)\mu_2 + k_1^*\mu_2 + k_2^*\mu_1$
- Values of μ_1 and μ_2 computed from M
- Hypothesis has two bits only: k_1^* and k_2^*

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]



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
- Implementation cost depends on the algebraic degree:
  - Linear: compute shares independently
  - Non-linear: higher degree $\Rightarrow$ more expensive
- Keccak round function
  - Linear mapping $\lambda = \pi \circ \rho \circ \theta$ followed by nonlinear χ:

$$x_i \leftarrow x_i + (x_{i+1} + 1)x_{i+2}$$

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:

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

Attack on the fast SW implementation

- L. Bettal et al. published ”Collision-Correlation Attack against a First-Order Masking Scheme for MAC based on SHA-3” [COSADE2014]
- since one of the share is kept constant it is possible to observe collisions in the computation and thus having a first order leakage

Hardware: two shares are not enough

- Unknown order in combinatorial logic!

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

- Glitches might give a first order leakage

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 χ
- Glitches become as second-order effect

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 χ
- Glitches become as second-order effect

Keccak and Side Channel

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 χ
- Glitches become as second-order effect

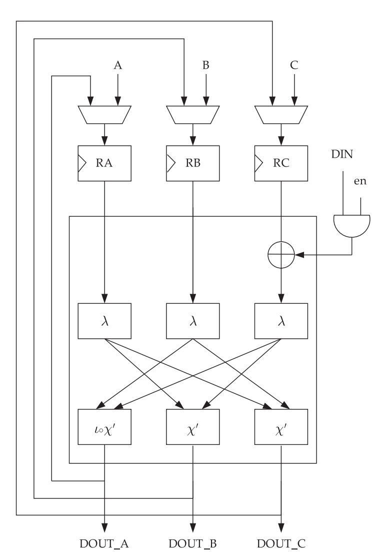
125 / 146

Three-share masking for χ

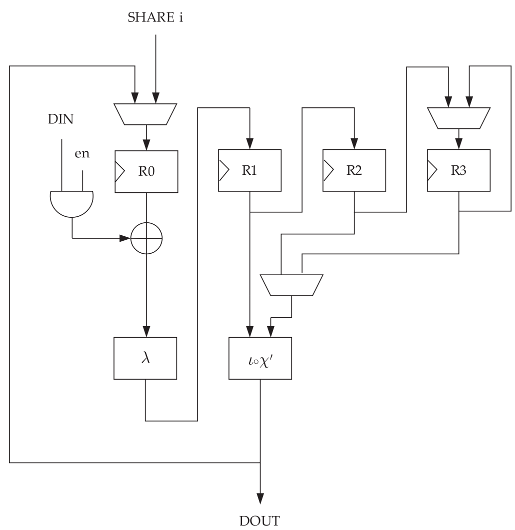
- Implementing χ in three shares:

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

One-cycle round architecture

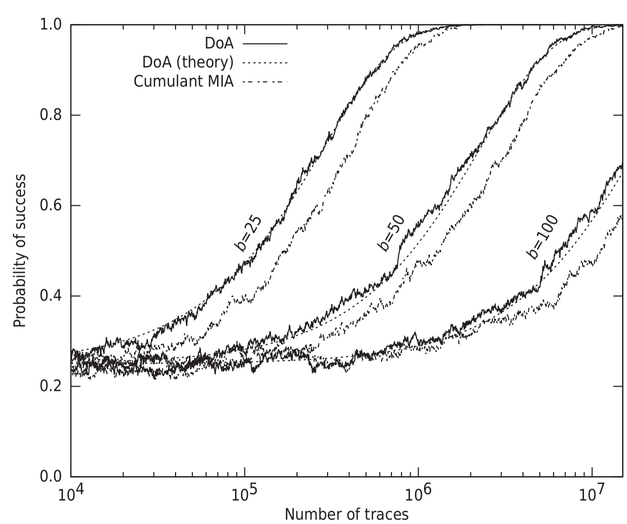


Three-cycle round architecture



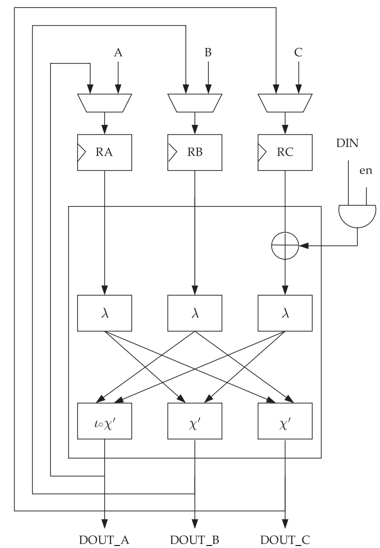
Evaluation of the countermeasure

- Generalization of results for protected implementation
  [Bertoni, Daemen, Debande, Le, Peeters, Van Assche, HASP 2012]



How to use the 3-share KECCAK architectures

- Randomness initialization:
  - fill registers with random a and b
  - Fill register with $c = a + b$
- Input absorbed in single share



A possible problem in our formulation?

Sharing for χ (cyclically on 5-bit rows) [BDPV SHA-3 2010]:

$$a'_i \leftarrow \chi'_i(b,c) \triangleq b_i + (b_{i+1}+1)b_{i+2} + b_{i+1}c_{i+2} + b_{i+2}c_{i+1},$$
$$b'_i \leftarrow \chi'_i(c,a) \triangleq c_i + (c_{i+1}+1)c_{i+2} + c_{i+1}a_{i+2} + c_{i+2}a_{i+1},$$
$$c'_i \leftarrow \chi'_i(a,b) \triangleq a_i + (a_{i+1}+1)a_{i+2} + a_{i+1}b_{i+2} + a_{i+2}b_{i+1},$$

This sharing is not uniform!

- Two possible problems:
  - Long-term: randomness evaporates until finally none is left
  - Short-term: input to *next* round is not uniform
- Two approaches:
  - Tweak architecture to restore uniformity [BDNNRV Cardis '13]
  - Study non-uniformity to see how bad it is
    - work in progress presented at Daghstul 2014: not a problem

Leakage Resiliance fashion

- M. Taha and P. Schaumont suggest to add an IV to the key and absorb one bit of IV every 3 rounds
  - pro: almost zero overhead
  - con: requiring a longer initialization phase particularly per MAC where IV is not requested
- see: "Side-Channel Countermeasure for SHA-3 At Almost-Zero Area Overhead" HOST 2014

Recap

- In this section we have seen how to:
  - attack an unprotected implementation
  - Countermeasure for HW and SW implementaitons
    - with known limitations and recent results

Outline

Standard time line

- November 2007: formal announcement
- October 2012: Selection of Keccak
- Year 2013: Dissemination: proposal of 2 capacities, withdrawn
- March 2014: Publication first draft of FIPS202, May minor changes
- Now: Preparation of Special Publication

Output length oriented approach

Output length	Collision resistance	Pre-image resistance	Required capacity	Relative perf.	SHA-3 instance
$n = 224$	$s \leq 112$	$s \leq 224$	$c = 448$	$\times 1.125$	**SHA3n224**
$n = 256$	$s \leq 128$	$s \leq 256$	$c = 512$	$\times 1.063$	**SHA3n256**
$n = 384$	$s \leq 192$	$s \leq 384$	$c = 768$	$\div 1.231$	**SHA3n384**
$n = 512$	$s \leq 256$	$s \leq 512$	$c = 1024$	$\div 1.778$	**SHA3n512**
n	$s \leq n/2$	$s \leq n$	$c = 2n$	$\times \frac{1600-c}{1024}$	

s: security strength level [NIST SP 800-57]
n:output length

- These instances address the SHA-3 requirements, but:
  - multiple security strengths each
  - levels outside of [NIST SP 800-57] range
- Performance penalty!

Security strength oriented approach

Security strength	Collision resistance	Pre-image resistance	Required capacity	Relative perf.	SHA-3 instance
$s = 112$	$n \geq 224$	$n \geq 112$	$c = 224$	$\times 1.343$	**SHA3c224**
$s = 128$	$n \geq 256$	$n \geq 128$	$c = 256$	$\times 1.312$	**SHA3c256**
$s = 192$	$n \geq 384$	$n \geq 192$	$c = 384$	$\times 1.188$	**SHA3c384**
$s = 256$	$n \geq 512$	$n \geq 256$	$c = 512$	$\times 1.063$	**SHA3c512**
s	$n \geq 2s$	$n \geq s$	$c = 2s$	$\times \frac{1600-c}{1024}$	**SHA3[c=2s]**

s: security strength level [NIST SP 800-57]
n:output length

- These SHA-3 instances
  - are consistent with philosophy of [NIST SP 800-57]
  - provide a one-to-one mapping to security strength levels
- Higher efficiency

FIPS 202: SHA-3 (draft out since April 4, 2014)

XOF	**SHA-2 drop-in replacements**
$\text{KECCAK}[c = 256](M\|\|11\|\|11)$	
	$\lfloor \text{KECCAK}[c = 448](M\|\|01) \rfloor_{224}$
$\text{KECCAK}[c = 512](M\|\|11\|\|11)$	$\lfloor \text{KECCAK}[c = 512](M\|\|01) \rfloor_{256}$
	$\lfloor \text{KECCAK}[c = 768](M\|\|01) \rfloor_{384}$
	$\lfloor \text{KECCAK}[c = 1024](M\|\|01) \rfloor_{512}$
SHAKE128 and **SHAKE256** with SAKURA coding	**SHA3-224** to **SHA3-512**

$$\text{SHAKE}(M) = \text{KECCAK}(M||\text{“message hop”}||\text{“final node”}||11)$$

Note: FIPS 202 contain the definition of KECCAK-$p[b, n_r]$

KECCAK towards the SHA-3 standard

Paddings

Three Types of Padding Bits

- Multi-rate padding
  - 10*1
  - for all KECCAK
- Domain Separation
  - 11 for RawSHAKE function
  - 01 for SHA-3 hash function
- Sakura coding for parallel hashing
  - 11 sequential RawSHAKE = SHAKE

eXtendable-Output Functions

What is a XOF (pronounced ”Zoff”)?

”A function on bit strings in which the output can be extended to any desired length.”

- Two input parameters: Message, Output Length
  - If XOF(M, 128) = AB,
  - then XOF(M, 256) = ABCD

Good for full domain hash, stream ciphers and KDF (XKDF)

[Ray Perlner, SHA 3 workshop 2014]

MACs

- predecessor: HMAC
- new
  - kmak: MAC(text) = KMAC(K, text) = H(Keypack(K, l) || text)
  - xmac: MAC(text) = XMAC(K, text, λ) = XOF(Keypack(K, l) || text, λ)
    - λ length of the output
- HMAC based on SHA-3

Others

- Authenticated Encryption
- XKDF: key derivation function based on XOF (XMAC)
- tree hashing
  - SHAKEs are ready, but would like to have a general standard to be used with SHA256 as well....
- See presentations of the SHA-3 workshop

Adoption by ETSI

- Recently SAGE (Security Algorithms Group of Experts) of ETSI (European Telecom Standard Institute) has defined TUAK
- It is based on KECCAK with capacity of 768 bits
- More detials in ETSI SP-130602

Recap

- In this section we have seen:
  - The time line of the SHA-3 standard
  - The concept behind the selection of parameters
  - The possible content of the upcoming special publication

Book

We are writing a book...

Conclusions

Thanks for your attention!
keccak@noekeon.org

Q?