TOPIC: 6.857

DATE: 2/20/13

FILE UNDER:

PAGE: L5.1

Admin:

Today:

- ☐ More hash function applications
- ☐ Keccak overview (SHA-3)

TOPIC

FILE UNDER

DATE

Theorem: If h is CR, then h is TCR.
(But converse doesn't hold.)

Theorem: h is OW $\not\Leftrightarrow$ h is CR
(neither implication holds)
But if h "compresses", then CR $\Rightarrow$ OW.

Hash function applications

① Password storage (for login)
- Store h(PW), not PW, on computer
- when user logs in, check hash of his PW agaist table.
- Disclosure of h(PW) should not reveal PW (or any equivalent pre-image)
- Need OW

② File modification detector
- For each file F, store h(F) securely (e.g. on off-line DVD)
- Can check if F has been modified by recomputing h(F)
- need WCR (aka TCR)
(Adversary wants to change F but not h(F).)

- Hashes of downloadable software = equivalent problem.

③ Digital signatures ("hash & sign")

PK_A = Alice's public key (for signature verification)

SK_A = Alice's secret key (for signing)

Signing: $\sigma = \text{sign}(SK_A, M)$ [Alice's sig on M]

Verify: $\text{Verify}(M, \sigma, PK_A) \in \{\text{True}, \text{False}\}$

Adversary wants to forge a signature that verifies.

- For large M, easier to sign h(M):

  $\sigma = \text{sign}(SK_A, h(M))$ ["hash & sign"]

  Verifier recomputes h(M) from M, then verifies σ.

  In essence, h(M) is a "proxy" for M.

- Need CR (Else Alice gets Bob to sign x, where $h(x) = h(x')$, then claims Bob really signed x', not x.)

- Don't need OW (e.g. h = identity is OK here.)

TOPIC:

DATE:

FILE UNDER:

④ Commitments

- Alice has value x (e.g. auction bid)
- Alice computes $C(x)$ ("commitment to x") & submits $C(x)$ as her "sealed bid"
- When bidding has closed, Alice should be able to "open" $C(x)$ to reveal x
- Binding property: Alice should not be able to open $C(x)$ in more than one way! (She is committed to just one x.)
- Secrecy (hiding): Auctioneer (or anyone else) seeing $C(x)$ should not learn anything about x.
- Non-malleability: Given $C(x)$, it shouldn't be possible to produce $C(x+1)$, say..
- How:

$$C(x) = h(r \| x) \quad r \in_R \{0,1\}^{256}$$

To open: reveal r & x

- Note that this method is randomized (as it must be for secrecy.
- Need: OW, CR, NM

(really need more, for secrecy, as $C(x)$ should not reveal partial information about x, even.)

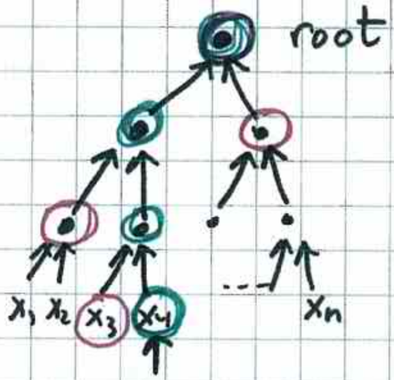
TOPIC:

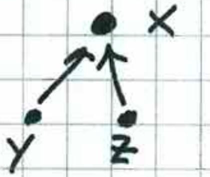
DATE:

FILE UNDER:

⑤ <u>To authenticate a collection of n objects</u>:

Build a tree with n leaves $x_1, x_2, \ldots, x_n$
& compute authenticator node as fn of values
at children... This is a "Merkle tree":





value at x
= h (value at y || value at z)

Root is authenticator for all n values $x_1, x_2, \ldots, x_n$
To authenticate x_i, give sibling of x_i &
sibling of all his ancestors up to root
Apply to: time-stamping data
authenticating whole file system
<u>Need:</u> CR

TOPIC:

FILE UNDER:

DATE:

PAGE

Hash function construction ("Merkle-Damgard" style)

- Choose output size d (e.g. $d = 256$ bits)
- Choose "chaining variable" size c (e.g. $c = 512$ bits)

  [Must have $c \geq d$; better if $c \geq 2 \cdot d$ ...]
- Choose "message block size" b (e.g. $b = 512$ bits)
- Design "compression function" f

  $$f : \{0,1\}^c \times \{0,1\}^b \rightarrow \{0,1\}^c$$

  [f should be OW, CR, PR, NM, TCR, ...]
- Merkle-Damgard is essentially a "mode of operation" allowing for variable-length inputs:

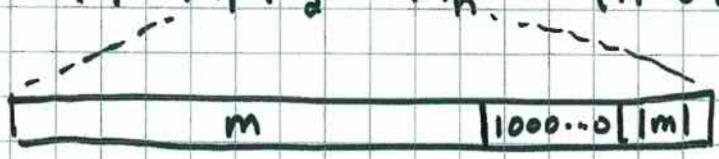
  * Choose a c-bit initialization vector IV, c_0

  [Note that c_0 is fixed & public.]

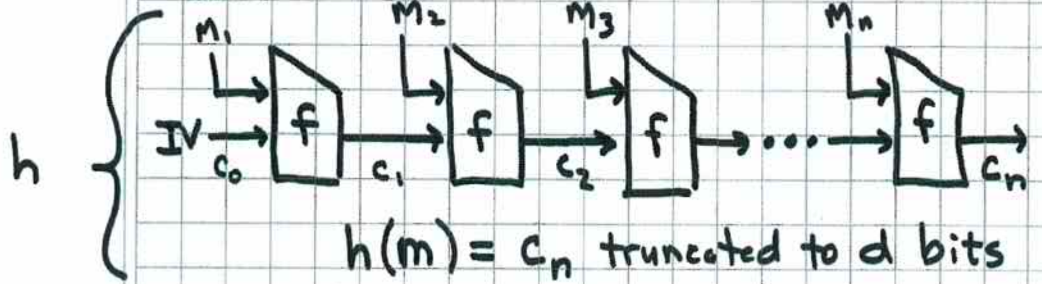
  * [Padding] Given message, append
    - 10^* bits
    - fixed-length representation of length of input

  so result is a multiple of b bits in length:

  $$M = M_1 \, M_2 \cdots M_n \quad (n \text{ b-bit blocks})$$



Then:



$h(m) = c_n$ truncated to d bits

<u>Theorem:</u> If f is CR, then so is h.

<u>Proof:</u> Given collision for h, can find one for f by working backwards through chain. $\blacksquare$

<u>Thm:</u> Similarly for OW.

<u>Common design pattern for f:</u>

$$f(c_{i-1}, M_i) = c_{i-1} \oplus E(M_i, c_{i-1})$$

where $E(K, M)$ is an encryption function (block cipher) with b-bit key and c-bit input/output blocks.

(Davies-Meyer construction)

TOPIC:

DATE:

FILE UNDER:

NOT DONE

"Merkle-Damgård Revisited" (Coron, Dodis, Malinaud, Puniya)

Is MD a "good" method?

What does this mean?

Suppose that f is a random oracle (fixed input length)

$$f: \{0,1\}^{b+c} \rightarrow \{0,1\}^c$$

Then is MD^f indistinguishable from a VIL RO?

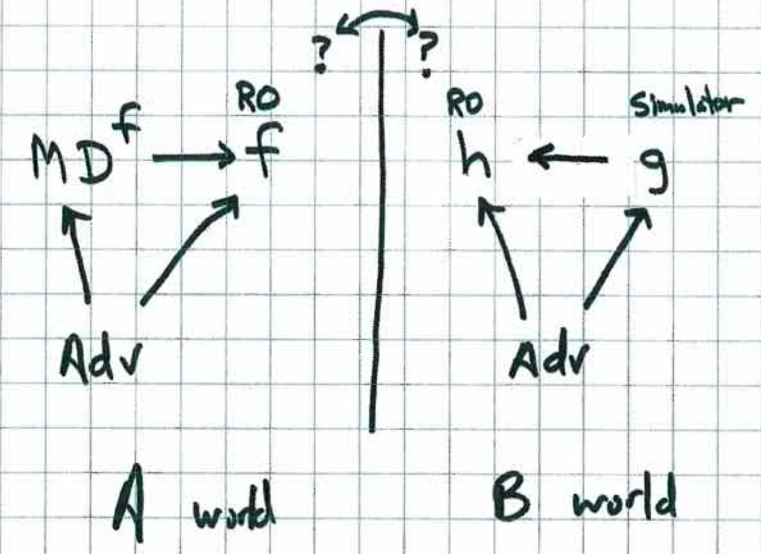
(VIL = "variable input length")

Adversary has access to:

Ⓐ MD^f and also to f (f is FIL)

Ⓑ RO h and also to g (h is VIL & g FIL)

where g is constructed to bear same relation to h as f does to MD^f ("simulator")



Note: g may call h, but doesn't see Adv's calls to h.

TOPIC:

DATE:

FILE UNDER:

NOT DONE

Standard construction MD^f fails (for $c=d$):

Can't build simulator g to bear right relation to h (i.e. so that h appears to be MD^g)

Example of problem (message extension): (sketch)

h & g should satisfy

$$MD^g(m_1 \| m_2) = h(m_1 \| m_2) = g(g(IV, m_1), m_2)$$

```
Adv: [ computes $u = h(m_1)$
     [ computes $v = g(u, m_2)$                (*)
     [ computes $w = h(m_1, m_2)$
     [ if $v = w$: answer "A world"
     [ else:       answer "B world"
```

Adv always right in A world, and almost always right in B world, since simulator g doesn't know how to answer query (*). [It didn't see query for u, so even though it can access h, it doesn't have ability to figure out m_1, and so reply to (*) in way that makes it consistent with $h(m_1, m_2)$.]

TOPIC:

DATE:

FILE UNDER:

NOT DONE

(But perhaps interesting...)

But, it is not hard to fix MD construction so it becomes "indistinguishable from RO" (given FIL RO f) [technically this is called "indifferentiability"].

Four methods: (any work to fix MD)

① Encode m to be "prefix-free" before applying MD:

e.g. $0 \| m_1 \| 0 \| m_2 \| 0 \| m_3 \| \cdots \| 1 \| m_n$

or $\underset{\uparrow}{L} \| m_1 \| m_2 \| \cdots \| m_n$

length of message in bits
m_n padded with 10^*

② Drop output bits:

Let $d = c/2$. Drop $c/2$ bits of output.

③ NMAC construction

$$g(MD^f(M))$$

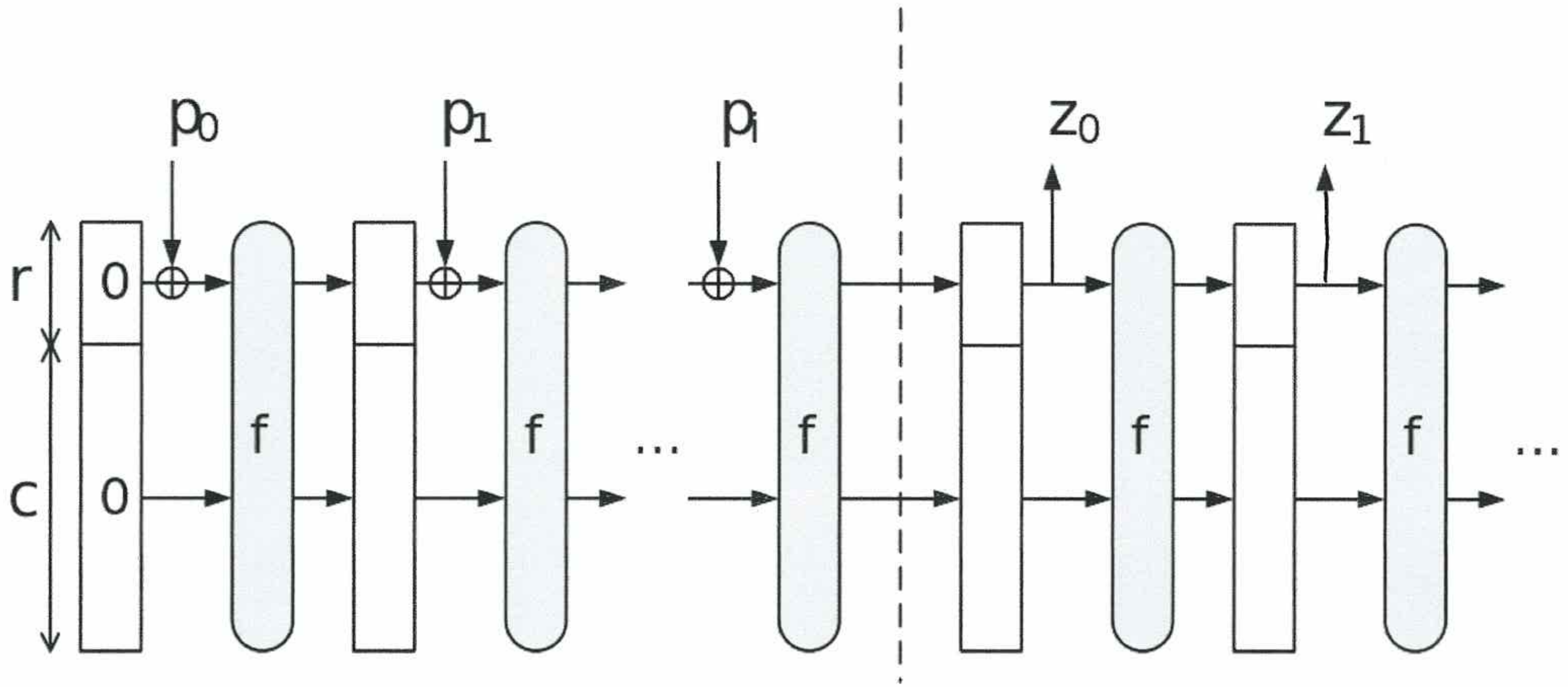
[g indep. function from $\{0,1\}^c$ to $\{0,1\}^d$]

④ HMAC construction:

$$MD^f(MD^f(M))$$

* With such methods, it is then "safe" to treat (modified) MD^f as a RO (assuming f is indistinguishable from a FIL R.O.)

Keccak



Keccak Sponge Construction

- d = output hash size in bits $\in \{224, 256, 384, 512\}$
- $c = 2d$ bits
- state size = $25w$ where w = word size (e.g. $w = 64$)
- $c + r = 25w$
- $r \geq d$ (so hash can be first d bits of z_0)
- Input padded with 10^*1 until length is a multiple of r
- f has 24 rounds (for $w=64$), not quite identical (round constant)
- f is public, efficient, invertible function from $\{0,1\}^{25w}$ to $\{0,1\}^{25w}$

e.g. $d = 256$
$c = 512$
$r = 1088$
$w = 64$

SHA-3

From Wikipedia, the free encyclopedia

SHA-3, originally known as **Keccak** (pronounced [kɛtʃak], like “ketchak”),[1] is a cryptographic hash function designed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, building upon RadioGatún. On October 2, 2012, Keccak was selected as the winner of the NIST hash function competition.[2] SHA-3 is not meant to replace SHA-2, as no significant attack on SHA-2 has been demonstrated. Because of the successful attacks on MD5, SHA-0 and theoretical attacks on SHA-1, NIST perceived a need for an alternative, dissimilar cryptographic hash, which became SHA-3. The authors claim 12.5 cycles per byte[3] on an Intel Core 2 CPU. However, in hardware implementations it is notably faster than all other finalists.[4]

SHA-3

General	
Designers	Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche.
Certification	SHA-3 winner
Detail	
Digest sizes	arbitrary
Speed	12.5 cpb on Core 2 [r=1024,c=576].

SHA-3 uses the sponge construction[5][6] in which message blocks are XORed into the initial bits of the state, which is then invertibly permuted. In the version used in SHA-3, the state consists of a 5×5 array of 64-bit words, 1600 bits total.

Keccak's authors have proposed additional, not-yet-standardized uses for the function, including an authenticated encryption system and a "tree" hash for faster hashing on certain architectures.[7] Keccak is also defined for smaller power-of-2 word sizes w down to 1 bit (25 bits total state). Small state sizes can be used to test cryptanalytic attacks, and intermediate state sizes (e.g., from w=4, 100 bits, to w=32, 800 bits) could potentially provide practical, lightweight, alternatives.

Contents

The block permutation

This is defined for any power-of-two word size, $w = 2^{\ell}$ bits. The main SHA-3 submission uses 64-bit words, $\ell = 6$.

The state can be considered to be a 5×5×w array of bits. Let $a[i][j][k]$ be bit $(i\times 5 + j)\times w + k$ of the input, using a little-endian convention. Index arithmetic is performed modulo 5 for the first two dimensions and modulo *w* for the third.

The basic block permutation function consists of 12+2 ℓ iterations of five sub-rounds, each individually very simple:

θ

Compute the parity of each of the 5×*w* (320, when *w* = 64) 5-bit columns, and exclusive-or that into two nearby columns in a regular pattern. To be precise, $a[i][j][k] \oplus= \text{parity}(a[0..4][j-1][k]) \oplus \text{parity}(a[0..4][j+1][k-1])$

ϱ

Bitwise rotate each of the 25 words by a different triangular number 0, 1, 3, 6, 10, 15, ... To be precise, $a[0][0]$ is not rotated, and for all $0 \le t \le 24$, $a[i][j][k] = a[i][j][k-(t+1)(t+2)/2]$, where

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}^t \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

π

Permute the 25 words in a fixed pattern. $a[j][2i+3j] = a[i][j]$

χ

Bitwise combine along rows, using $a = a \oplus (\neg b \,\&\, c)$. To be precise, $a[i][j][k] \oplus= \neg a[i][j+1][k] \,\&\, a[i][j+2][k]$. This is the only non-linear operation in SHA-3.

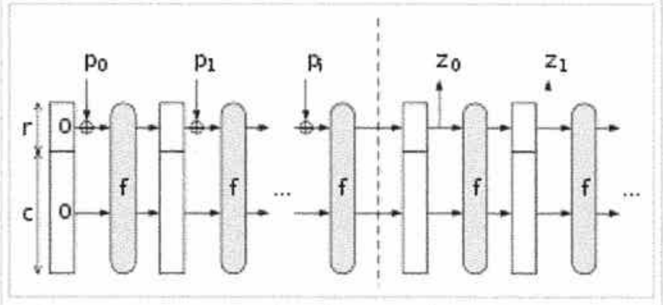
ι

Exclusive-or a round constant into one word of the state. To be precise, in round *n*, for $0 \le m \le \ell$, $a[0][0][2^m-1]$ is exclusive-ORed with bit $m+7n$ of a degree-8 LFSR sequence. This breaks the symmetry that is preserved by the other sub-rounds.

Hashing variable-length messages

SHA-3 uses the "sponge construction", where input is "absorbed" into the hash state at a given rate, then an output hash is "squeezed" from it at the same rate.

To absorb *r* bits of data, the data is XORed into the leading bits of the state, and the block permutation is applied. To squeeze, the first *r* bits of the state are produced as output, and the block permutation is applied if additional output is desired.



The sponge construction for hash functions. p_i are input, z_i are hashed output. The unused "capacity" c should be twice the desired resistance to collision or preimage attacks.

Central to this is the "capacity" of the hash function, which is the $c=25w-r$ state bits that are not touched by input or output. This can be adjusted based on security requirements, but the SHA-3 proposal sets a conservative $c=2n$, where *n* is the size of the output hash. Thus *r*, the number of message bits processed per block permutation, depends on the output hash size. The rate *r* is 1152, 1088, 832, or 576 (144, 136, 104 and 72 bytes) for 224, 256, 384 and 512-bit hash sizes, respectively, when *w* is 64.

To ensure the message can be evenly divided into *r*-bit blocks, it is padded with the bit pattern 10^*1: a 1 bit, zero or more 0 bits (maximum $r-1$), and a final 1 bit. The final 1 bit is required because the sponge construction security proof requires that the final message block is not all-zero.

To compute a hash, initialize the state to 0, pad the input, and break it into r-bit pieces. Absorb the input into the state; that is, for each piece, XOR it into the state and then apply the block permutation.

After the final block permutation, the leading n bits of the state are the desired hash. Because r is always greater than n, there is actually never a need for additional block permutations in the squeezing phase. However, arbitrary output length may be useful in applications such as optimal asymmetric encryption padding. In this case, n is a security parameter rather than the output size.

Although not part of the SHA-3 competition requirements, smaller variants of the block permutation can be used, for hash output sizes up to half their state size, if the rate r is limited appropriately. For example, a 256-bit hash can be computed using 25 32-bit words if $r = 800–2×256 = 288$ (36 bytes per iteration).

Tweaks

Throughout the NIST hash function competition, entrants are permitted to "tweak" their algorithms to address issues that are discovered. Changes that have been made to Keccak are:

- The number of rounds was increased from $12+\ell$ to $12+2\ell$ to be more conservative about security.
- The message padding was changed from a more complex scheme to the simple 10^*1 pattern described above.
- The rate r was increased to the security limit, rather than rounding down to the nearest power of 2.

Examples of SHA-3 (Keccak) variants

Note: Pending the standardization of SHA-3, there is no specification of particular SHA-3 functions yet.

Hash values of empty string.

```
Keccak-224("")
0x f71837502ba8e10837bdd8d365adb85591895602fc552b48b7390abd
Keccak-256("")
0x c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
Keccak-384("")
0x 2c23146a63a29acf99e73b88f8c24eaa7dc60aa771780ccc006afbfa8fe2479b2dd2b21362337441ac12b515911957ff
Keccak-512("")
0x 0eab42de4c3ceb9235fc91acffe746b29c29a8c366b7c60e4e67c466f36a4304c00fa9caf9d87976ba469bcbe06713b4
```

Even a small change in the message will (with overwhelming probability) result in a mostly different hash, owing to the avalanche effect. For example, adding a period to the end of the sentence:

```
Keccak-224("The quick brown fox jumps over the lazy dog")
0x 310aee6b30c47350576ac2873fa89fd190cdc488442f3ef654cf23fe
Keccak-224("The quick brown fox jumps over the lazy dog.")
0x c59d4eaeac728671c635ff645014e2afa935bebffdb5fbd207ffdeab

Keccak-256("The quick brown fox jumps over the lazy dog")
0x 4d741b6f1eb29cb2a9b9911c82f56fa8d73b04959d3d9d222895df6c0b28aa15
Keccak-256("The quick brown fox jumps over the lazy dog.")
0x 578951e24efd62a3d63a86f7cd19aaa53c898fe287d2552133220370240b572d
```

```
Keccak-384("The quick brown fox jumps over the lazy dog")
0x 283990fa9d5fb731d786c5bbee94ea4db4910f18c62c03d173fc0a5e494422e8a0b3da7574dae7fa0baf005e504063b3
Keccak-384("The quick brown fox jumps over the lazy dog.")
0x 9ad8e17325408eddb6edee6147f13856ad819bb7532668b605a24a2d958f88bd5c169e56dc4b2f89ffd325f6006d820b

Keccak-512("The quick brown fox jumps over the lazy dog")
0x d135bb84d0439dbac432247ee573a23ea7d3c9deb2a968eb31d47c4fb45f1ef4422d6c531b5b9bd6f449ebcc449ea94d
Keccak-512("The quick brown fox jumps over the lazy dog.")
0x ab7192d2b11f51c7dd744e7b3441febf397ca07bf812cceae122ca4ded6387889064f8db9230f173f6d1ab6e24b6e50f
```

References

1. ^ "The Keccak sponge function family: Specifications summary" (http://keccak.noekeon.org/specs_summary.html) . http://keccak.noekeon.org/specs_summary.html. Retrieved 2011-05-11.
2. ^ "NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition" (http://www.nist.gov/itl/csd/sha-100212.cfm) . NIST. http://www.nist.gov/itl/csd/sha-100212.cfm. Retrieved 2012-10-02.
3. ^ Keccak implementation overview Version 3.2 http://keccak.noekeon.org/Keccak-implementation-3.2.pdf
4. ^ Guo, Xu; Huang, Sinan; Nazhandali, Leyla; Schaumont, Patrick (Aug. 2010), "Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations" (http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/SCHAUMONT_SHA3.pdf) , *NIST 2nd SHA-3 Candidate Conference*: 12, http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/SCHAUMONT_SHA3.pdf, retrieved 2011-02-18 Keccak is second only to Luffa, which did not advance to the final round.
5. ^ "Sponge Functions" (http://sponge.noekeon.org/) . Ecrypt Hash Workshop 2007. http://sponge.noekeon.org/.
6. ^ "On the Indifferentiability of the Sponge Construction" (http://sponge.noekeon.org/) . EuroCrypt 2008. http://sponge.noekeon.org/.
7. ^ NIST, Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition (http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf) , sections 5.1.2.1 (mentioning "tree mode"), 6.2 ("other features", mentioning authenticated encryption), and 7 (saying "extras" may be standardized in the future)

External links

- The Keccak web site (http://keccak.noekeon.org/)
- A Cryptol implementation of Keccak (http://plaintext.crypto.lo.gy/article/495/untwisted-a-cryptol-implementation-of-keccak-part-1)
- A VHDL source codes developed in the Cryptographic Engineering Research Group (CERG) at George Mason University (http://cryptography.gmu.edu/athena/index.php?id=source_codes)
- Erlang NIF implementation based on the NIST reference code (http://github.com/b/sha3)

Retrieved from "http://en.wikipedia.org/w/index.php?title=SHA-3&oldid=539034891"
Categories: Cryptographic hash functions | NIST hash function competition

The KECCAK sponge function family

Guido Bertoni[1], Joan Daemen[1], Michaël Peeters[2] and Gilles Van Assche[1]

[1]STMicroelectronics
[2]NXP Semiconductors

Pages

- Home
- News
- Files
- Specifications summary
- Tune KECCAK to your requirements
- Third-party cryptanalysis
- Our papers
- KECCAK Crunchy Crypto Collision and Pre-image Contest
- The KECCAK Team

Documents

- The KECCAK reference
- Files for the KECCAK reference
- The KECCAK SHA-3 submission
- KECCAK implementation overview
- Cryptographic sponge functions
- **all files...**

Notes

- Note on side-channel attacks and their countermeasures
- Note on zero-sum distinguishers of KECCAK-*f*
- Note on KECCAK parameters and usage
- On alignment in KECCAK

Software and other files

- Reference and optimized code in C
- Hardware implementation in VHDL
- Known-answer and Monte Carlo test results
- KECCAKTOOLS
- KECCAK in Python
- **all files...**

Figures

- KECCAK-*f* state [ODG] [PDF] [PNG]
- Pieces of state [ODG] [PDF] [PNG]
- Step θ [ODG] [PDF] [PNG]
- Step ρ [ODG] [PDF] [PNG]
- Step π [ODG] [PDF] [PNG]
- Step χ [ODG] [PDF] [PNG]

Links

- KeccakTools documentation
- RadioGatún
- Cryptographic sponges
- NIST SHA-3 hash function competition
- The SHA-3 zoo
- eBASH
- Animated Keccak (in German)

Specifications summary

Keccak (pronounced [kɛtʃak], like "ketchak") is a family of hash functions that has been submitted as candidate to NIST's hash algorithm competition (SHA-3). The text below is a quick description of Keccak using pseudo-code. In no way should this introductory text be considered as a formal and reference description of Keccak. Instead the goal here is to present Keccak with emphasis on readability and clarity. For a more formal description, the reader is invited to read the reference specifications in [1].

Structure of Keccak

Keccak is a family of hash functions that is based on the sponge construction, and hence is a sponge function family. In Keccak, the underlying function is a permutation chosen in a set of seven Keccak-*f* permutations, denoted Keccak-*f*[*b*], where $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ is the width of the permutation. The width of the permutation is also the width of the state in the sponge construction.

The state is organized as an array of 5×5 lanes, each of length $w \in \{1, 2, 4, 8, 16, 32, 64\}$ (b=25w). When implemented on a 64-bit processor, a lane of Keccak-*f*[1600] can be represented as a 64-bit CPU word.

We obtain the Keccak[*r*,*c*] sponge function, with parameters capacity *c* and bitrate *r*, if we apply the sponge construction to Keccak-*f*[*r*+*c*] and by applying a specific padding to the message input.

Pseudo-code description

We first start with the description of Keccak-*f* in the pseudo-code below. The number of rounds n_r depends on the permutation width, and is given by $n_r = 12+2l$, where $2^l = w$. This gives 24 rounds for Keccak-*f*[1600].

```
Keccak-f[b](A) {
  forall i in 0…$n_r$-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
}
```

In the pseudo-code above, the following conventions are in use. All the operations on the indices are done modulo 5. `A` denotes the complete permutation state array, and `A[x,y]` denotes a particular lane in that state. `B[x,y]`, `C[x]`, `D[x]` are intermediate variables. The constants `r[x,y]` are the rotation offsets (see Table 2), while `RC[i]` are the round constants (see Table 1). **`rot`**`(W,r)` is the usual bitwise cyclic shift operation, moving bit at position *i* into position *i*+`r` (modulo the lane size).

Then, we present the pseudo-code for the Keccak[*r*,*c*] sponge function, with parameters capacity *c* and bitrate *r*. The description below is restricted to the case of messages that span a whole number of bytes. For messages with a number of bits not dividable by 8, we refer to the specifications [1] for more details. Also, we assume for simplicity that *r* is a multiple of the lane size; this is the case for the SHA-3 candidate parameters in [2].

```
Keccak[r,c](M) {
  Initialization and padding
  S[x,y] = 0,                               forall (x,y) in (0…4,0…4)
  P = M || 0x01 || 0x00 || … || 0x00
  P = P xor (0x00 || … || 0x00 || 0x80)

  Absorbing phase
  forall block $P_i$ in P
    S[x,y] = S[x,y] xor $P_i$[x+5*y],       forall (x,y) such that x+5*y < r/w
    S = Keccak-f[r+c](S)

  Squeezing phase
  Z = empty string
  while output is requested
    Z = Z || S[x,y],                        forall (x,y) such that x+5*y < r/w
    S = Keccak-f[r+c](S)

  return Z
}
```

In the pseudo-code above, `S` denotes the state as an array of lanes. The padded message `P` is organised as an array of blocks P_i, themselves organized as arrays of lanes. The `||` operator denotes the usual byte string concatenation.

Round constants

The round constants `RC[i]` are given in the table below for the maximum lane size 64. For smaller sizes, they are simply truncated. The formula can be found in [1].

Table 1: The round constants `RC[i]`

RC[0]	0x0000000000000001	**RC[12]**	0x000000008000808B
RC[1]	0x0000000000008082	**RC[13]**	0x800000000000008B
RC[2]	0x800000000000808A	**RC[14]**	0x8000000000008089
RC[3]	0x8000000080008000	**RC[15]**	0x8000000000008003
RC[4]	0x000000000000808B	**RC[16]**	0x8000000000008002
RC[5]	0x0000000080000001	**RC[17]**	0x8000000000000080

RC[6]	0x8000000080008081	**RC[18]**	0x000000000000800A
RC[7]	0x8000000000008009	**RC[19]**	0x800000008000000A
RC[8]	0x000000000000008A	**RC[20]**	0x8000000080008081
RC[9]	0x0000000000000088	**RC[21]**	0x8000000000008080
RC[10]	0x0000000080008009	**RC[22]**	0x0000000080000001
RC[11]	0x000000008000000A	**RC[23]**	0x8000000080008008

Rotation offsets

The rotation offsets `r[x,y]` are given in the table below. The formula can be found in [1].

Table 2: the rotation offsets

	x = 3	**x = 4**	**x = 0**	**x = 1**	**x = 2**
y = 2	25	39	3	10	43
y = 1	55	20	36	44	6
y = 0	28	27	0	1	62
y = 4	56	14	18	2	61
y = 3	21	8	41	45	15

References

[1] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, The Keccak reference, 2011

[2] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, The Keccak SHA-3 submission, 2011