Keccak and SHA-3: code and standard updates

Guido Bertoni[1] Joan Daemen[1] Michaël Peeters[2]
Gilles Van Assche[1] Ronny Van Keer[1]

[1]STMicroelectronics

[2]NXP Semiconductors

FOSDEM 2015, Brussels, January 31st & February 1st, 2015

Outline

Outline

What is a hash function?

What is a hash function?

```
#!/bin/ash

notmagritte()
{
  echo "this is a ash function!"
}
```

What is a hash function?

```
#!/bin/ash

notmagritte()
{
  echo "this is a ash function!"
}
```

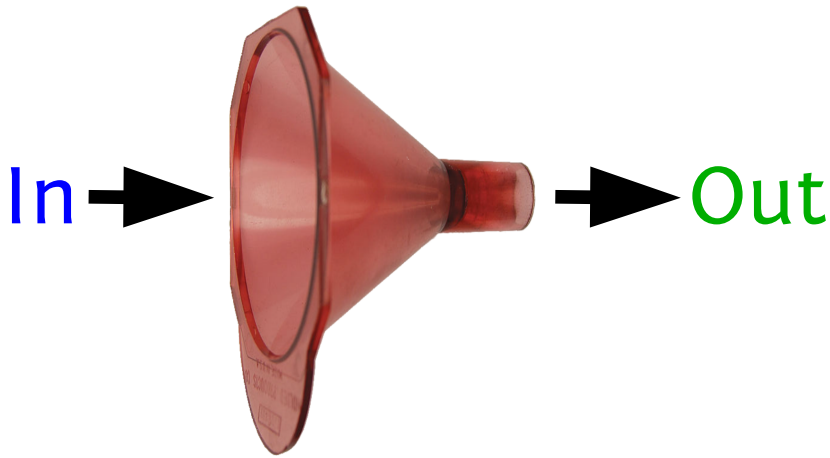
This is *not* a **hash** function!

What is a hash function?

$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$

```
#!/bin/ash

notmagritte()
{
  echo "this is a ash function!"
}
```

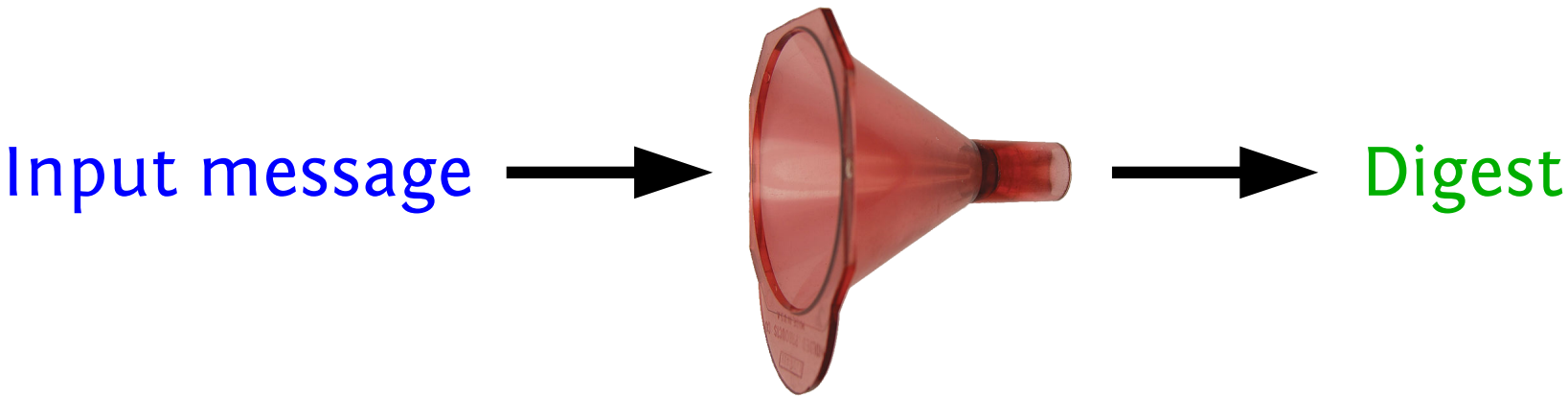
This is *not* a **hash** function!



This *is* a **hash** function!

Cryptographic hash functions

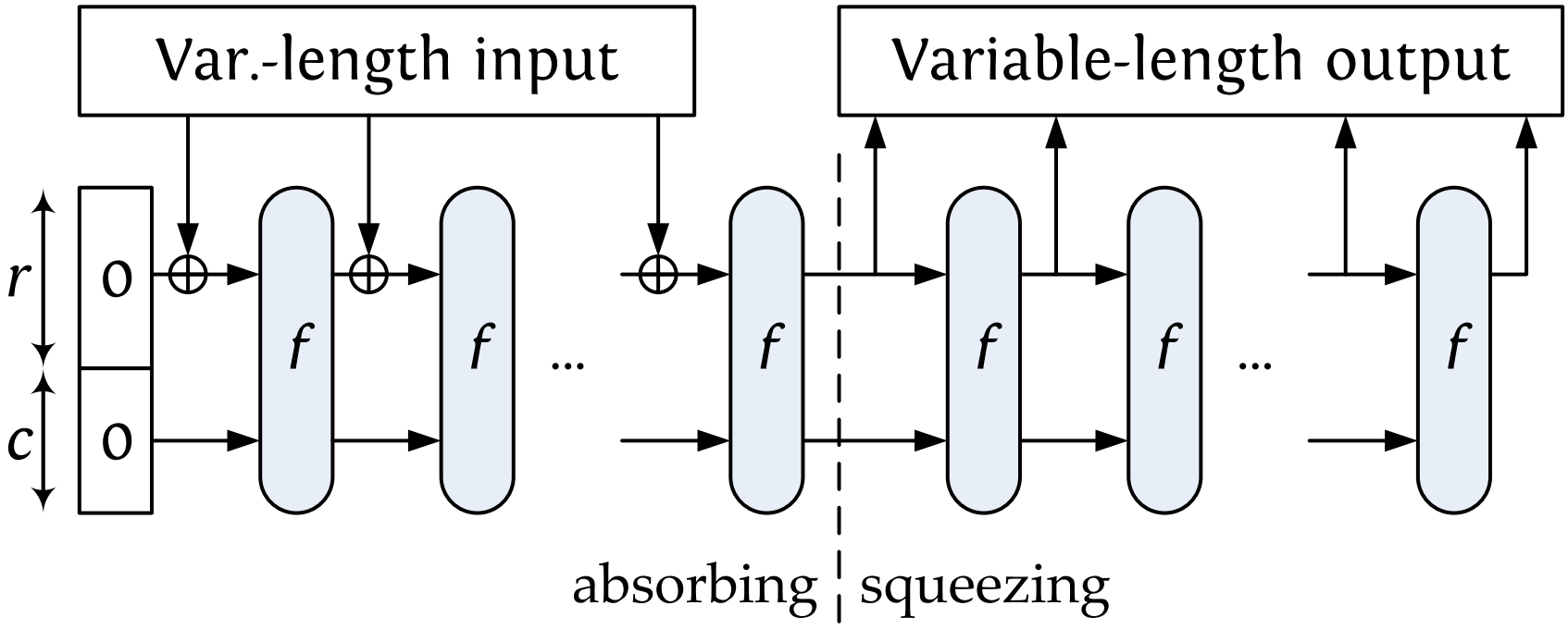
$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$



- MD5: $n = 128$ (Ron Rivest, 1992)
- SHA-1/2: $n \in \{160, 224, 256, 384, 512\}$ (NSA, NIST, 1995-2001)

...and Keccak? It is a (cryptographic) *sponge* function!

Cryptographic sponge functions



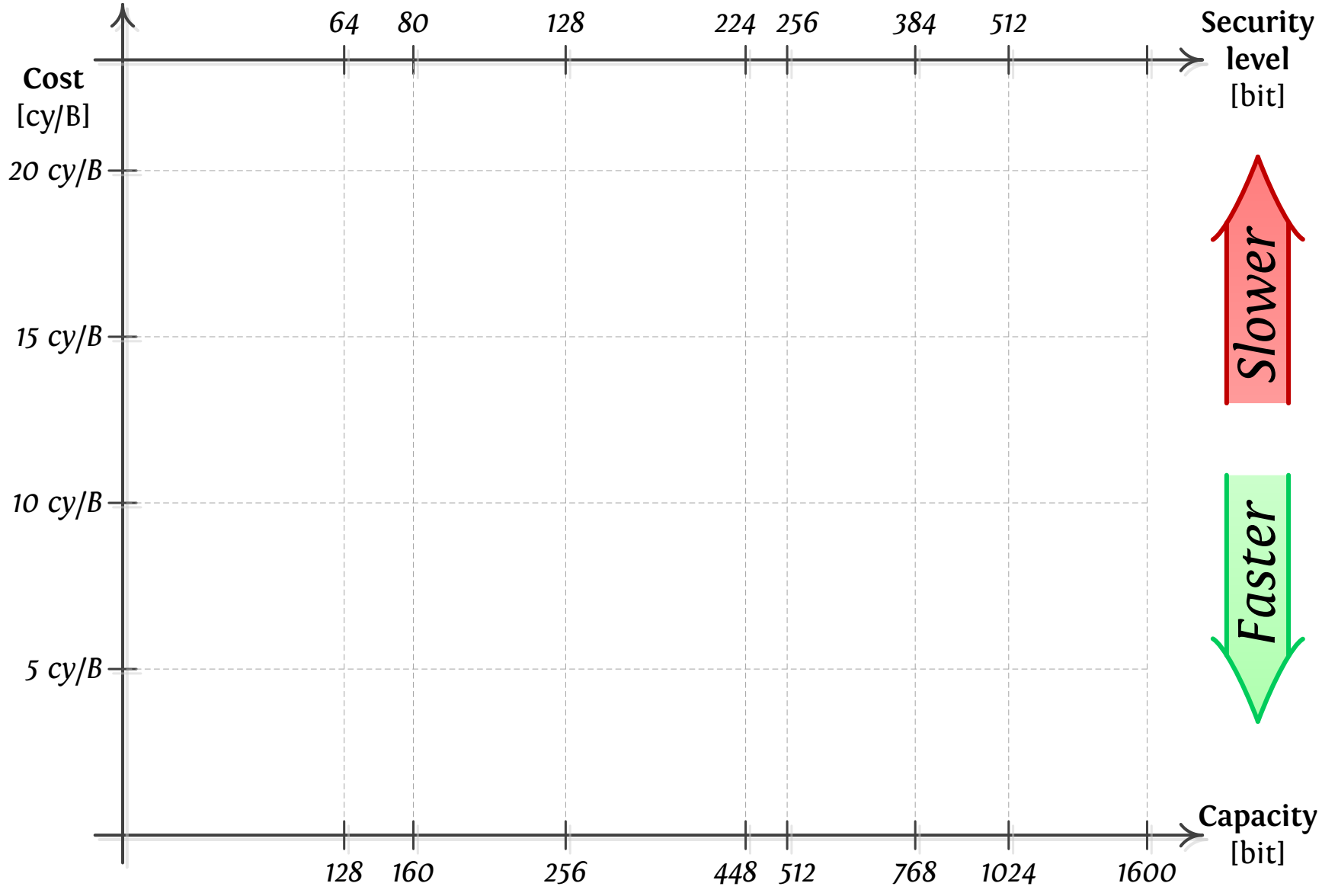
- Arbitrary *input* and *output* length
- More flexible than regular hash functions
- Parameters
  - r bits of *rate* (defines the speed)
  - c bits of *capacity* (defines the security level)
- Keccak uses the permutation Keccak-f

Keccak-f in pseudo-code

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

Round[b](A,RC) {
  θ step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4], forall x in 0…4
  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
}
```

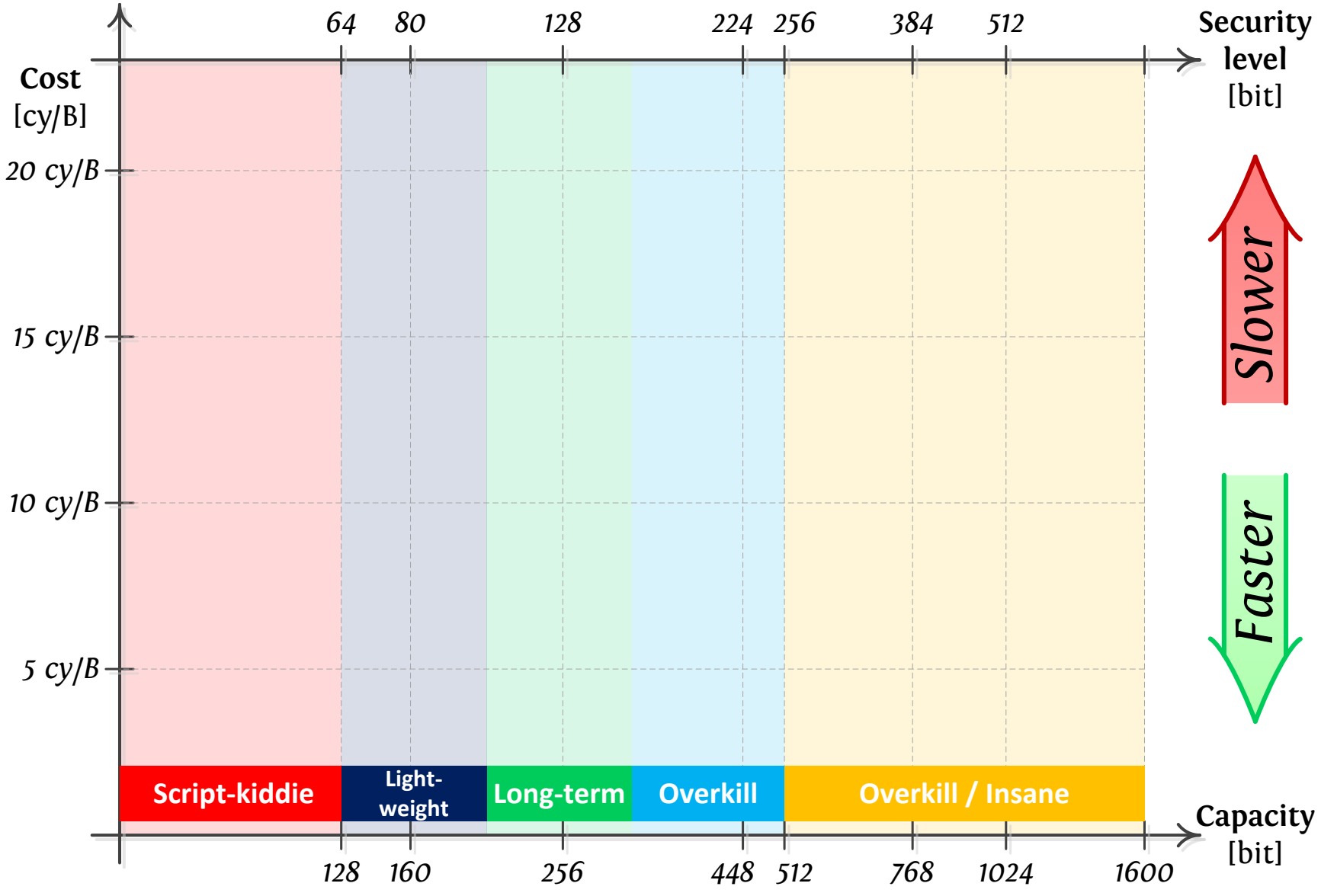
7 *widths* b ($= r + c$): 25, 50, 100, 200, 400, 800, and 1600 bits.

Sponge tuning: capacity ⇒ security level



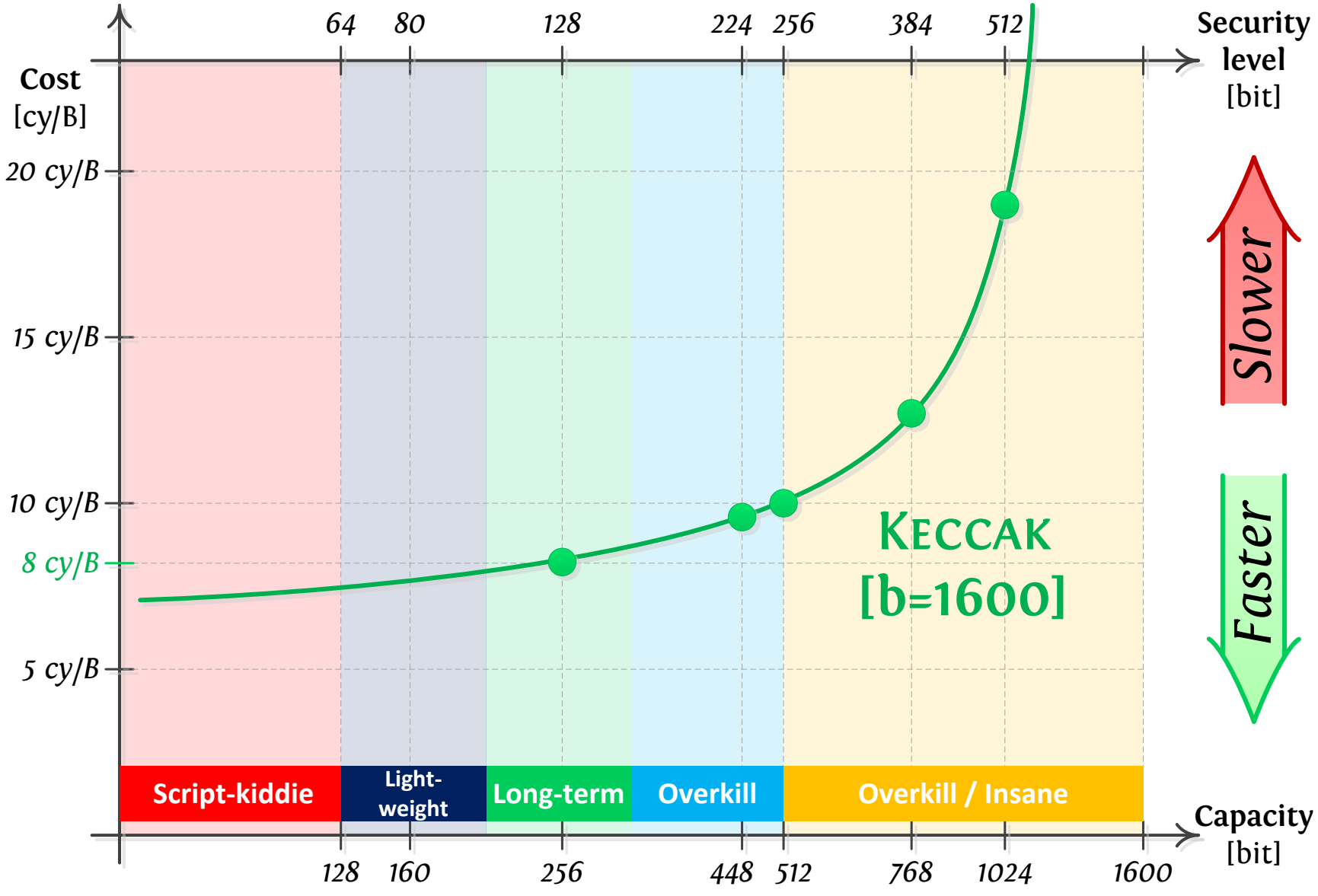
[eBASH, hydra6, http://bench.cr.yp.to/]

Sponge tuning: capacity ⇒ security level



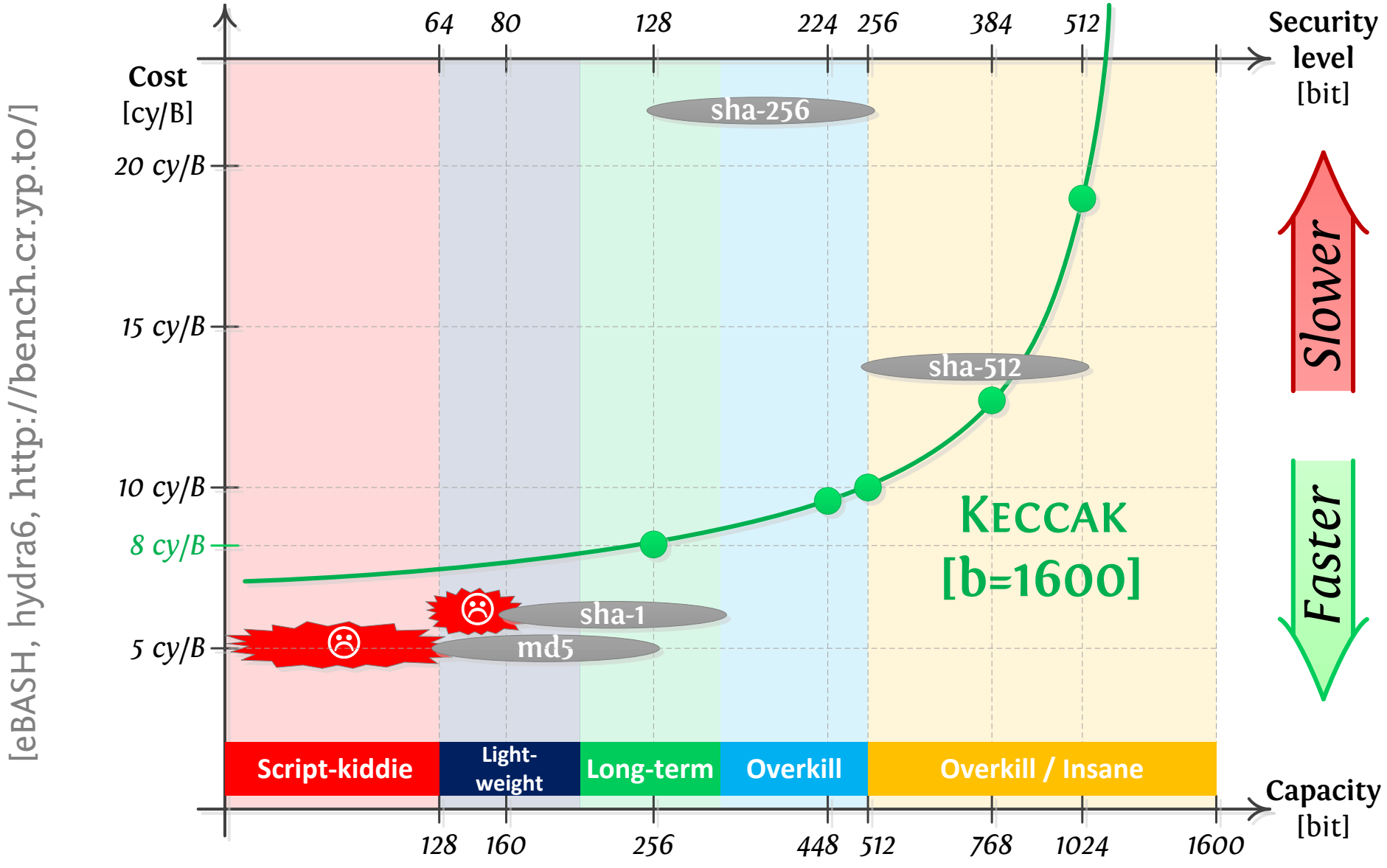
[eBASH, hydra6, http://bench.cr.yp.to/]

Sponge tuning: capacity ⇒ security level

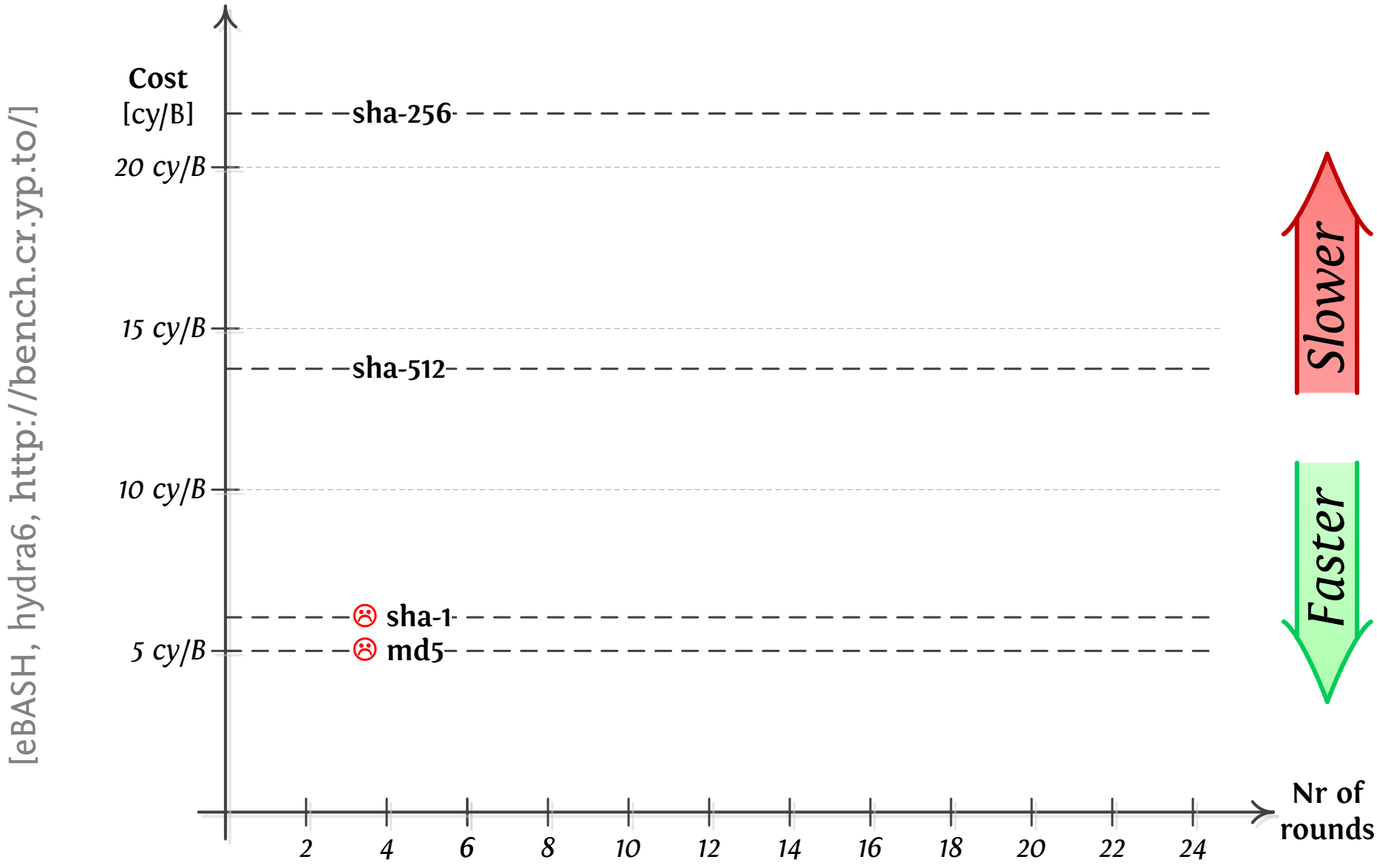


[eBASH, hydra6, http://bench.cr.yp.to/]

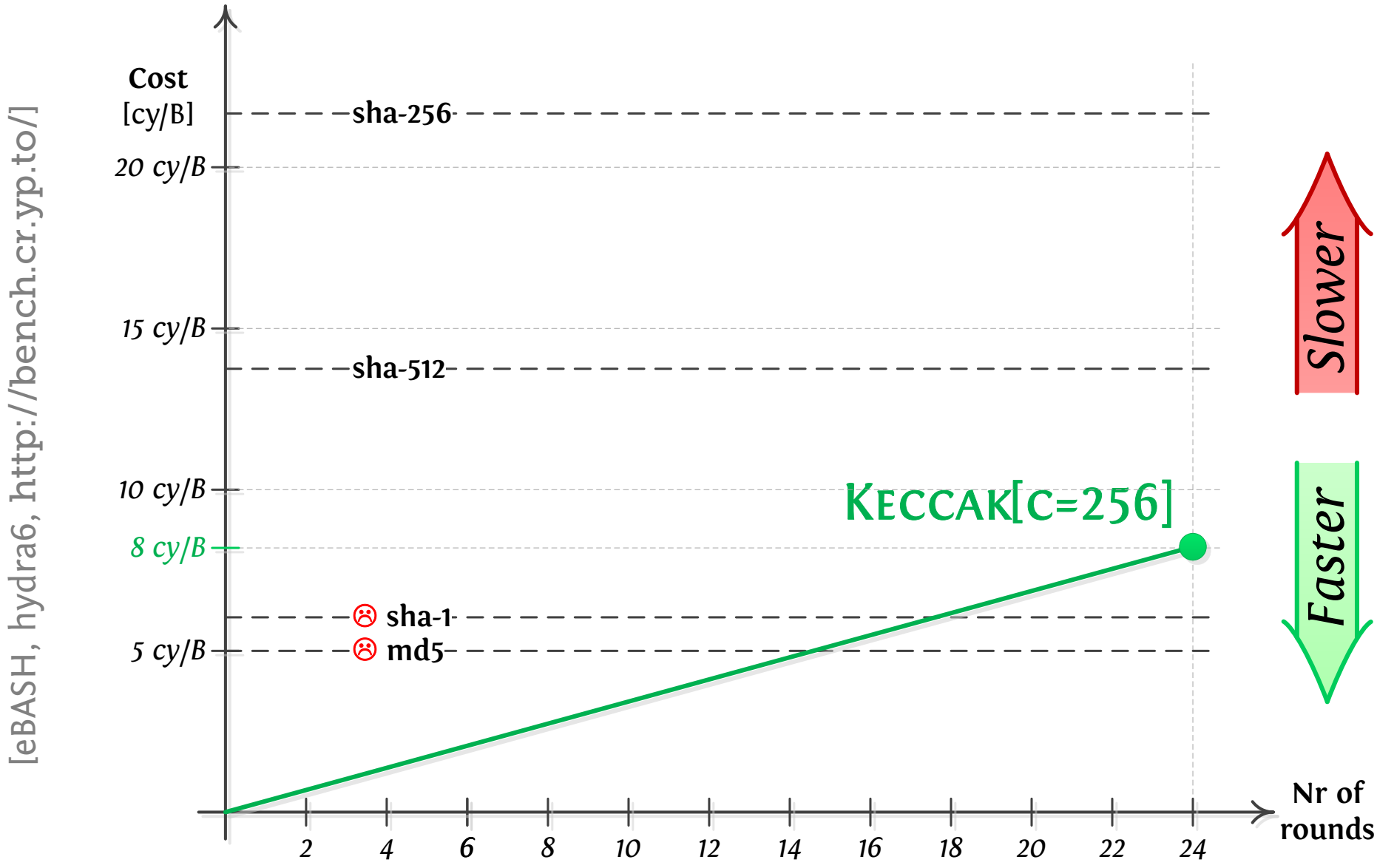
Sponge tuning: capacity ⇒ security level



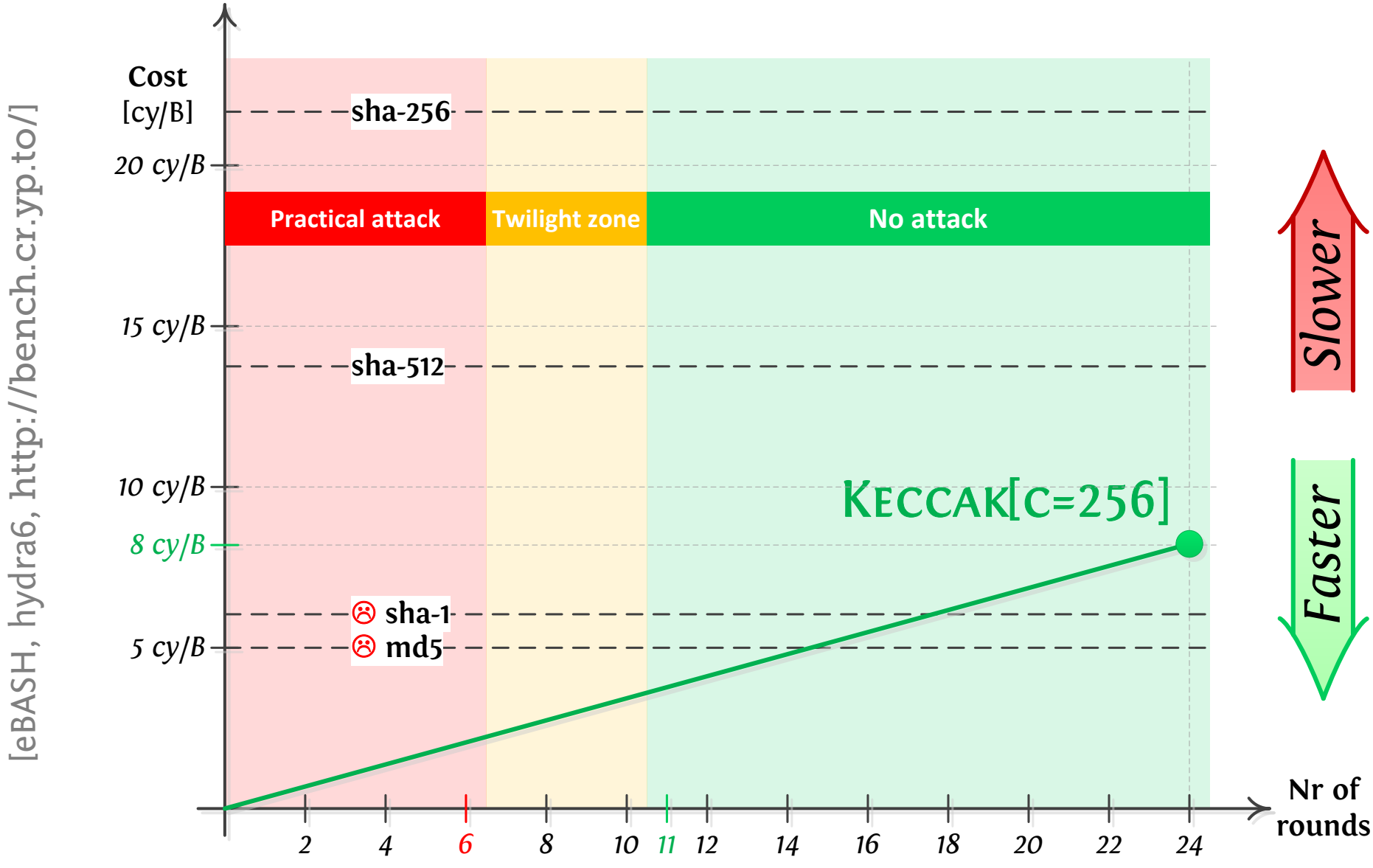
Keccak tuning: number of rounds ⇒ safety margin



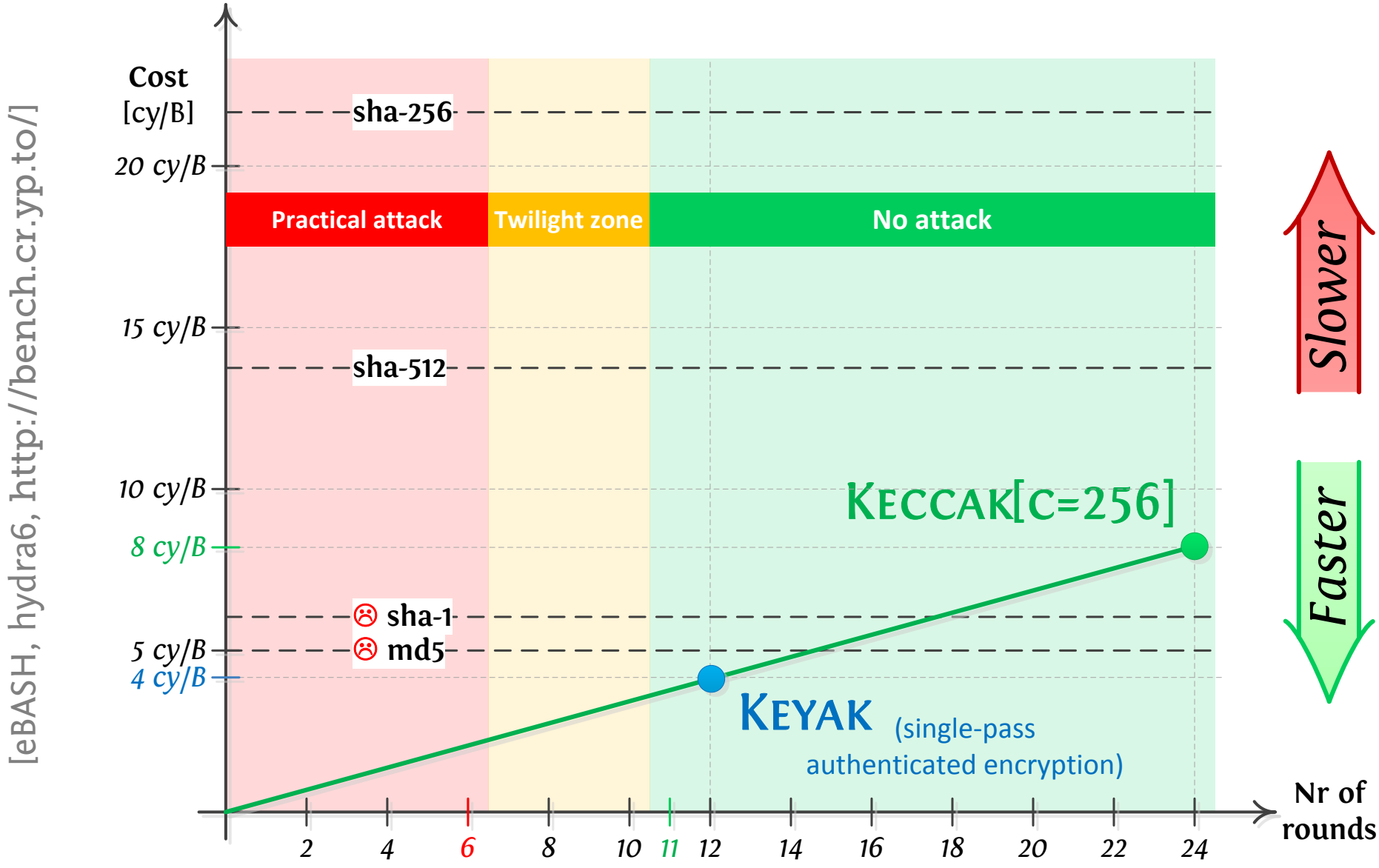
Keccak tuning: number of rounds ⇒ safety margin



Keccak tuning: number of rounds ⇒ safety margin



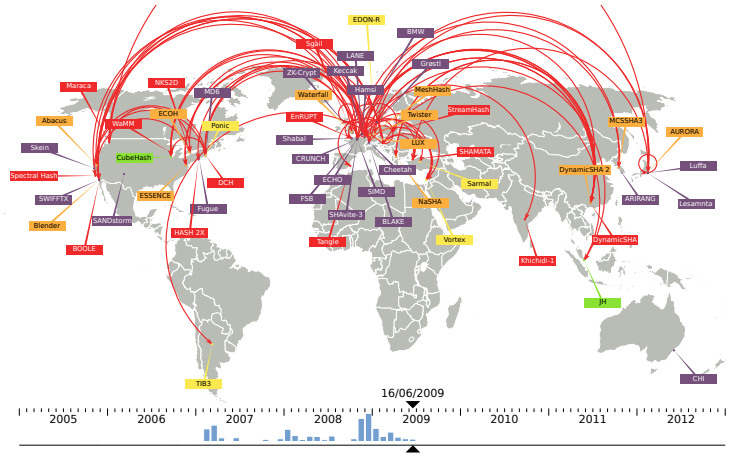
Keccak tuning: number of rounds ⇒ safety margin



Outline

The SHA-3 contest

- 2000-2006: crisis for standard hash function standards
  - MD5: practically broken
  - SHA-1: theoretically broken
  - SHA-2: serious doubts on foundations
- November 2007: NIST announces SHA-3 contest
  - goal: FIPS standard
  - scope: stand-ins for all 4 SHA-2
  - method: public competition like AES
  - response: 64 submissions
- Summer 2008: start with 51 proposals
- October 2012: Keccak = SHA-3



[courtesy of C. De Cannière]

The long road to the SHA-3 FIPS

By Piet Musterd (flickr.com)

- February 2013: NIST-Keccak-team meeting
  - SHA-2 replacement by now less urgent
  - …but Keccak is more than just hashing!
- NIST disseminates joint SHA-3 proposal
- Summer 2013: Snowden revelations
  - alleged NSA back door in DUAL EC DRBG
  - SHA-3 proposal framed as “NIST weakening Keccak”
- Early 2014: standard takes shape addressing public concerns
- Friday, April 4, 2014: draft FIPS 202 for public comments
- August 2014: NIST announces plans at SHA-3 conference
- Mid 2015 (expected): FIPS 202 official

FIPS 202: what is inside?

By Nicole Doherty (flickr.com)

- Content
  - KECCAK instances for
    - 4 hash functions
    - 2 XOFs
  - KECCAK-*f* all 7 block widths
    - even reduced-round versions
    - unlike AES FIPS that has only 1 of the 5 Rijndael widths
  - sponge construction
- Concept: toolbox for building other functions
  - tree hashing, MAC, encryption, ...
  - dedicated *special publications* (NIST SP 800-XX) under development

http://csrc.nist.gov/groups/ST/hash/sha-3/Aug2014/index.html

XOF: eXtendable Output Function

"XOF: a function in which the output can be extended to any length."

- Good for full domain hash, stream ciphers and key derivation
  [Ray Perlner, SHA 3 workshop 2014]
- Quite natural for sponge
  - keeps state and delivers more output upon request
  - bits of output do not depend on the number of bits requested
- Allows simplification:
  - instead of separate hash functions per output length
  - a single XOF can cover all use cases:

$$\text{H-256}(M) = \lfloor \text{XOF}(M) \rfloor_{256}$$

Domain separation

By Adam Fagen (flickr.com)

- Some protocols and applications need
  - multiple hash functions or XOFs
  - that should be *independent*
- With a single XOF?
- Yes: using domain separation
  - output of $\mathrm{XOF}(M||0)$ and $\mathrm{XOF}(M||1)$ are independent
  - ...unless XOF has a cryptographic weakness
- Generalization to 2^n functions with D an n-bit *diversifier*

$$\mathrm{XOF}_D(M) = \mathrm{XOF}(M||D)$$

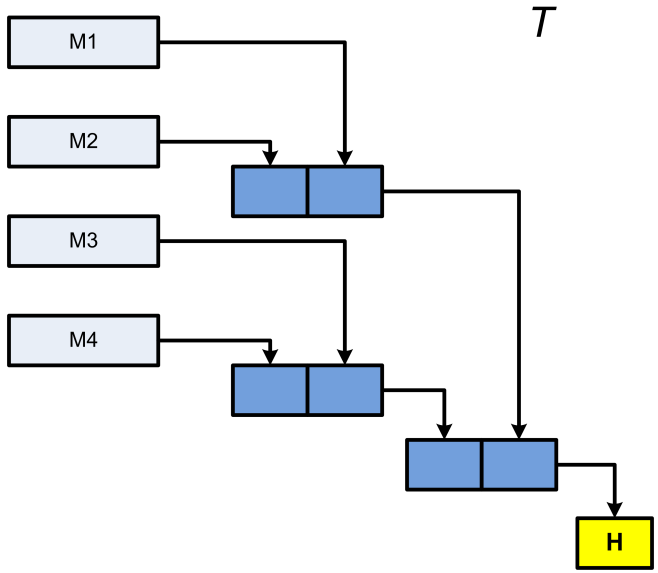
- Variable-length diversifiers: suffix-free set of strings

The XOFs and hash functions in FIPS 202

- Four drop-in replacements identical to those in KECCAK submission
- Two *extendable output functions* (XOF)
- Tree-hashing ready: SAKURA coding [Keccak team, ePrint 2013/231]

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

Tree hashing



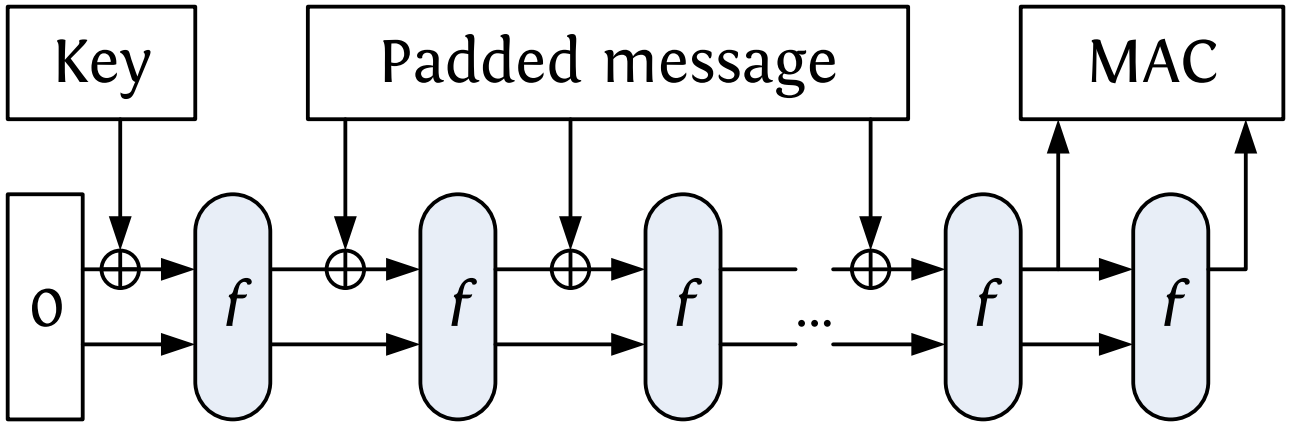
Features:

- hash recomputation when modifying small part of file
- peer-to-peer applications: Gnutella, BitTorrent etc.
- performance:

function	instruction	cycles/byte
KECCAK$[c = 256] \times 1$	x86_64	7.70
KECCAK$[c = 256] \times 2$	AVX2 (128-bit only)	5.30
KECCAK$[c = 256] \times 4$	AVX2	2.87

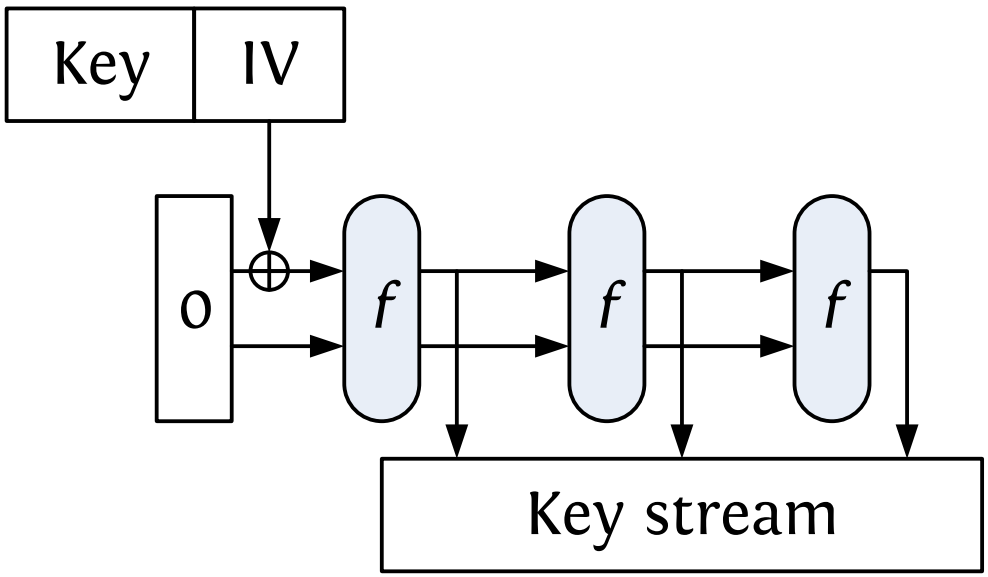
CPU: Haswell with AVX2 256-bit SIMD

MAC (and key derivation)



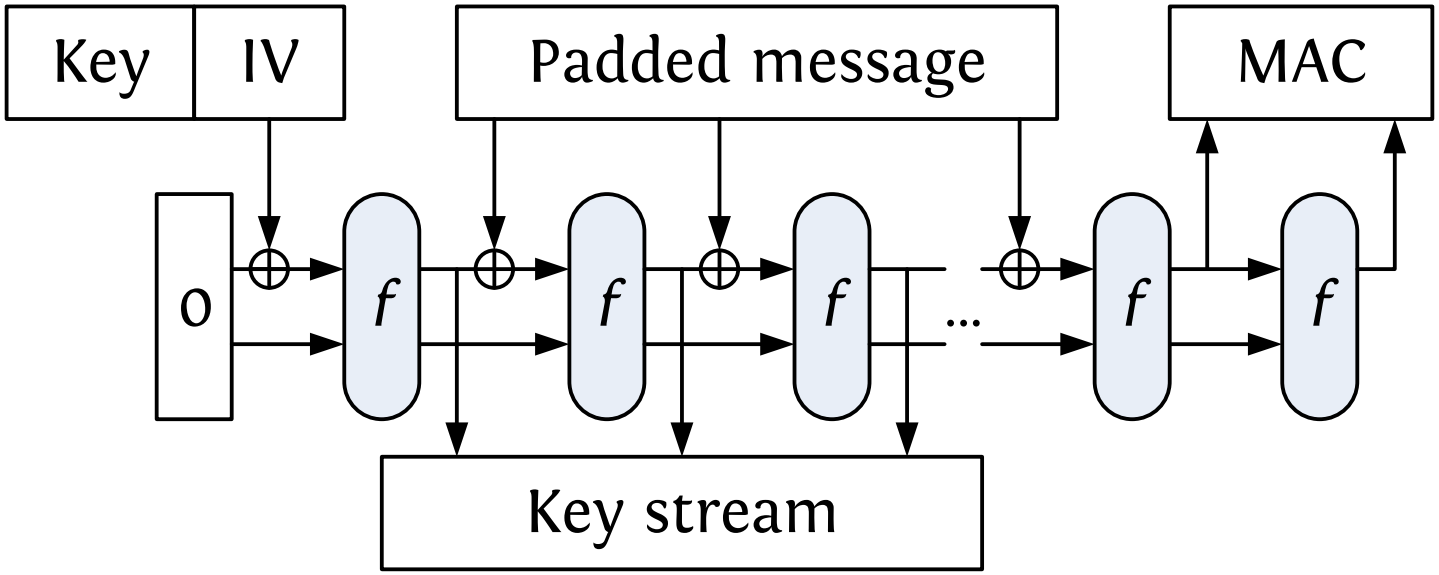
- $\mathrm{KMAC}[K](M) = \mathrm{H}(K||M)$
- $\mathrm{XMAC}[K](M, \lambda) = \mathrm{XOF}(K||M||\lambda)$
  - λ length of the output
- XKDF: key derivation function based on XOF (XMAC)
- HMAC [FIPS 198] no longer needed!

Stream encryption



- Encryption: add key stream to plaintext bit per bit

Single-pass authenticated encryption

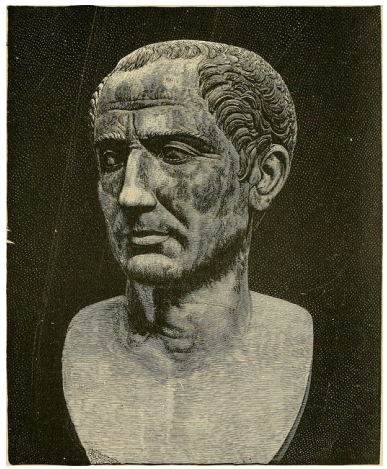


- Encryption with MAC for free!
- Secure messaging (*SSL/TLS*, *SSH*, *IPSEC* ...)
- Same primitive KECCAK-*f* but in a (slightly) different mode
  - **Duplex** construction
  - also for random generation with reseeding (/dev/urandom ...)

Outline

The CAESAR competition

"horum omnium fortissimi sunt Belgae"

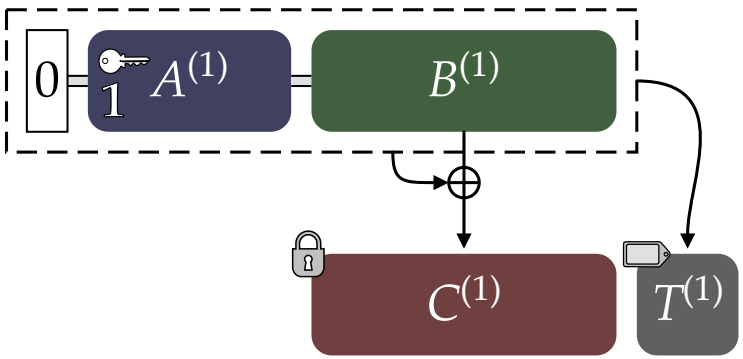


- Public competition for *authenticated ciphers*
  - consortium from academia and industry
  - aims for portfolio instead of single winner
- Timeline
  - submission deadline: March 15, 2014
  - 57 submissions
    - many block cipher modes using AES
    - about a dozen sponge-based,
    - including our submissions: KETJE and KEYAK
  - 3 rounds foreseen
  - target end date: December 2017

http://competitions.cr.yp.to/caesar-submissions.html

KEYAK in a nutshell

- KECCAK-$p[1600, n_r = 12]$ or KECCAK-$p[800, n_r = 12]$, $c = 256$
- sequential and parallel instances



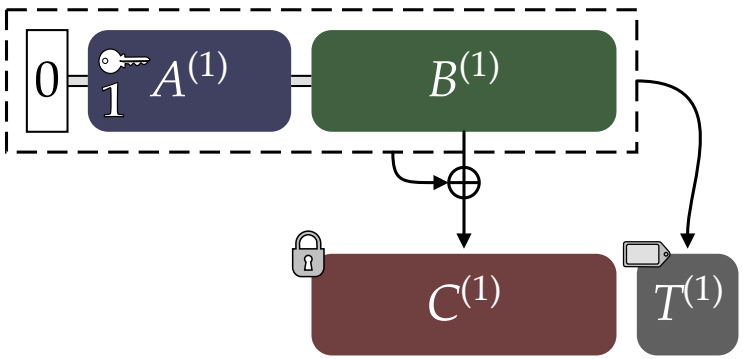
$A^{(1)}$ contains the key and must be unique, e.g.,

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

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

KEYAK in a nutshell

- KECCAK-$p[1600, n_r = 12]$ or KECCAK-$p[800, n_r = 12]$, $c = 256$
- sequential and parallel instances



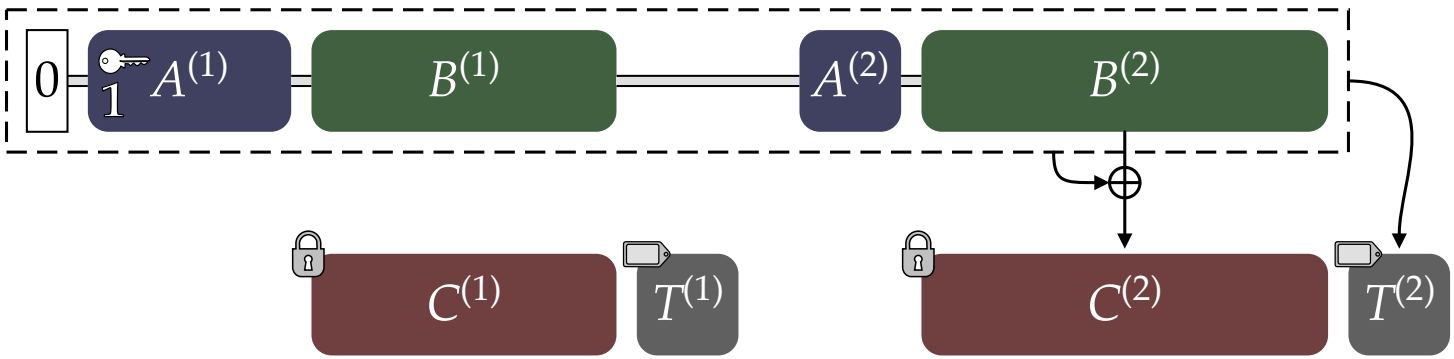
$A^{(1)}$ contains the key and must be unique, e.g.,

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

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

KEYAK in a nutshell

- KECCAK-$p[1600, n_r = 12]$ or KECCAK-$p[800, n_r = 12]$, $c = 256$
- sequential and parallel instances



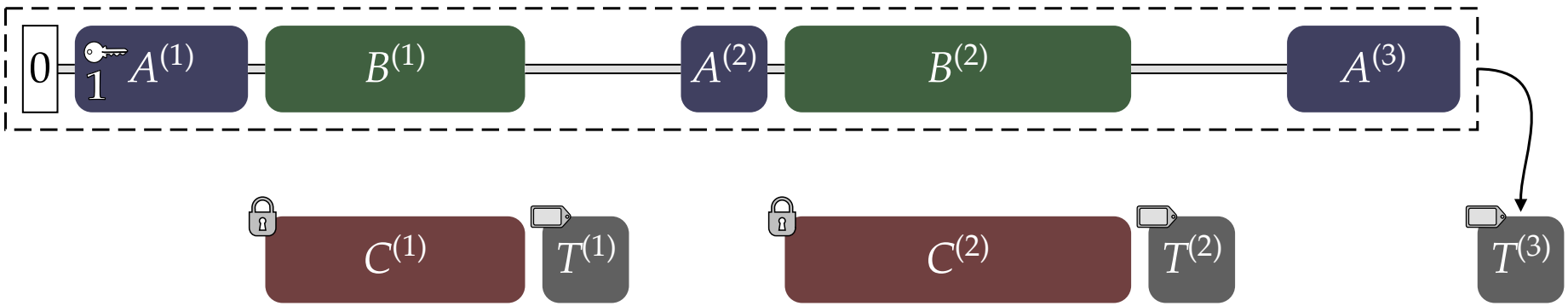
$A^{(1)}$ contains the key and must be unique, e.g.,

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

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

KEYAK in a nutshell

- KECCAK-$p[1600, n_r = 12]$ or KECCAK-$p[800, n_r = 12]$, $c = 256$
- sequential and parallel instances



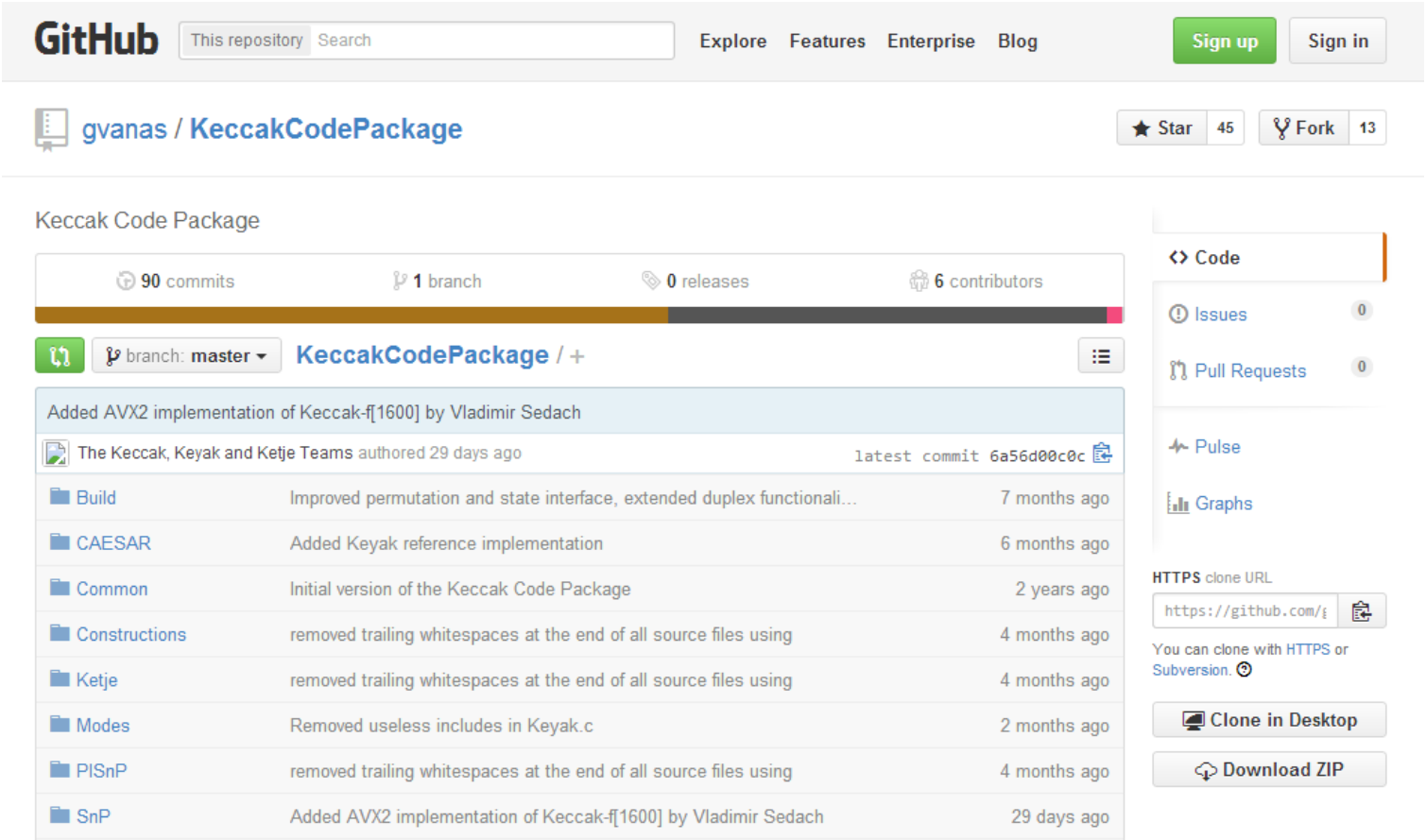
$A^{(1)}$ contains the key and must be unique, e.g.,

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

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

Outline

Where to find the latest KECCAK implementations?



https://github.com/gvanas/KeccakCodePackage

Extending the scope of software implementations?

In the old package, there were

- 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 the old package, there were

- 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 ... ?

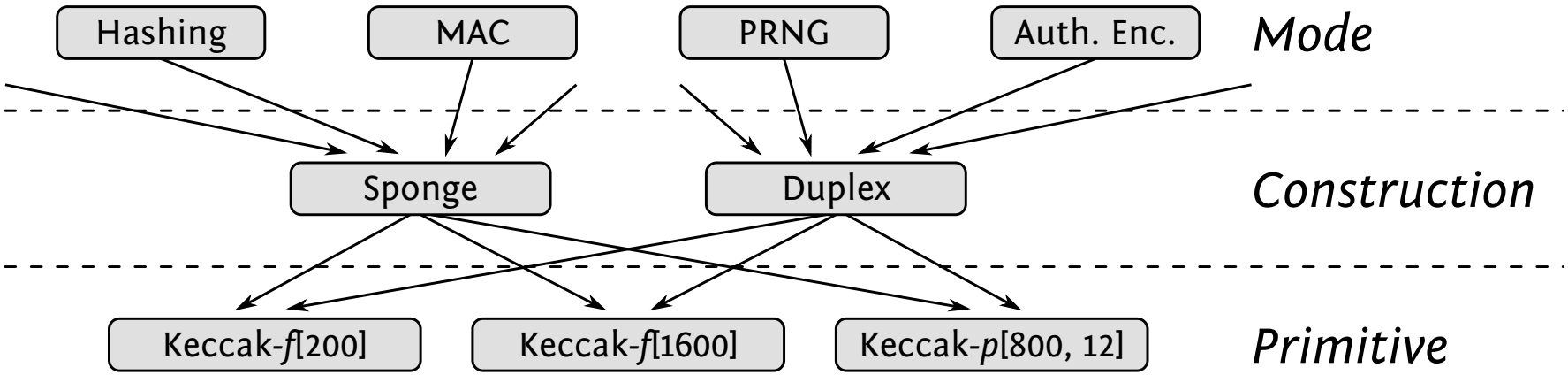
A heterogenous set of software implementations

By Magalie L'Abbé (flickr.com)

There were implementations

- with **different structures**
- with/without flexible capacity
- with/without an input queue

Goals of a layered approach



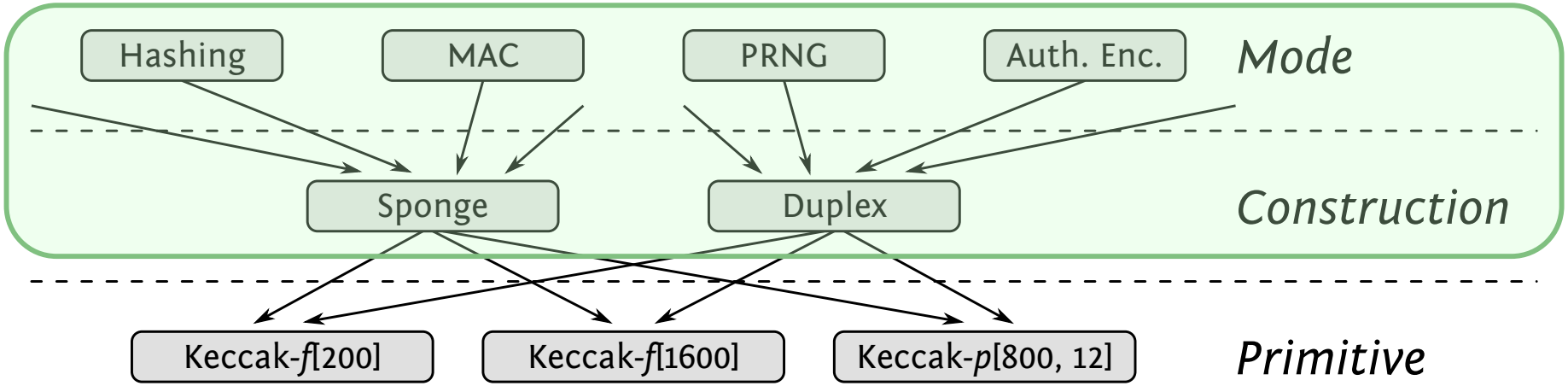
Generic

- 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

Goals of a layered approach



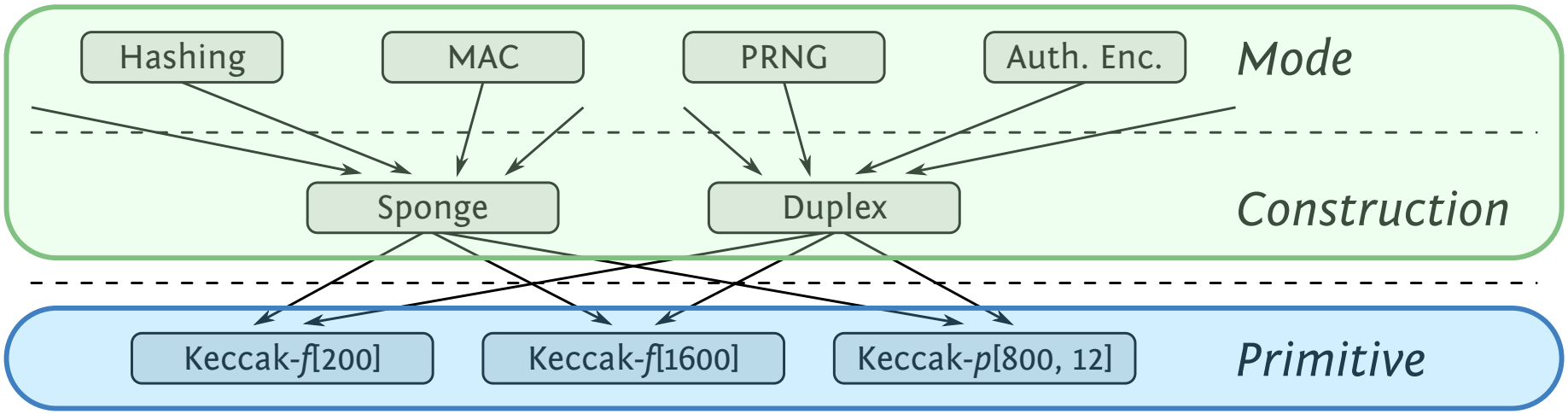
Generic

- 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

Goals of a layered approach



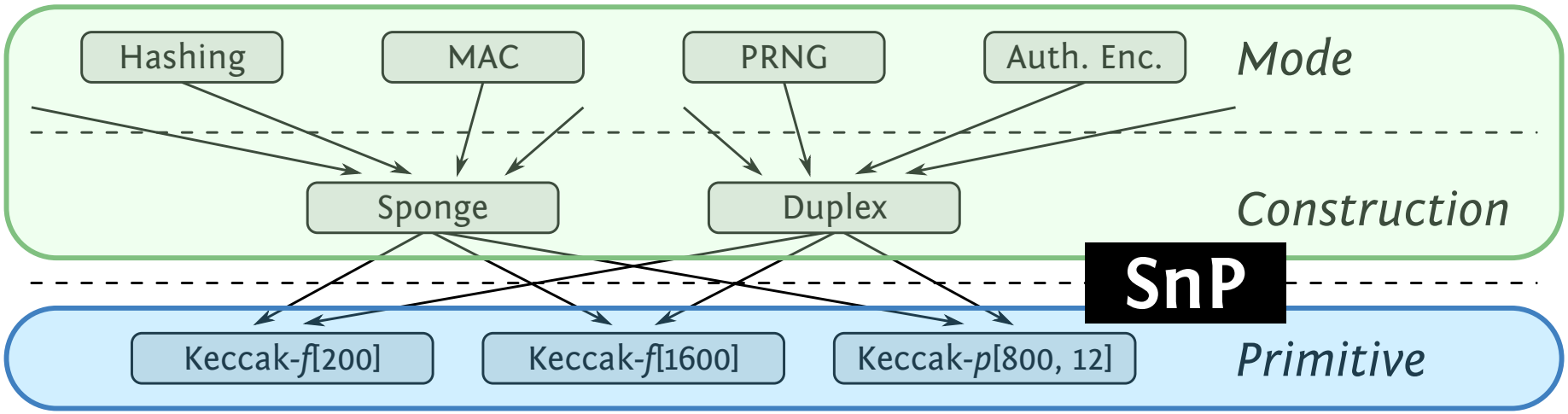
Generic

- 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

Goals of a layered approach



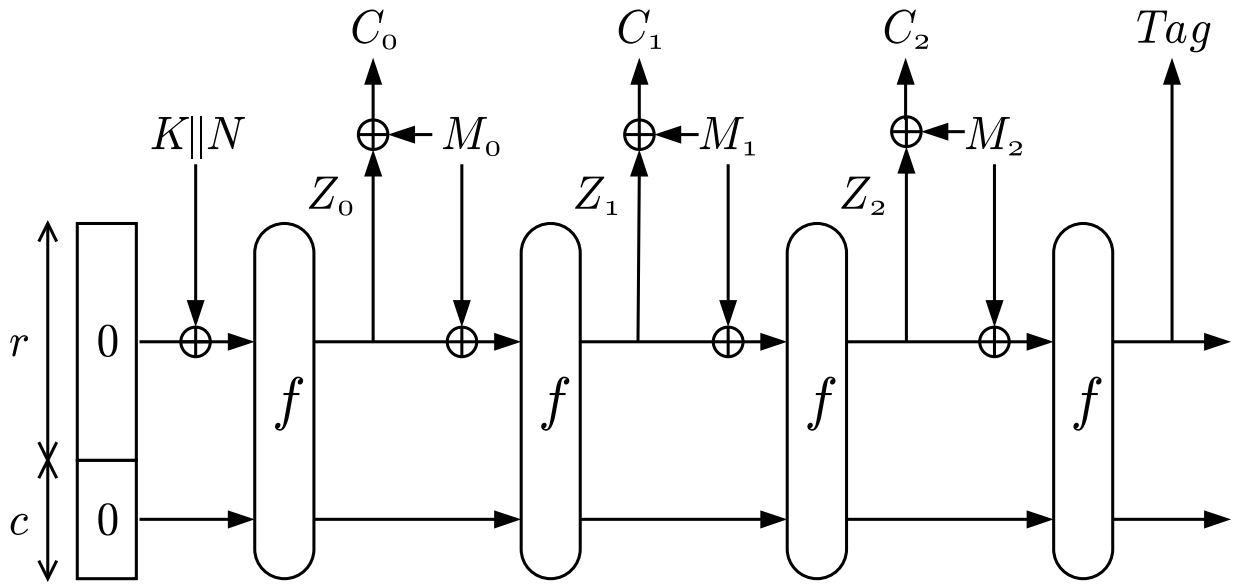
Generic

- 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

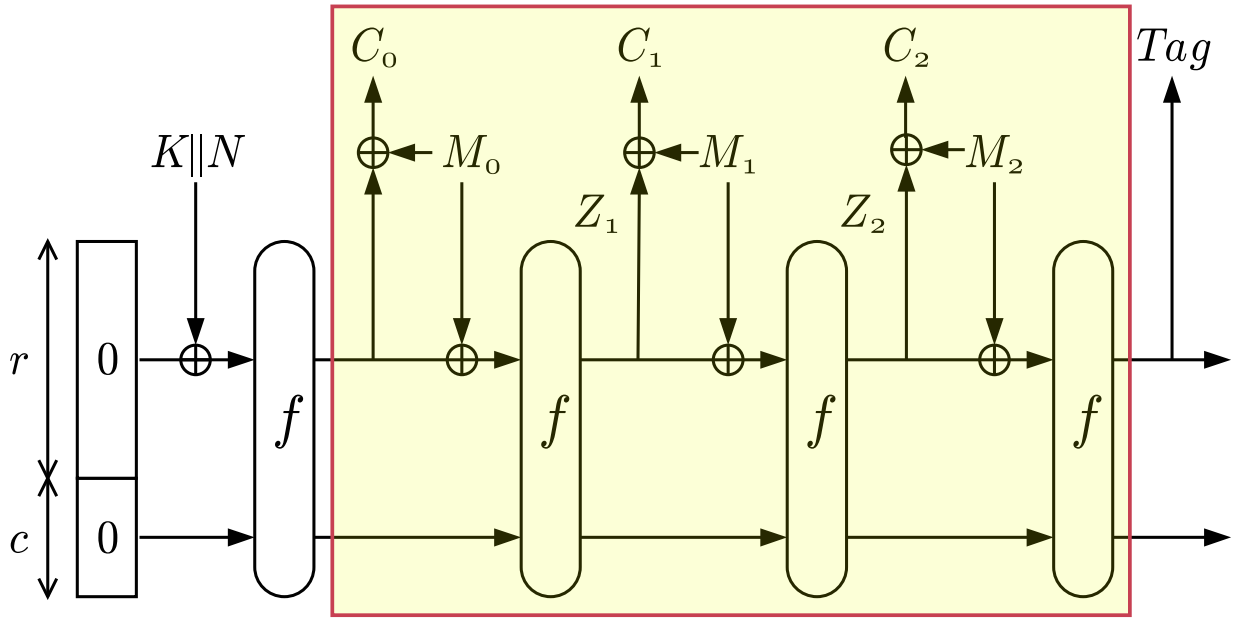
SnP (= State and Permutation)



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)

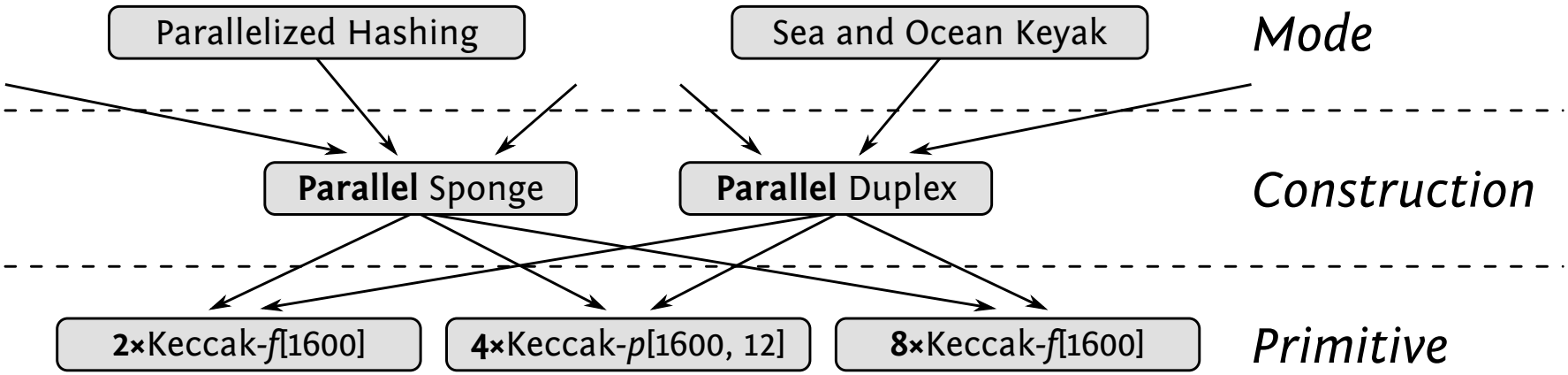


Specialized repeated application of some operations

(optional)

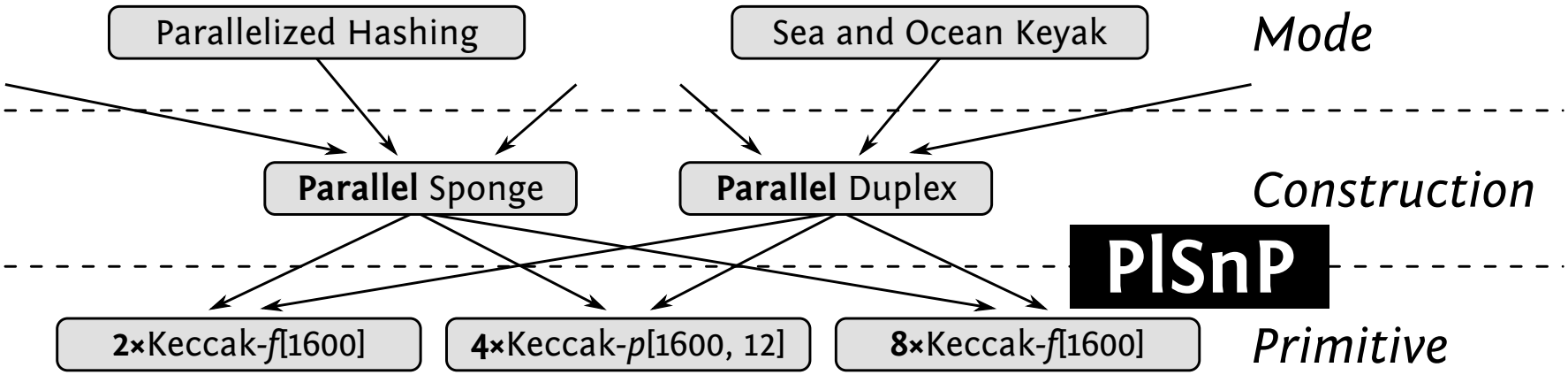
SnP_FBWL_Absorb/Squeeze/Wrap/Unwrap

Parallel processing



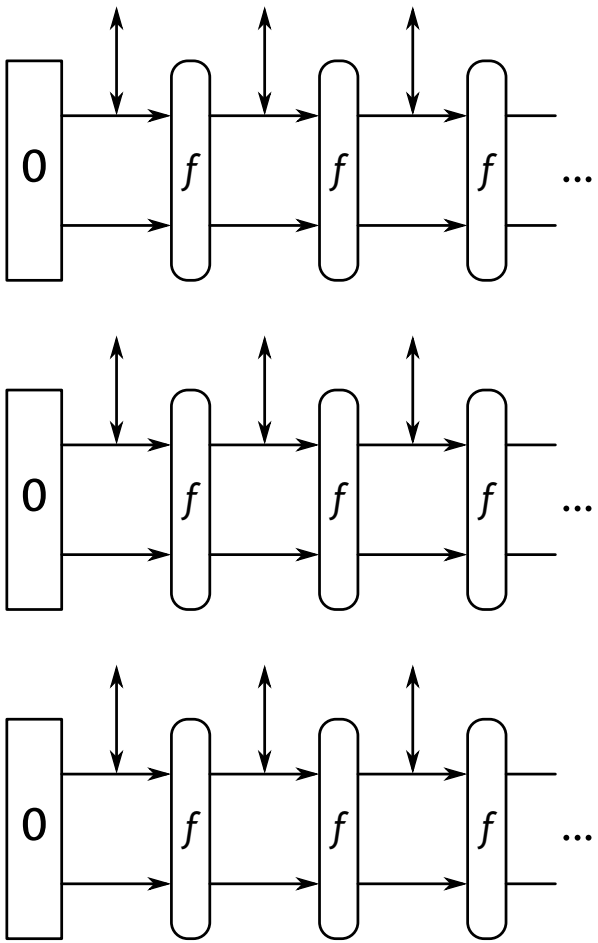
- Some modes exploit parallelism
- To exploit this, we need:
  - sponge functions and duplex objects running in parallel
  - permutation applied on several states in parallel

Parallel processing



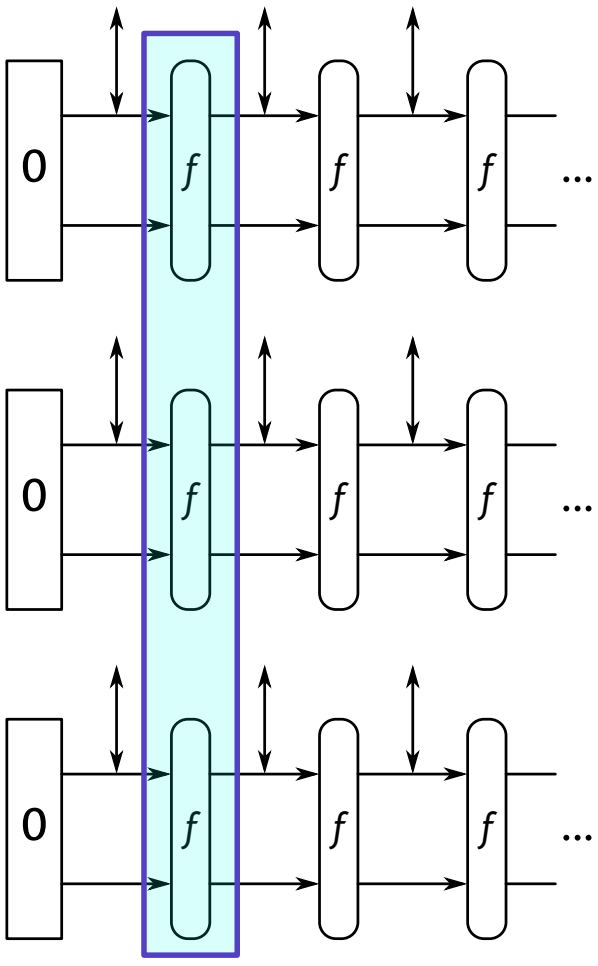
- Some modes exploit parallelism
- To exploit this, we need:
  - sponge functions and duplex objects running in parallel
  - permutation applied on several states in parallel

PlSnP (= Parallel States and Permutations)



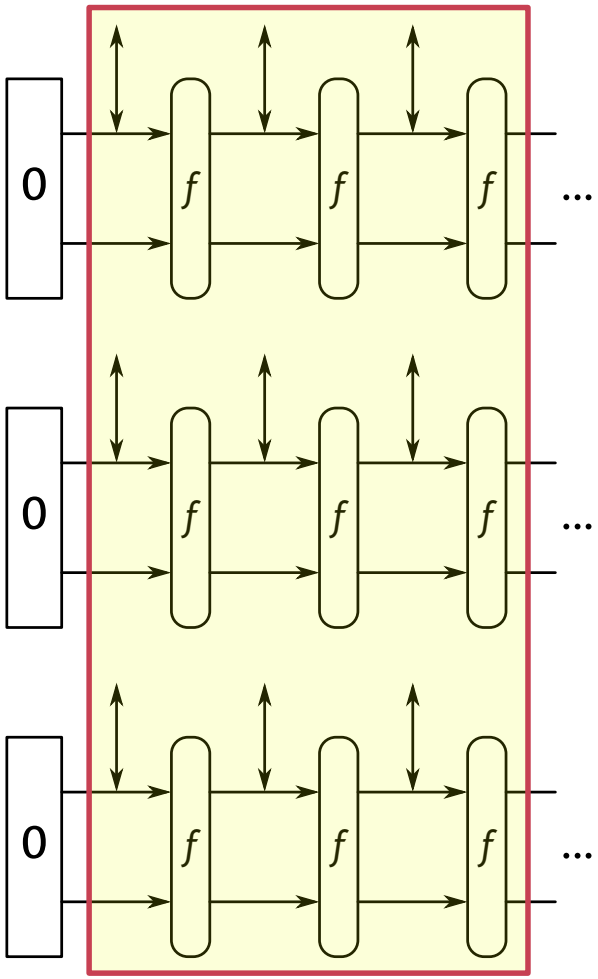
- SnP on individual instances
- Some SnP functions parallelized
  - Parallel application of f
- **PlSnP FBWL** for repeated operations

PlSnP (= Parallel States and Permutations)



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

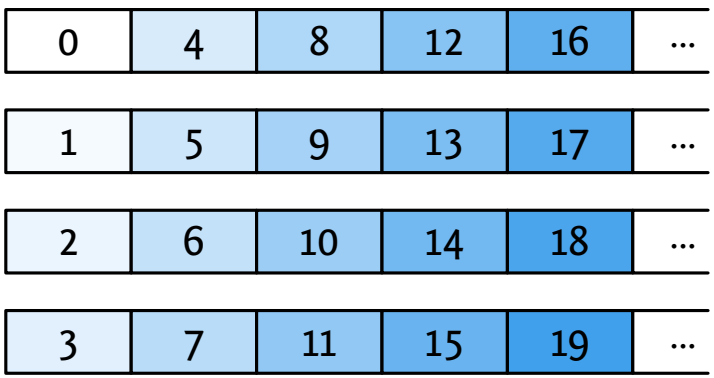
PlSnP (= Parallel States and Permutations)



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

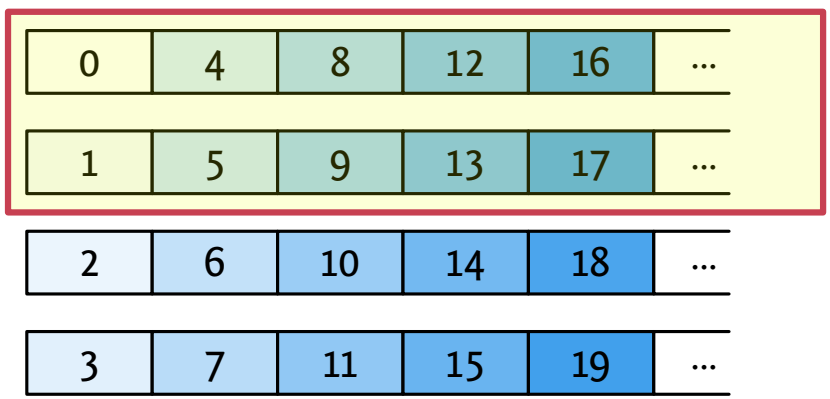
PlSnP FBWL: parameterized block layout

Interleaving (blocks of r bits) in 4 lines



PlSnP FBWL: parameterized block layout

Interleaving (blocks of r bits) in 4 lines

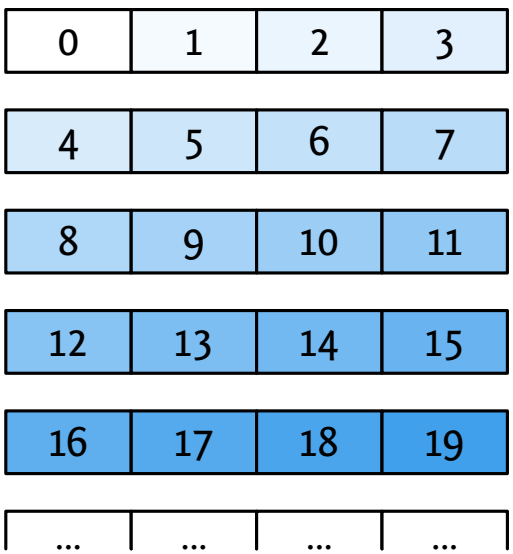


Assuming 2-way parallelism:

→ 4 blocks
↓ 1 block

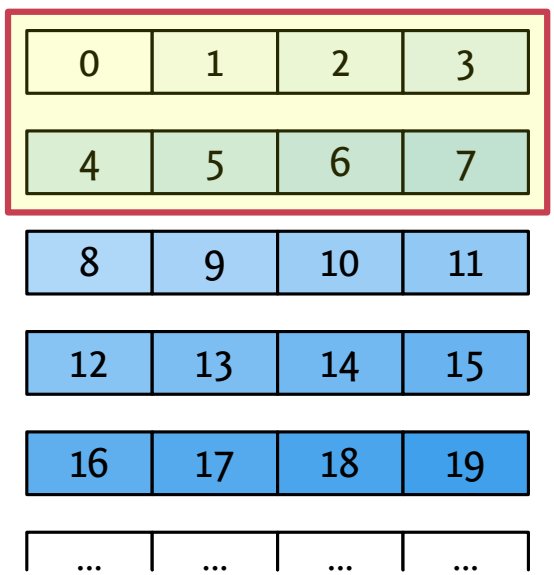
PlSnP FBWL: parameterized block layout

Segmenting in 4 blocks of r bits each



PlSnP FBWL: parameterized block layout

Segmenting in 4 blocks of r bits each



Assuming 2-way parallelism:

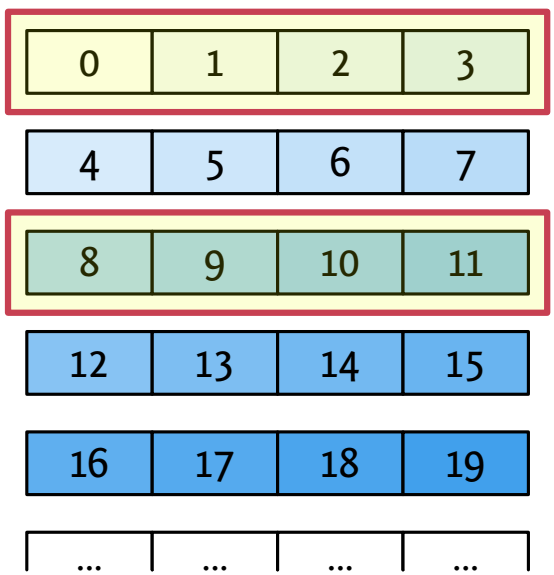
→ 1 block

↓ 4 blocks

(2 consecutive lines)

PlSnP FBWL: parameterized block layout

Segmenting in 4 blocks of r bits each



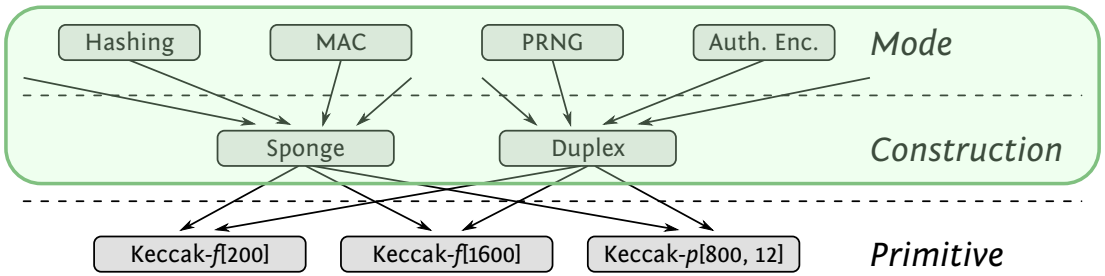
Assuming 2-way parallelism:

→ 1 block

↓ 8 blocks

(even/odd lines)

Constructions and modes



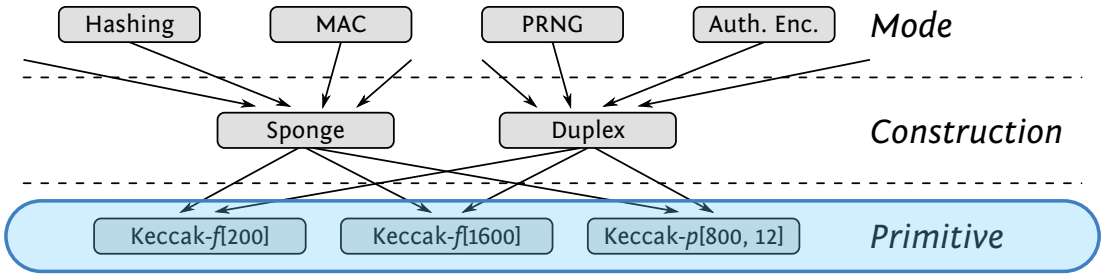
Currently in the KCP

- SHA-3 hashing and SHAKE XOFs
- RIVER and LAKE KEYAK
- KETJE (*)
- Anything using sponge or duplex directly

Nice to have

- Pseudo-random bit sequence generator

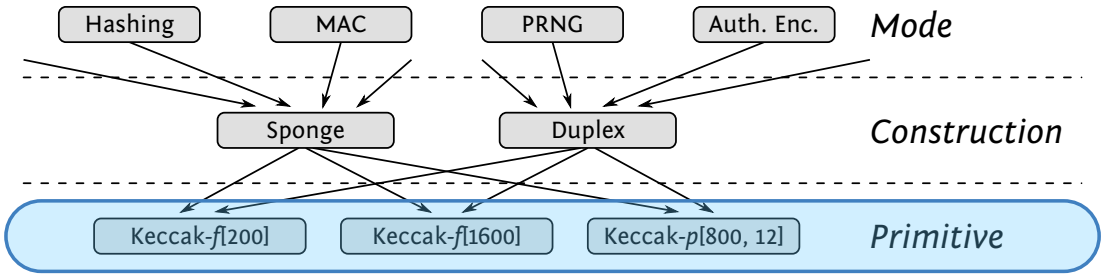
Primitives



KECCAK-$f[200 \text{ to } 1600]$, KECCAK-$p[200 \text{ to } 1600, n_r]$

- Reference implementations
- Optimized impl. in C of KECCAK-$f[1600]$ and -$p[1600, n_r = 12]$
- Optimized impl. in C of KECCAK-$f[800]$ and -$p[800, n_r = 12]$
- Assembly optimized for
  - x86_64 (KECCAK-$f[1600]$ and KECCAK-$p[1600, n_r = 12]$ only)
  - ARMv6M, ARMv7M, ARMv7A, NEON
  - AVR8

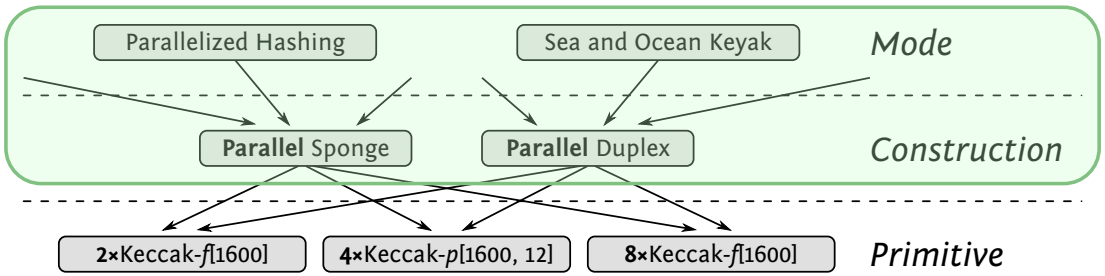
Primitives



On the to-do list

- Some implementations still to be migrated
- Optimized in C for 400-bit width and smaller
- ARMv8, AVX-512, (your favorite platform here)

Parallel constructions and modes



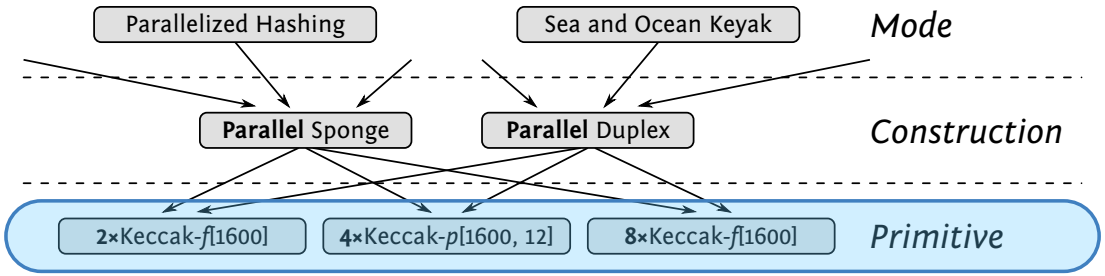
Currently in the KCP

- 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
- $2 \times$ KECCAK-$f[1600]/p[1600, n_r = 12]$ using SSE, XOP or AVX

Many things on the to-do list

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

If you want to help...

We welcome comments and contributions on:

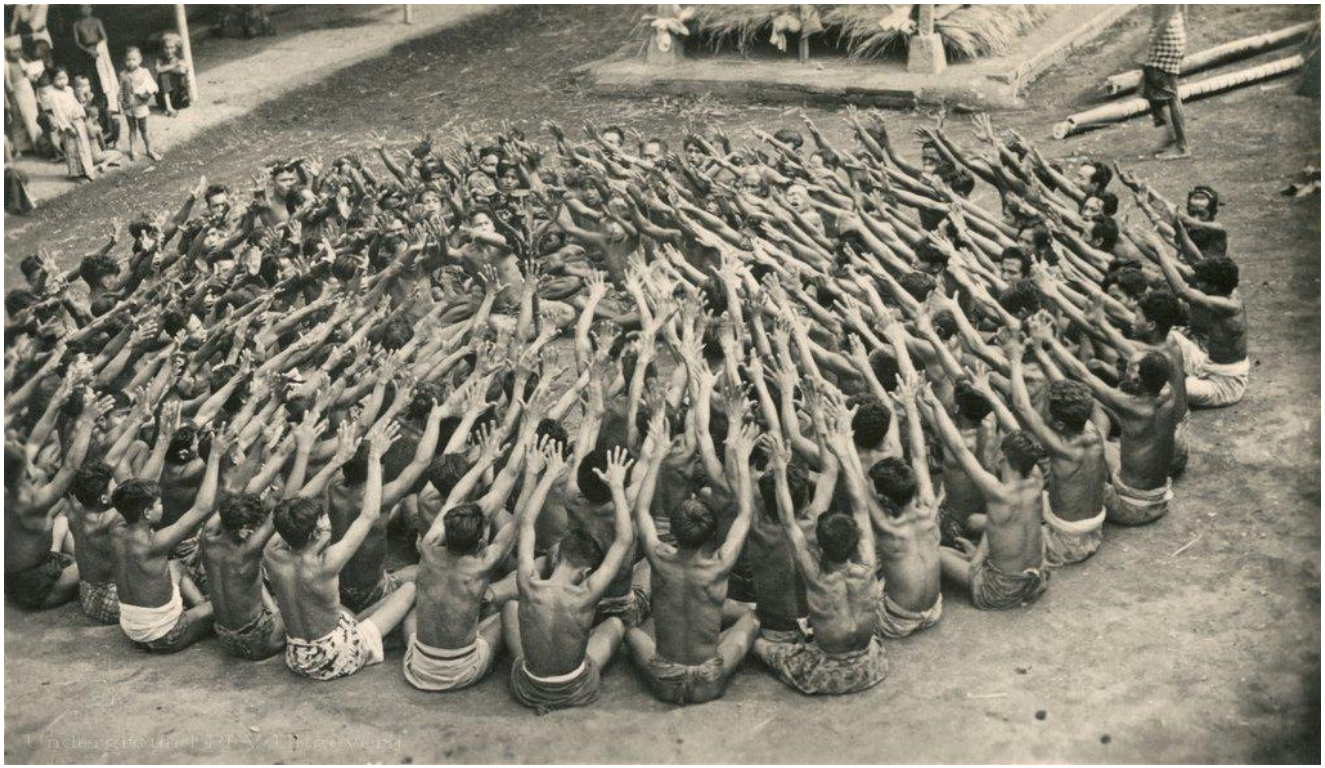
- better/more **optimized** implementations
- stucture of the package
  - ... a library ... ?

By @Doug88888 (flickr.com)

keccak@noekeon.org

https://github.com/gvanas/KeccakCodePackage

Questions?



More information on
http://sponge.noekeon.org/
http://keccak.noekeon.org/