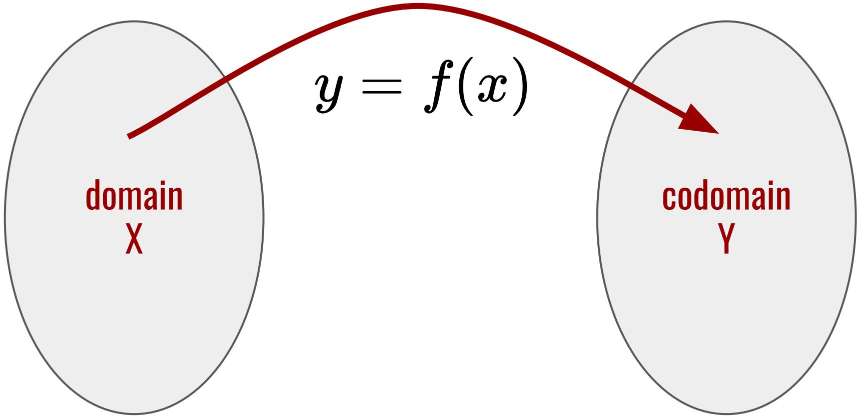
Hash Functions

Computer and Network Security

Emilio Coppa

Hash Function



y is often called:

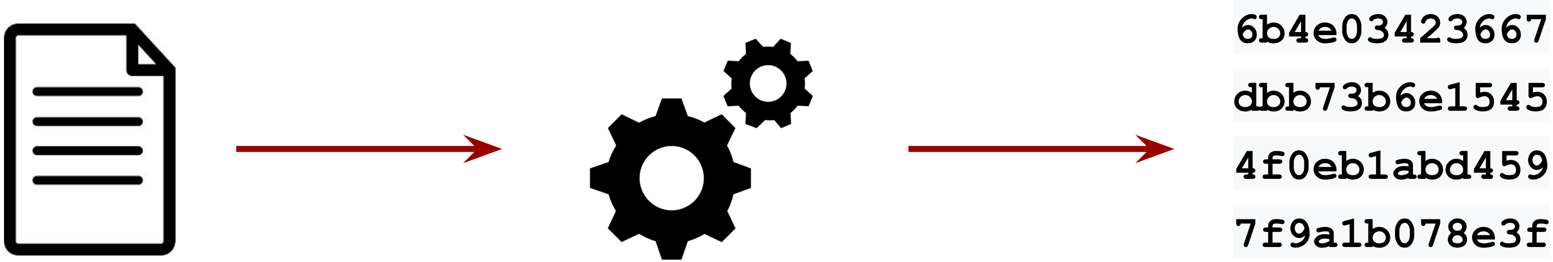
- hash value
- hash code
- digest

- Goal. map large domains into smaller codomains: |X| >> |Y|
- HF are used extensively in data structures, e.g., hash tables (example: f(x) = ax + b mod p)
- Crucial to minimize collisions (collisions in hash tables degrades performance)

Why we care for hash functions?

For instance, we could use them for data integrity...

- we want a compact value to represent a large amount of data
- changing even one bit in the data should make the value change (unpredictably!)



6b4e03423667
dbb73b6e1545
4f0eb1abd459
7f9a1b078e3f

Document

Hash Function

Hash Value

Cryptographic Hash Function

Properties of a cryptographic hash function:

1. Arbitrary input length: we would like to process even large files (e.g., an entire hard disk). Notice that encryption schemes often impose restriction on the input length.

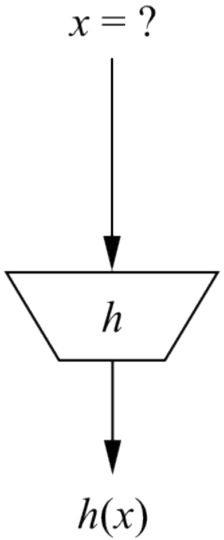
2. Fixed and short output length: in several application context, we would like to have a compact output value given (even very large) input. Notice that encryption often have a output length that is equal or larger than the input length.

3. Efficiency. Fast even when processing large inputs. Notice that encryption schemes can be very slow on large files.

Security Requirements of Cryptographic Hash Functions

1. Preimage resistance (or one wayness):

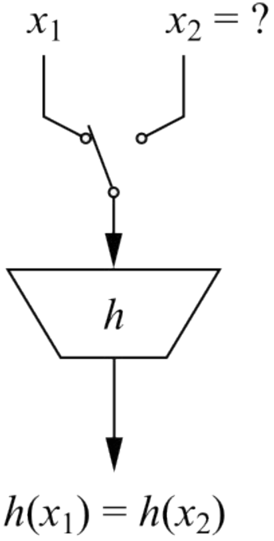
   For a given output y = h(x), it is computationally infeasible to find any input x such that h(x) = y, i.e., h(x) is one-way.



Security Requirements of Cryptographic Hash Functions (2)

2. Second preimage resistance (or weak collision resistance):

Given x_1, and thus $h(x_1)$, it is computationally infeasible to find any x_2 such that $h(x_1) = h(x_2)$.

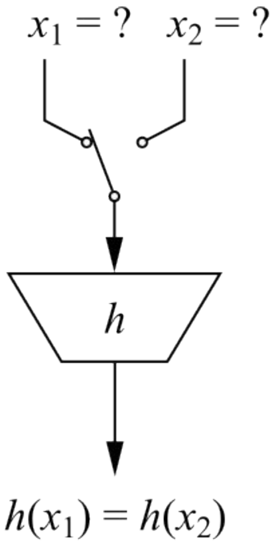


Security Requirements of Cryptographic Hash Functions (3)

3. **Collision resistance**
   (or **strong collision resistance**):

It is computationally infeasible to find any pairs (x_1, x_2) where $x_1 \mathrel{!}= x_2$ such that $h(x_1) = h(x_2)$.

Different from prop 2 because here attacker can choose x1 and x2



Hash functions without collisions?

Since by design an hash function maps a large domain into a smaller codomain then there is no possible way of avoiding collisions or implementing an hash function with no collision:

1. Dirichlet's drawer principle: "if you have 10 socks and only 9 drawers, then at least one drawer will contain more than one sock".

2. Pigeonhole principle: "if you have 20 pigeons but only 19 boxes, then at least one box will contain at least two pigeons".

However, a cryptographic hash function requires that should be hard to find collisions.

Second preimage attack: brute force

Given (x, h(x)) an adversary can always perform a brute force attack to find (x’, h(x’)) such that x != x’ and h(x) = h(x’). If he does not exploit any property from the specific hash function h(), then adversary can try to compute h(x’) for any possible x’ until a collision is found.

The expected attack complexity is:

2^n attempts where n is the number of bits for the output

[illegible]

Hence n must large enough to make the attack infeasible, e.g., n = 80.

Collision resistance attack

Now the attacker has two degrees of freedom to find a collision. This makes even a “simple” brute force attack much easier than what we may expect.

Let’s reason on the attack complexity using a simplified formulation of the problem “if we have a party, how many people should I invite to have two people with the same birthday?” This is also called the Birthday Paradox.

Remark. In second preimage, the simplified problem would be “if we have a party, how many people should I invite to have one people with birthday on a specific date XYZ?”. In this case, on average, we expect that the answer is 365 people.

Birthday Paradox

$$Pr(\text{collision with 2 people}) = \frac{1}{365}$$

$$Pr(\text{no collision with 2 people}) = 1 - \frac{1}{365}$$

$$Pr(\text{no collision with 3 people}) = (1 - \frac{1}{365})(1 - \frac{2}{365})$$

$$\vdots$$

$$Pr(\text{no collision with t people}) = \underline{\prod_{i=1}^{t-1}(1 - \frac{i}{365})}$$

How many people we need for a probability of at least 0.5 of one collision?

$$\underline{Pr(\text{one collision with 23 people})} = 1 - \prod_{i=1}^{22}(1 - \frac{i}{365}) \approx \boxed{0.5}$$

What is the relation between 23 and 365? The root square of 365 is “almost” 23.

Birthday Paradox (2)

Generalizing over hash functions with n-bit output:

$$\underline{P(\text{no collision})} = \left(1 - \frac{1}{2^n}\right)\left(1 - \frac{2}{2^n}\right)\cdots\left(1 - \frac{t-1}{2^n}\right)$$

$$= \prod_{i=1}^{t-1}\left(1 - \frac{i}{2^n}\right)$$

After some simplifications (see 11.2.3 of “Understanding Cryptography”) we get that for 0.5 probability of a collision:

$$\boxed{t \approx 2^{(n+1)/2}\sqrt{\ln\left(\frac{1}{1-\lambda}\right)} \qquad \lambda = 1 - Pr(\text{no collision})}$$

Birthday Paradox (3)

In practice, we can see that the attack complexity is roughly the square root of the output space.

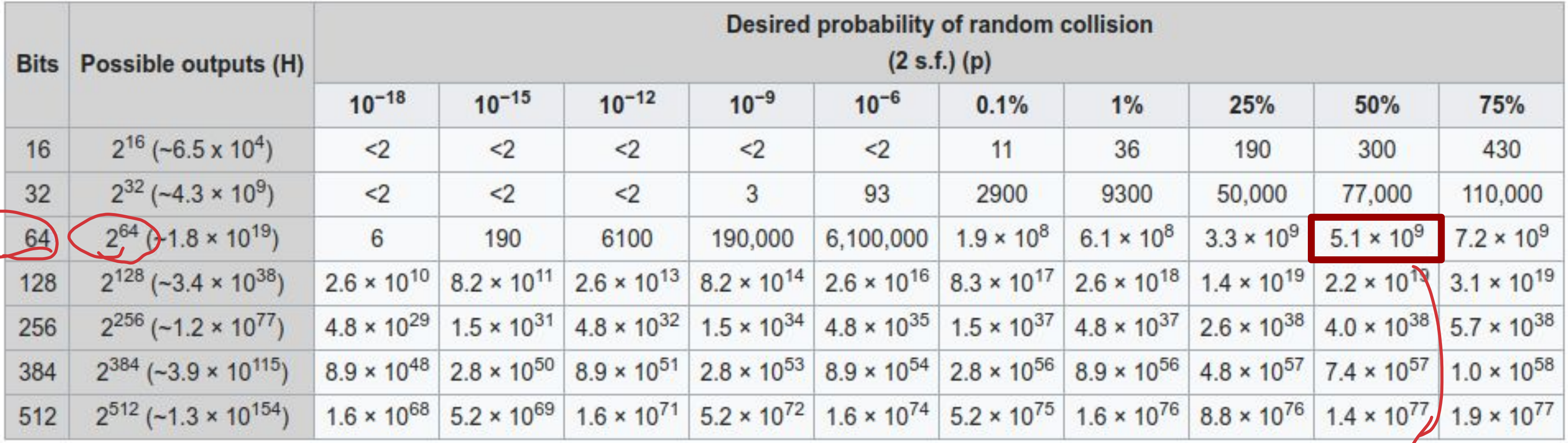
E.g., output length is n=80 bits, output space is 2^{80} , the expected number of attempts to find a collision with 0.5 probability is:

$$\sqrt[2]{2^{80}} = 2^{40}$$

Hence, if want to have an expected complexity of at least 2^{80}, we need actually an output space of at least 2^{160}!

→ P 0.5 to discover collision.

Birthday Bound



Bits	Possible outputs (H)	Desired probability of random collision (2 s.f.) (p)									
		10^{-18}	10^{-15}	10^{-12}	10^{-9}	10^{-6}	0.1%	1%	25%	50%	75%
16	2^{16} (~6.5×10^{4})	<2	<2	<2	<2	<2	11	36	190	300	430
32	2^{32} (~4.3×10^{9})	<2	<2	<2	3	93	2900	9300	50,000	77,000	110,000
64	2^{64} (~1.8×10^{19})	6	190	6100	190,000	6,100,000	1.9×10^{8}	6.1×10^{8}	3.3×10^{9}	$\boxed{5.1 \times 10^{9}}$	7.2×10^{9}
128	2^{128} (~3.4×10^{38})	2.6×10^{10}	8.2×10^{11}	2.6×10^{13}	8.2×10^{14}	2.6×10^{16}	8.3×10^{17}	2.6×10^{18}	1.4×10^{19}	2.2×10^{19}	3.1×10^{19}
256	2^{256} (~1.2×10^{77})	4.8×10^{29}	1.5×10^{31}	4.8×10^{32}	1.5×10^{34}	4.8×10^{35}	1.5×10^{37}	4.8×10^{37}	2.6×10^{38}	4.0×10^{38}	5.7×10^{38}
384	2^{384} (~3.9×10^{115})	8.9×10^{48}	2.8×10^{50}	8.9×10^{51}	2.8×10^{53}	8.9×10^{54}	2.8×10^{56}	8.9×10^{56}	4.8×10^{57}	7.4×10^{57}	1.0×10^{58}
512	2^{512} (~1.3×10^{154})	1.6×10^{68}	5.2×10^{69}	1.6×10^{71}	5.2×10^{72}	1.6×10^{74}	5.2×10^{75}	1.6×10^{76}	8.8×10^{76}	1.4×10^{77}	1.9×10^{77}

Near to 2^{32}

$= \sqrt{2^{64}}$

Result for n=64 and p=0.5 is called birthday bound

Strong collision implies weak collision

Proof. we show that $\neg\text{weak} \Rightarrow \neg\text{strong}$

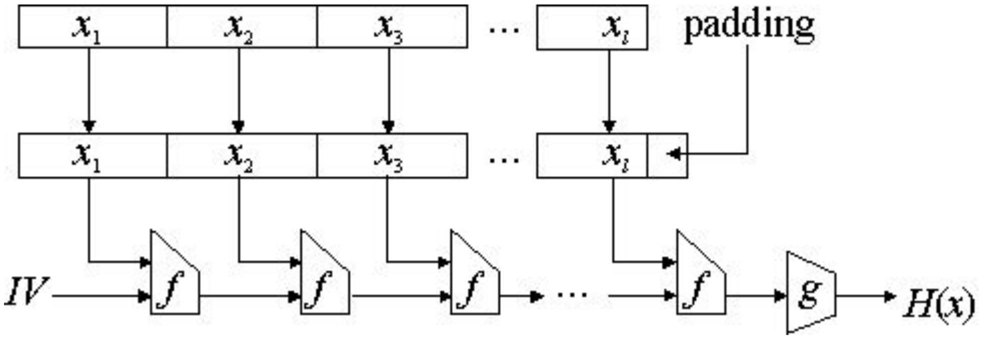
Idea. Pick randomly x, compute h(x), find x' such that x != x' and h(x)=h(x'). (a, b) shows that the hash function is not collision resistant.

More formally:

- suppose there is polynomial algorithm A_h: $A_h(x) = x'$ such that $h(x) = h(x')$
- construct a polynomial algorithm $B_h() = (x, x')$: randomly picks x and returns $(x, A_h(x))$

Merkle-Damgård construction

Several cryptographic hash function takes as input a fixed block size (e.g., 256 bits). To fulfill the requirement on arbitrary input length, we can use the Merkle-Damgård Construction (MDC):



IMPLIES

- IV is usually constant
- if hash function f is collision resistant, then its MDC extension is collision resistant

Real-World Cryptographic Hash Functions

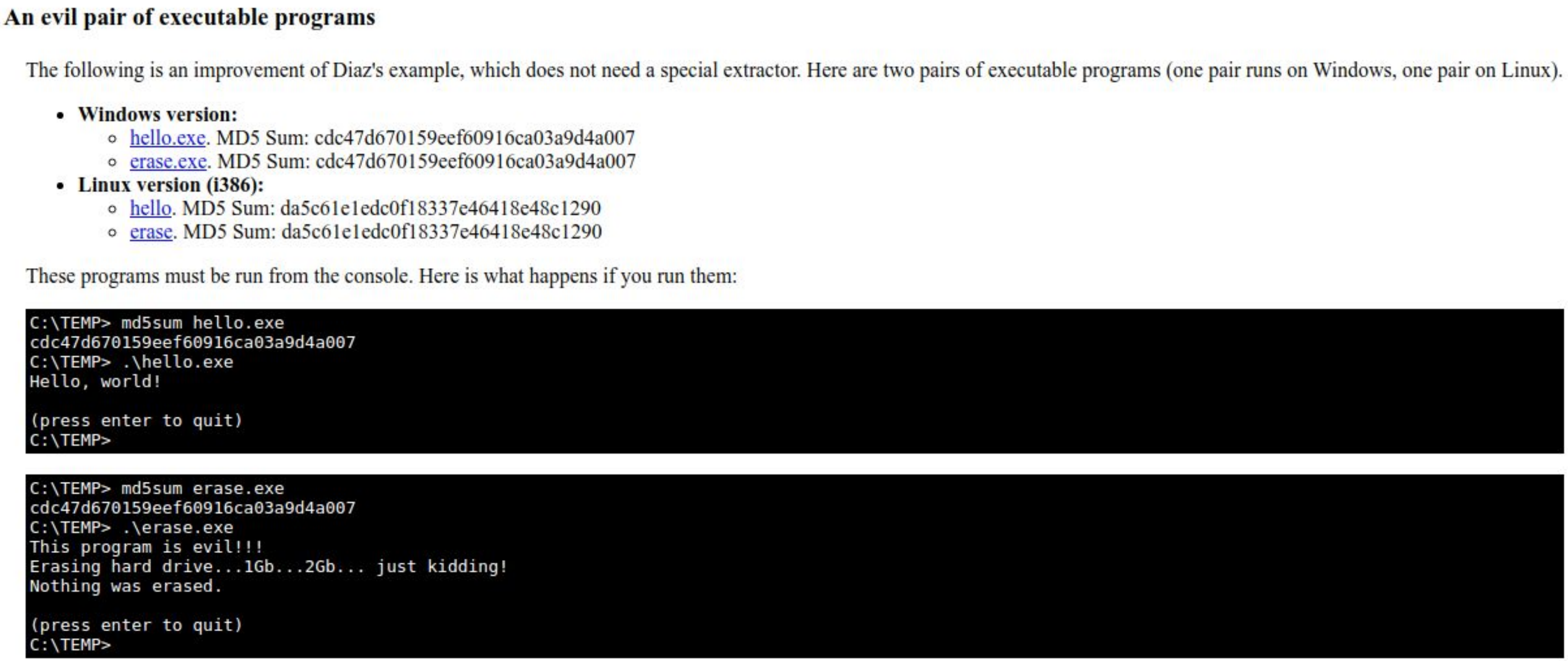
- MD (Message Digest) family:
  - MD4 by Ronald Rivest (1990): first collision attack in 1995. In 2007, an attack can generate collisions in less than 2 MD4 hash operations. A theoretical preimage attack also exists.

  - MD5 by Ronald Rivest (1992): collision attack in seconds on a 2.6 GHz Pentium 4 processor (complexity of 2^24.1). A chosen-prefix collision attack that can produce a collision for two inputs with specified prefixes within seconds, using off-the-shelf computing hardware (complexity 2^39). Pre-image attack still hard (2009: $2^{123.4}$)

- SHA (Secure Hash Algorithm) family:
  - SHA-1 by NSA (1995): the most popular cryptographic hash function, inspired by MD algorithms. Since 2005 considered not fully secure, Google was able to generate a collision in 2017. As of 2020, chosen-prefix attacks against SHA-1 are now practical.

  - SHA-3: current solution part of a standard, proposed in 2015 in response to an open call from NIST (similar to AES).

MD5 collision demo (code and discussion)



An evil pair of executable programs

The following is an improvement of Diaz's example, which does not need a special extractor. Here are two pairs of executable programs (one pair runs on Windows, one pair on Linux).

- **Windows version:**
  - hello.exe. MD5 Sum: cdc47d670159eef60916ca03a9d4a007
  - erase.exe. MD5 Sum: cdc47d670159eef60916ca03a9d4a007
- **Linux version (i386):**
  - hello. MD5 Sum: da5c61e1edc0f18337e46418e48c1290
  - erase. MD5 Sum: da5c61e1edc0f18337e46418e48c1290

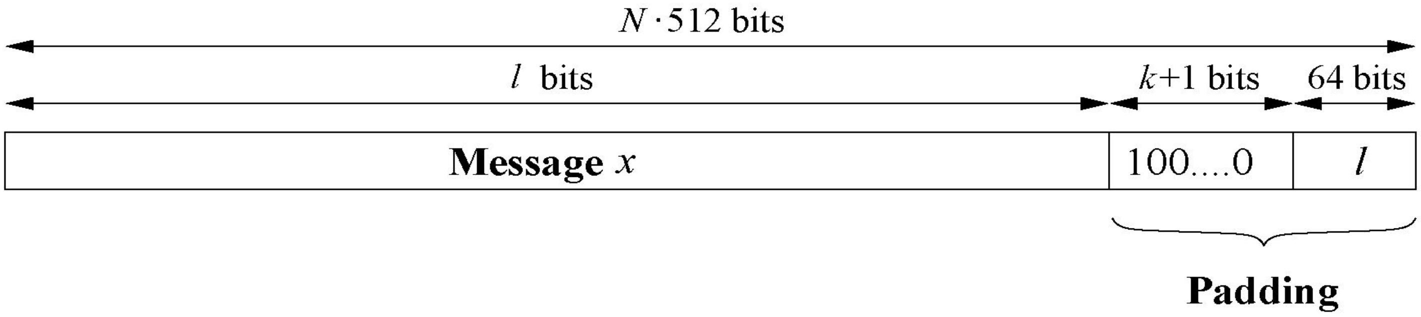
These programs must be run from the console. Here is what happens if you run them:

```
C:\TEMP> md5sum hello.exe
cdc47d670159eef60916ca03a9d4a007
C:\TEMP> .\hello.exe
Hello, world!

(press enter to quit)
C:\TEMP>
```

```
C:\TEMP> md5sum erase.exe
cdc47d670159eef60916ca03a9d4a007
C:\TEMP> .\erase.exe
This program is evil!!!
Erasing hard drive...1Gb...2Gb... just kidding!
Nothing was erased.

(press enter to quit)
C:\TEMP>
```

SHA-1

- input length: up to 2^{64}
- block size: 512 bits
- output length: 160 bits
- original message is padded with (10...00) and input length (l) to reach a multiple of block size:



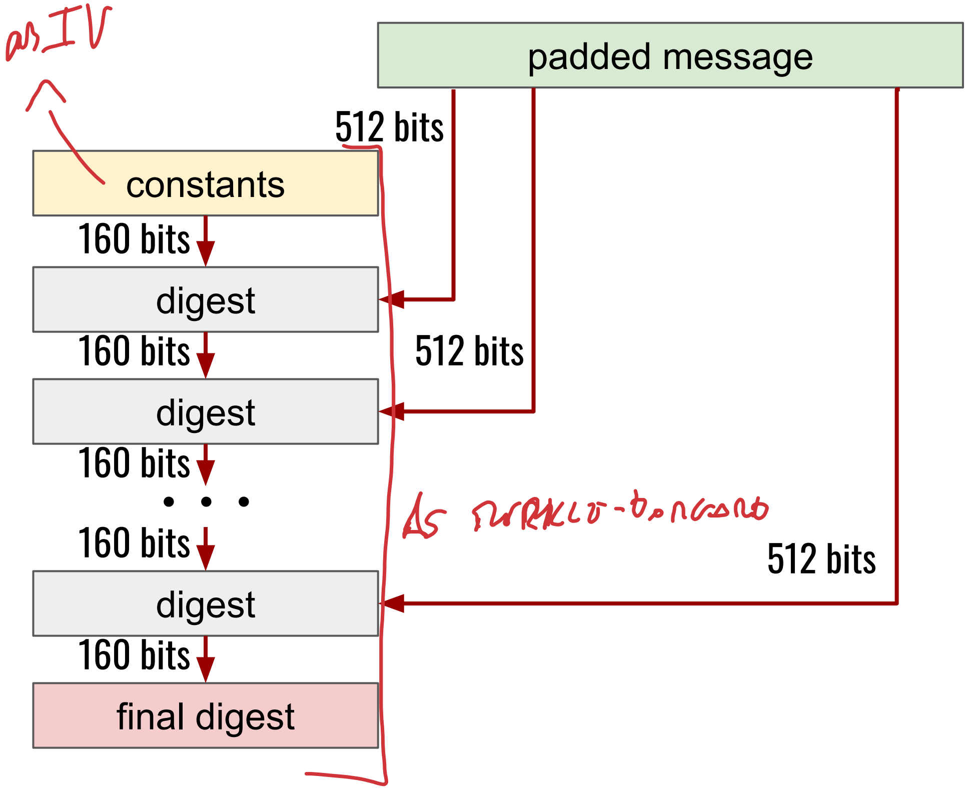
- the padded message is then split in blocks of 512 bits

SHA-1 (2)

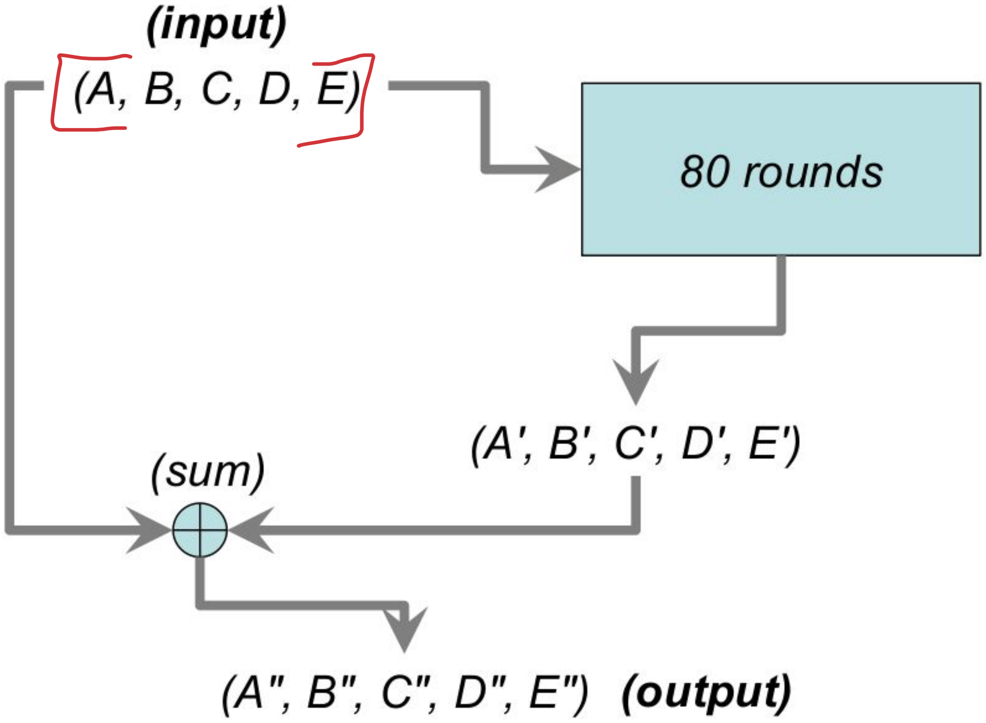
It is based on Merkle-Dåmgard construction:

- each digest stage is composed by 80 rounds

- initial digest is constant and composed by five 32 bit constants {A, B, C, D, E}, which are called registers and are updated by each digest computation



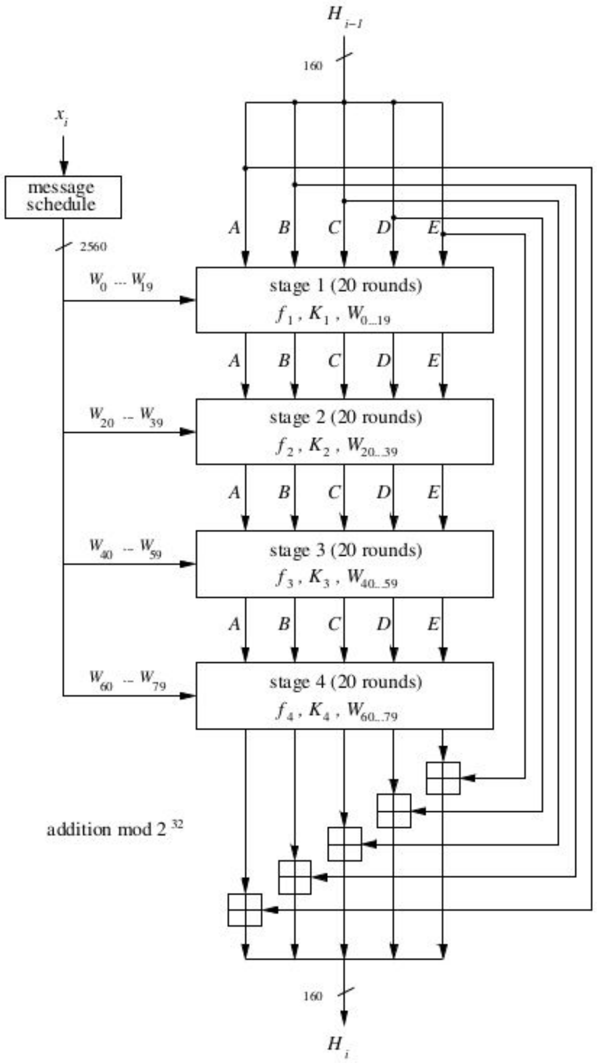
SHA-1: digest computations



Split in four stages of 20 rounds each

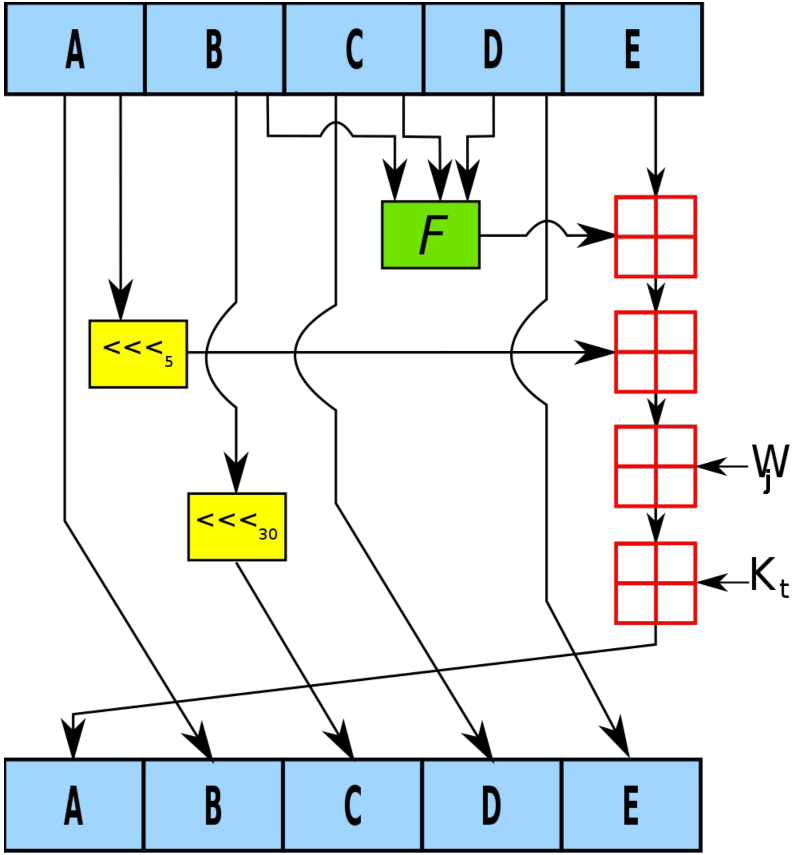
SHA-1: digest computation

- 4 stages
- t is stage number
- f(t, B, C, D): non-linear function
- W_j: 32 bit word obtained by expanding original 16 words (512 bit block) into 80 words (using shift and xor)



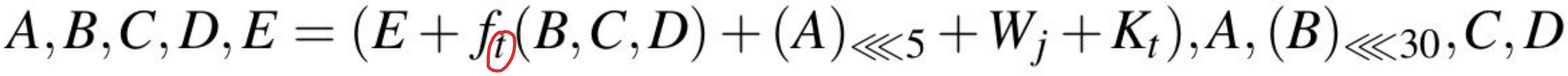
SHA-1: one round

- t is stage number
- f(t, B, C, D): non-linear function
- W_j: 32 bit word obtained by expanding original 16 words (512 bit block) into 80 words (using shift and xor)
- <<<: left bit rotation
- K_t: round constant

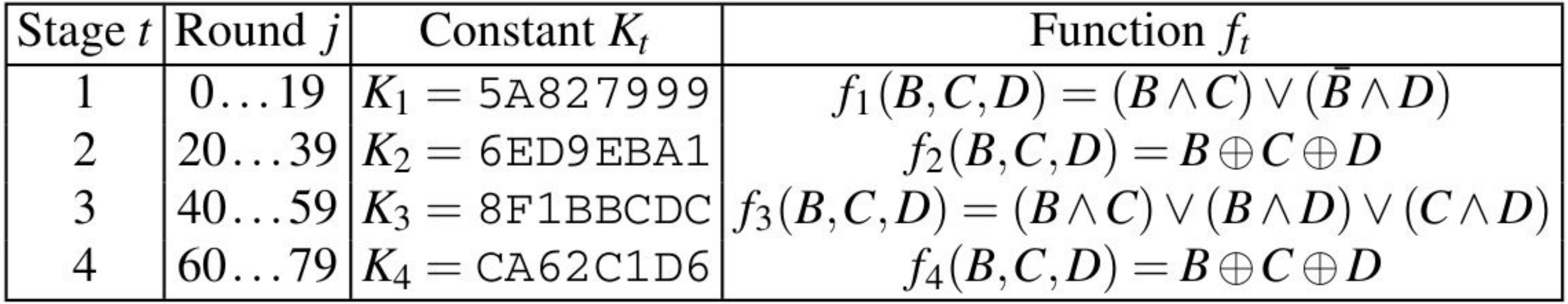


$$A,B,C,D,E = (E + f_t(B,C,D) + (A)_{\lll 5} + W_j + K_t), A, (B)_{\lll 30}, C, D$$

SHA-1: round constants and round functions



$$A,B,C,D,E = (E + f_t(B,C,D) + (A)_{\lll 5} + W_j + K_t), A, (B)_{\lll 30}, C, D$$



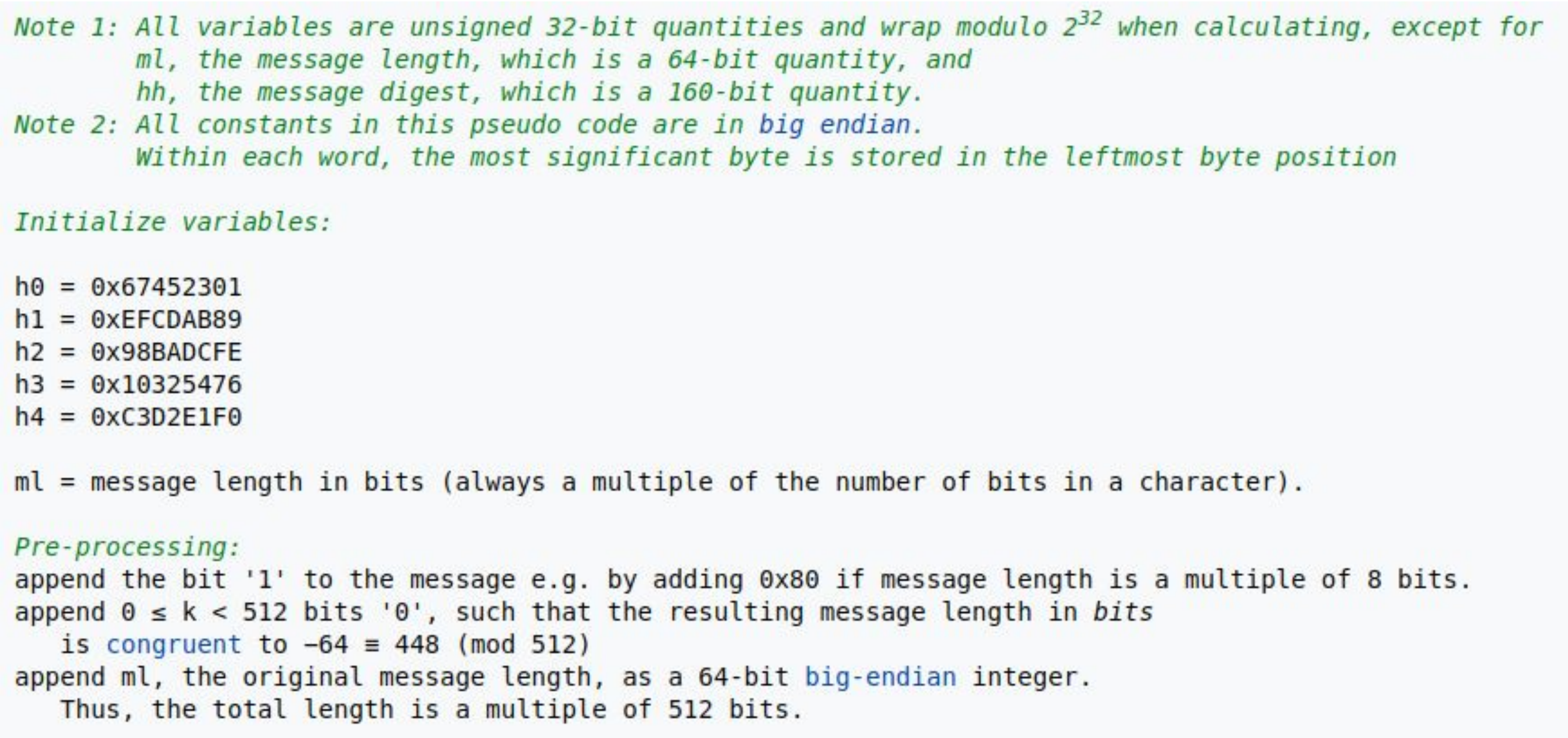
Stage t	Round j	Constant K_t	Function f_t
1	$0 \dots 19$	$K_1 = \texttt{5A827999}$	$f_1(B,C,D) = (B \wedge C) \vee (\bar{B} \wedge D)$
2	$20 \dots 39$	$K_2 = \texttt{6ED9EBA1}$	$f_2(B,C,D) = B \oplus C \oplus D$
3	$40 \dots 59$	$K_3 = \texttt{8F1BBCDC}$	$f_3(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
4	$60 \dots 79$	$K_4 = \texttt{CA62C1D6}$	$f_4(B,C,D) = B \oplus C \oplus D$

SHA-1: message schedule

$$W_j = \begin{cases} x_i^{(j)} & 0 \leq j \leq 15 \\ (W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3}) \lll 1 & 16 \leq j \leq 79, \end{cases}$$

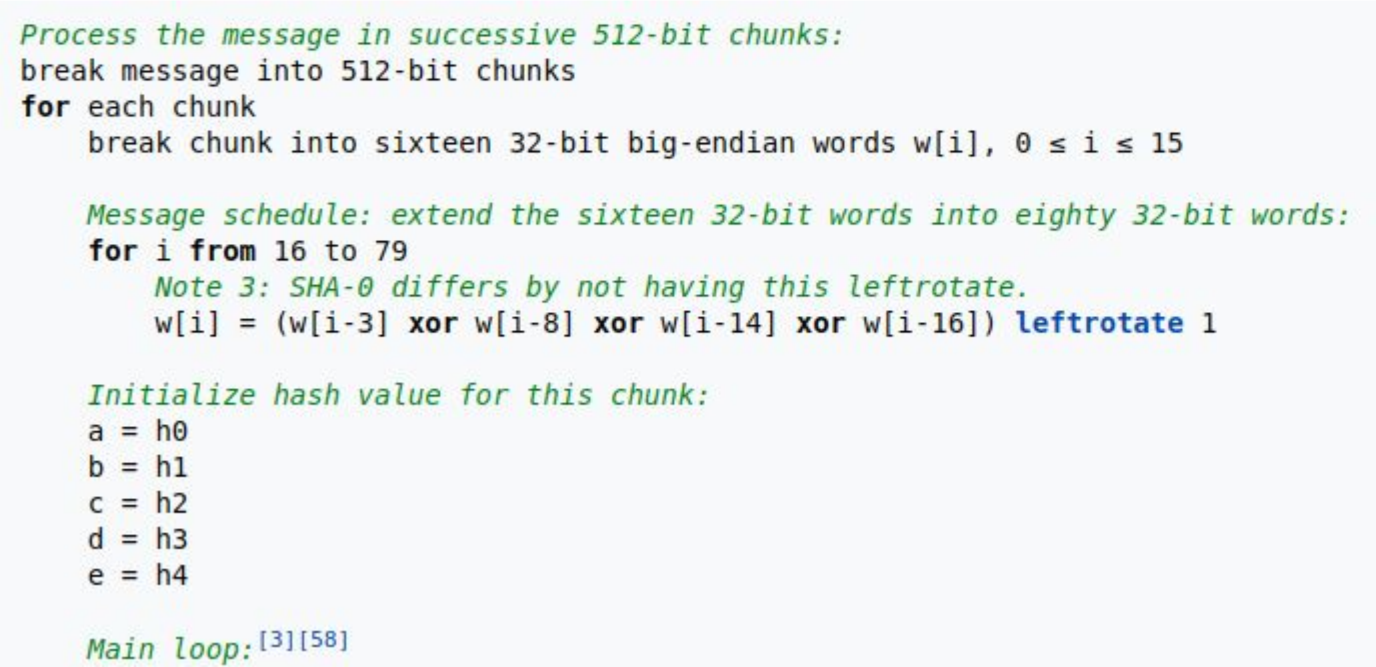
where $x_i^{(j)}$ is the j-th 32-bit word from the 512-bit block x_i

SHA-1: pseudocode (from wikipedia)



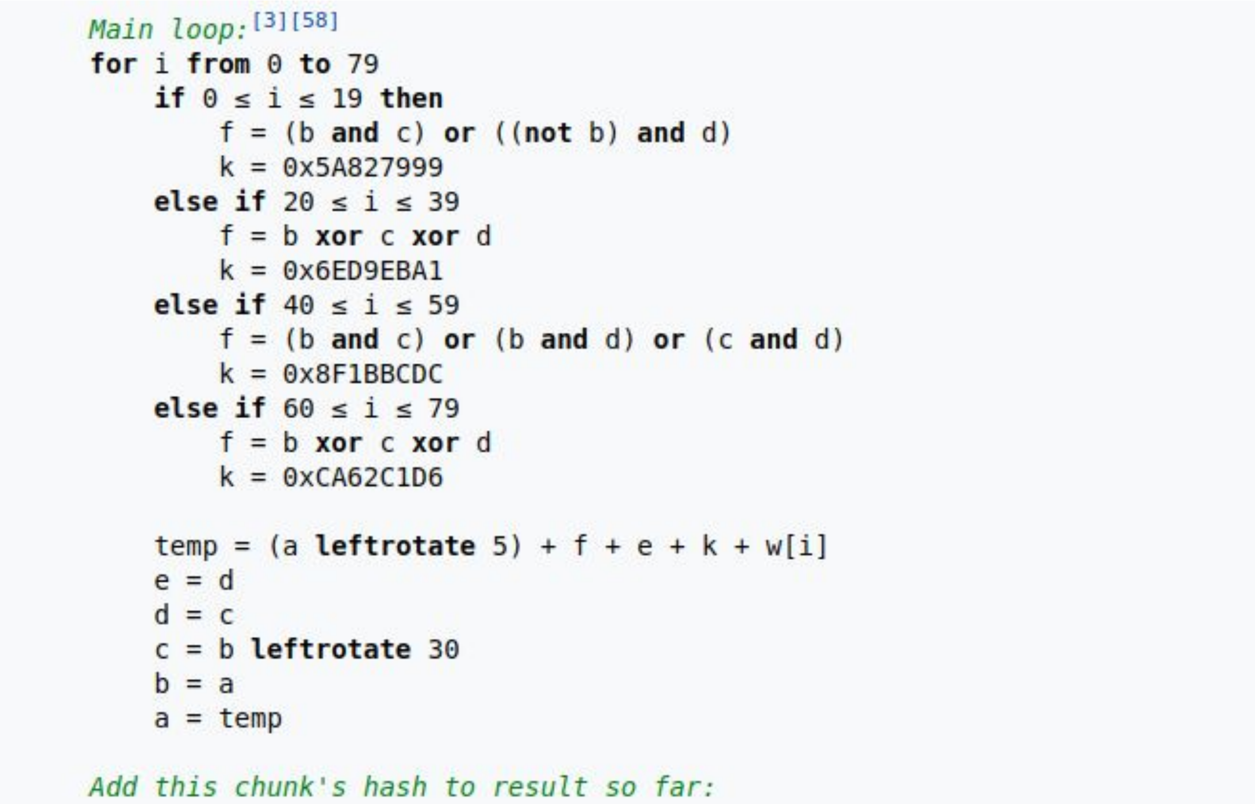
```
Note 1: All variables are unsigned 32-bit quantities and wrap modulo $2^{32}$ when calculating, except for
        ml, the message length, which is a 64-bit quantity, and
        hh, the message digest, which is a 160-bit quantity.
Note 2: All constants in this pseudo code are in big endian.
        Within each word, the most significant byte is stored in the leftmost byte position

Initialize variables:

h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

ml = message length in bits (always a multiple of the number of bits in a character).

Pre-processing:
append the bit '1' to the message e.g. by adding 0x80 if message length is a multiple of 8 bits.
append 0 ≤ k < 512 bits '0', such that the resulting message length in bits
   is congruent to −64 ≡ 448 (mod 512)
append ml, the original message length, as a 64-bit big-endian integer.
   Thus, the total length is a multiple of 512 bits.
```

SHA-1: pseudocode (2)



```
Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

    Message schedule: extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
        Note 3: SHA-0 differs by not having this leftrotate.
        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1

    Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    Main loop:[3][58]
```

SHA-1: pseudocode (3)

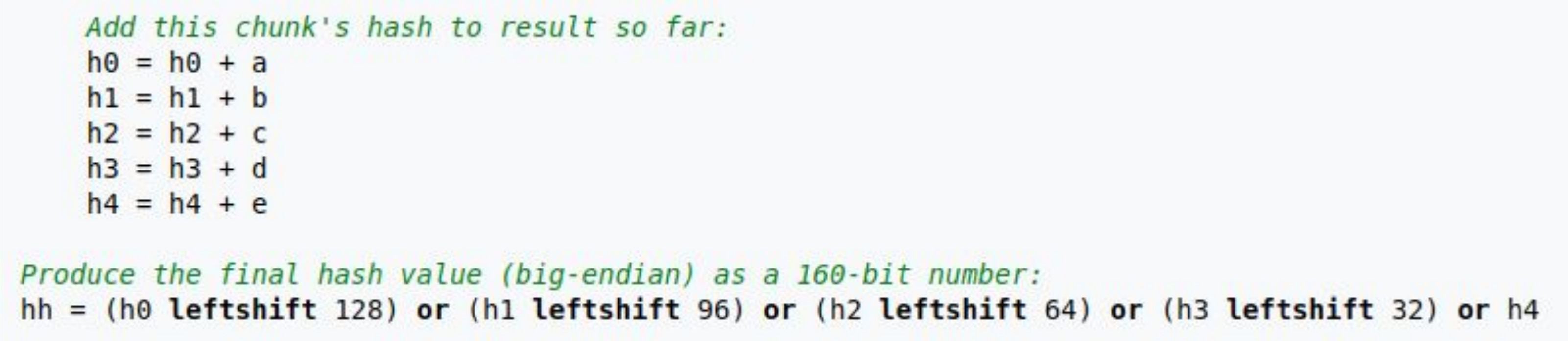


Main loop:[3][58]

```
for i from 0 to 79
    if 0 ≤ i ≤ 19 then
        f = (b and c) or ((not b) and d)
        k = 0x5A827999
    else if 20 ≤ i ≤ 39
        f = b xor c xor d
        k = 0x6ED9EBA1
    else if 40 ≤ i ≤ 59
        f = (b and c) or (b and d) or (c and d)
        k = 0x8F1BBCDC
    else if 60 ≤ i ≤ 79
        f = b xor c xor d
        k = 0xCA62C1D6

    temp = (a leftrotate 5) + f + e + k + w[i]
    e = d
    d = c
    c = b leftrotate 30
    b = a
    a = temp
```

Add this chunk's hash to result so far:

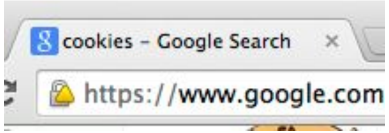
SHA-1: pseudocode (4)



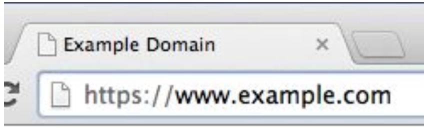
```
    Add this chunk's hash to result so far:
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

Produce the final hash value (big-endian) as a 160-bit number:
hh = (h0 leftshift 128) or (h1 leftshift 96) or (h2 leftshift 64) or (h3 leftshift 32) or h4
```

Dismissing SHA-1 in Chrome

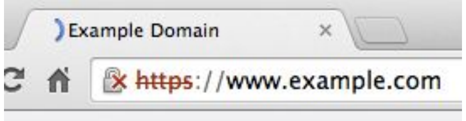
- Chrome 39 (Branch point 26 September 2014) – secure, but with minor errors:

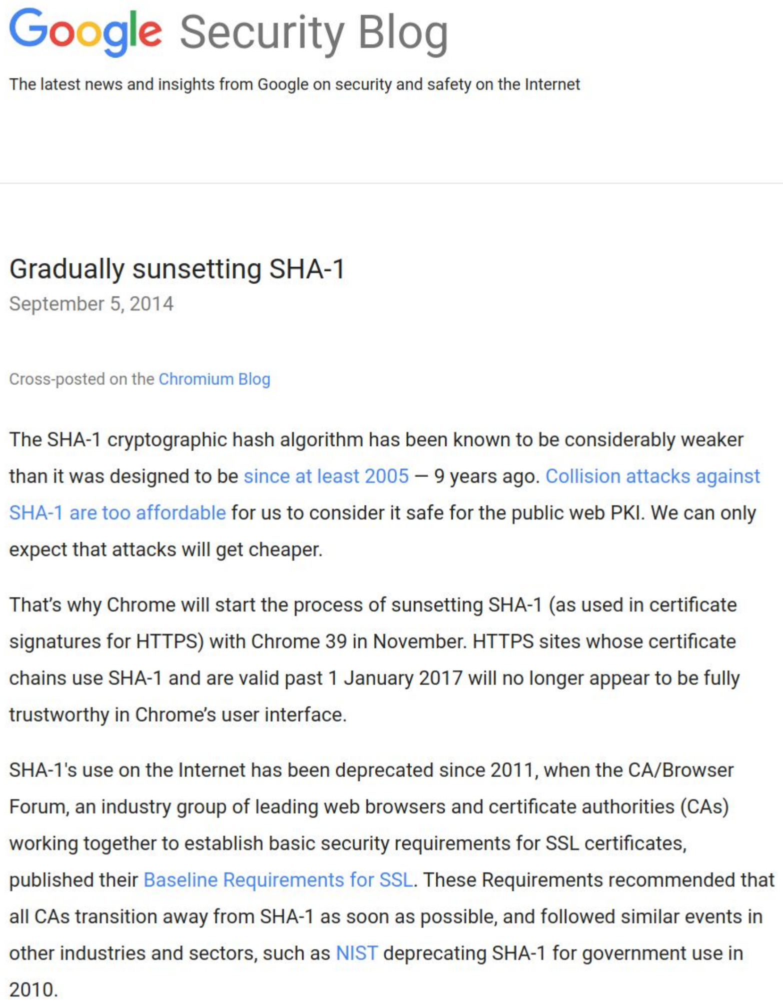


- Chrome 40 (Branch point 7 November 2014; Stable after holiday season) – neutral, lacking security:



- Chrome 41 (Branch point in Q1 2015) – affirmatively insecure:





Google Security Blog

The latest news and insights from Google on security and safety on the Internet

Gradually sunsetting SHA-1

September 5, 2014

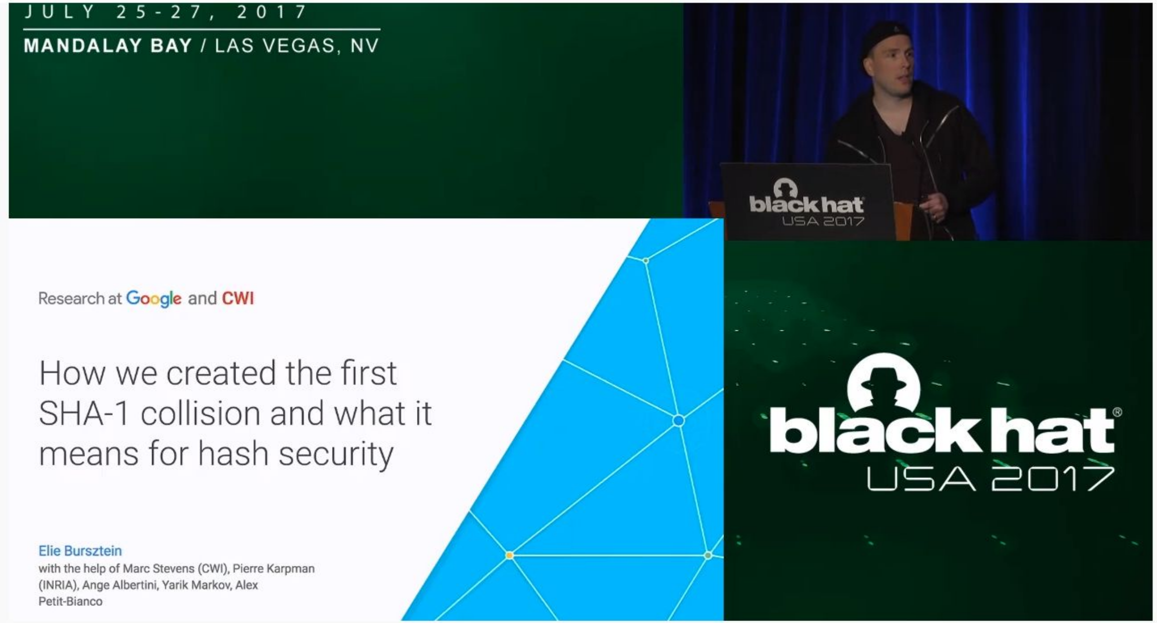
Cross-posted on the Chromium Blog

The SHA-1 cryptographic hash algorithm has been known to be considerably weaker than it was designed to be since at least 2005 — 9 years ago. Collision attacks against SHA-1 are too affordable for us to consider it safe for the public web PKI. We can only expect that attacks will get cheaper.

That's why Chrome will start the process of sunsetting SHA-1 (as used in certificate signatures for HTTPS) with Chrome 39 in November. HTTPS sites whose certificate chains use SHA-1 and are valid past 1 January 2017 will no longer appear to be fully trustworthy in Chrome's user interface.

SHA-1's use on the Internet has been deprecated since 2011, when the CA/Browser Forum, an industry group of leading web browsers and certificate authorities (CAs) working together to establish basic security requirements for SSL certificates, published their Baseline Requirements for SSL. These Requirements recommended that all CAs transition away from SHA-1 as soon as possible, and followed similar events in other industries and sectors, such as NIST deprecating SHA-1 for government use in 2010.

Google: How We Created the First SHA-1 Collision and What it Means for Hash Security



JULY 25-27, 2017

MANDALAY BAY / LAS VEGAS, NV

Research at Google and CWI

How we created the first SHA-1 collision and what it means for hash security

Elie Bursztein
with the help of Marc Stevens (CWI), Pierre Karpman (INRIA), Ange Albertini, Yarik Markov, Alex Petit-Bianco

Talk by Brusztein @ Black Hat USA 2017

SHA-3

- Original name is Keccak Algorithm (the name should be inspired by the “kecak” dance)

- Designed by Guido Bertoni and Joan Daemen (author of AES), Michael Peeters, and Gilles Van Assche. Reference implementation.

- Selected as the new standard by NIST: open call in 2007, 51 submission in 2008, 14 finalists in 2009, announced as the winner in 2012, SHA-3 standard in 2015.

- Very different from MD/SHA-1, based on the idea of “sponge” construction:
  - absorbing phase: input block x_i is read-in and processed
  - squeezing phase: output is produced

SHA-3

- Output length:
  - 224 (same resistance as 3DES when using birthday attack),
  - 256
  - 384 (same resistance as 192 AES when using birthday attack)
  - 512

- Keccak state size b, also called bus, is:

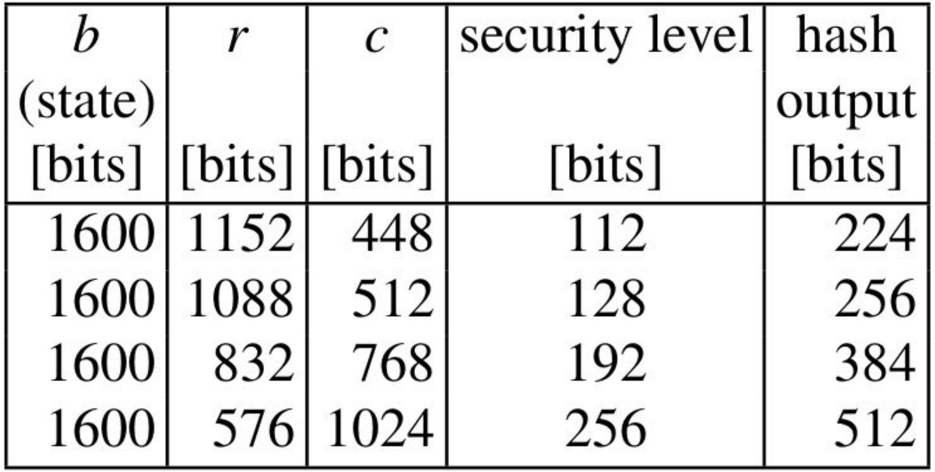
$$b = 25 \cdot 2^l \text{ where } l \in \{0, 1, 2, \ldots, 6\}$$
$$b = \{25, 50, 100, 200, 400, 800, 1600\}$$

- SHA-3: only use b = 1600

SHA-3

- Number of rounds: $n_r = 12 + 2l$
  In SHA-3: $n_r = 24$

- block size r, capacity c = b - r

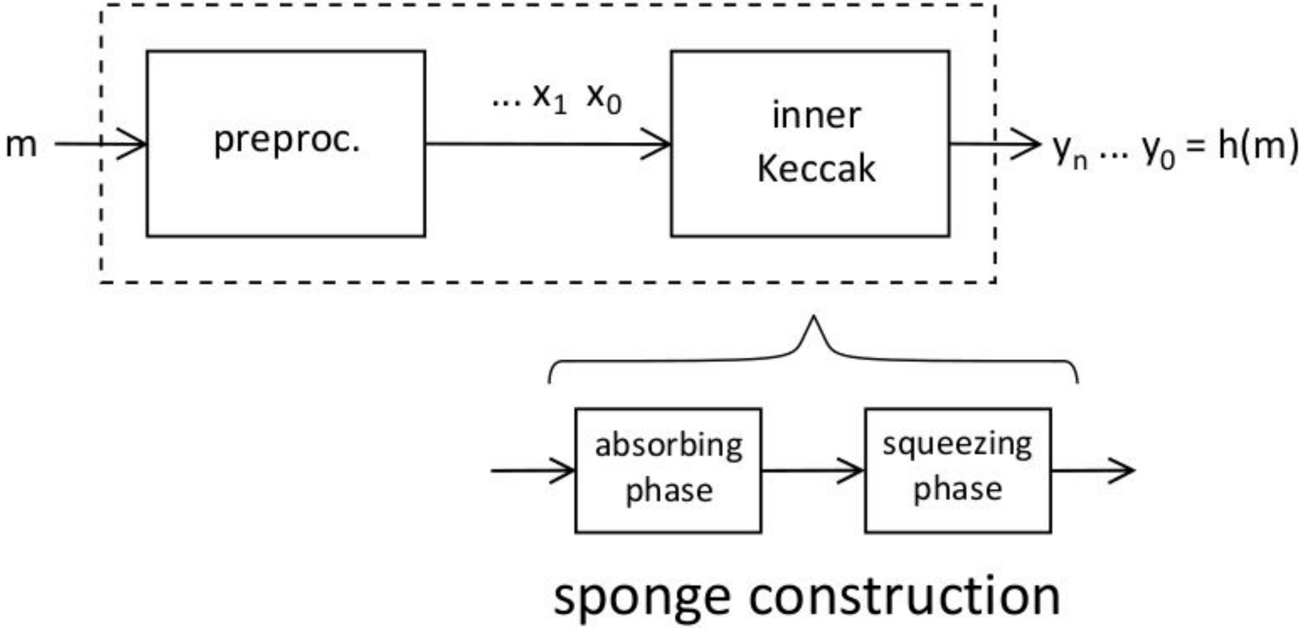


b (state) [bits]	r [bits]	c [bits]	security level [bits]	hash output [bits]
1600	1152	448	112	224
1600	1088	512	128	256
1600	832	768	192	384
1600	576	1024	256	512

Chapter 11 of Understanding Cryptography by Christof Paar and Jan Pelzl

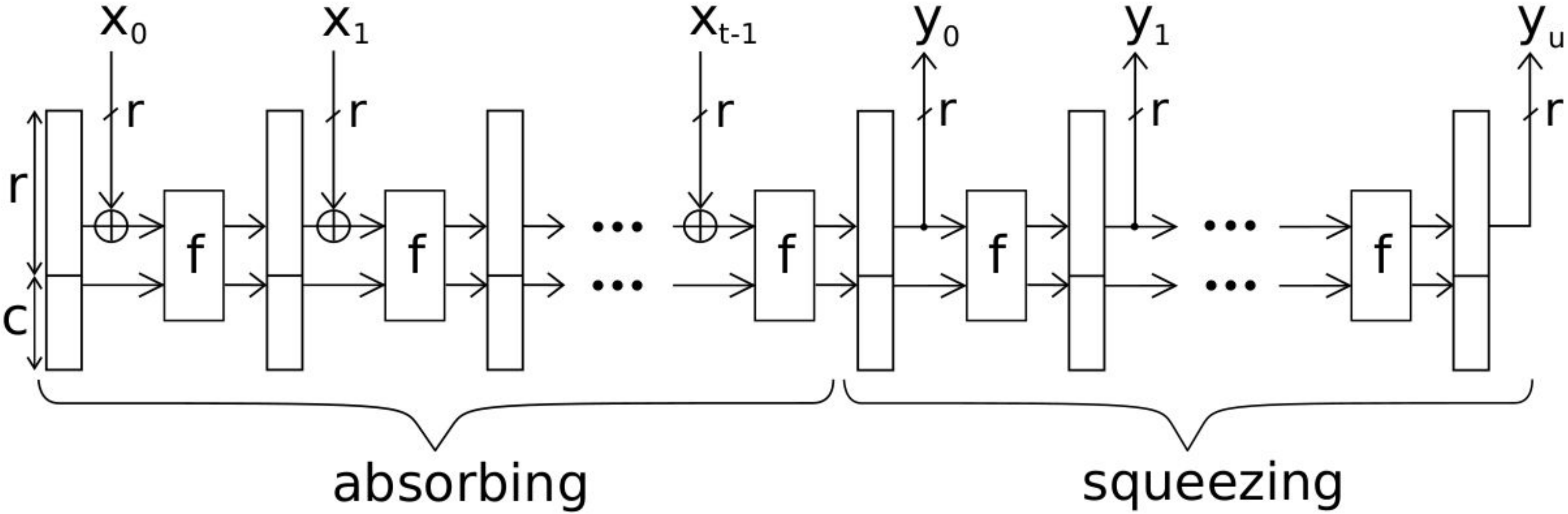
SHA-3

Preprocessing: padding to make message length a multiple of the block size r



Chapter 11 of Understanding Cryptography by Christof Paar and Jan Pelzl

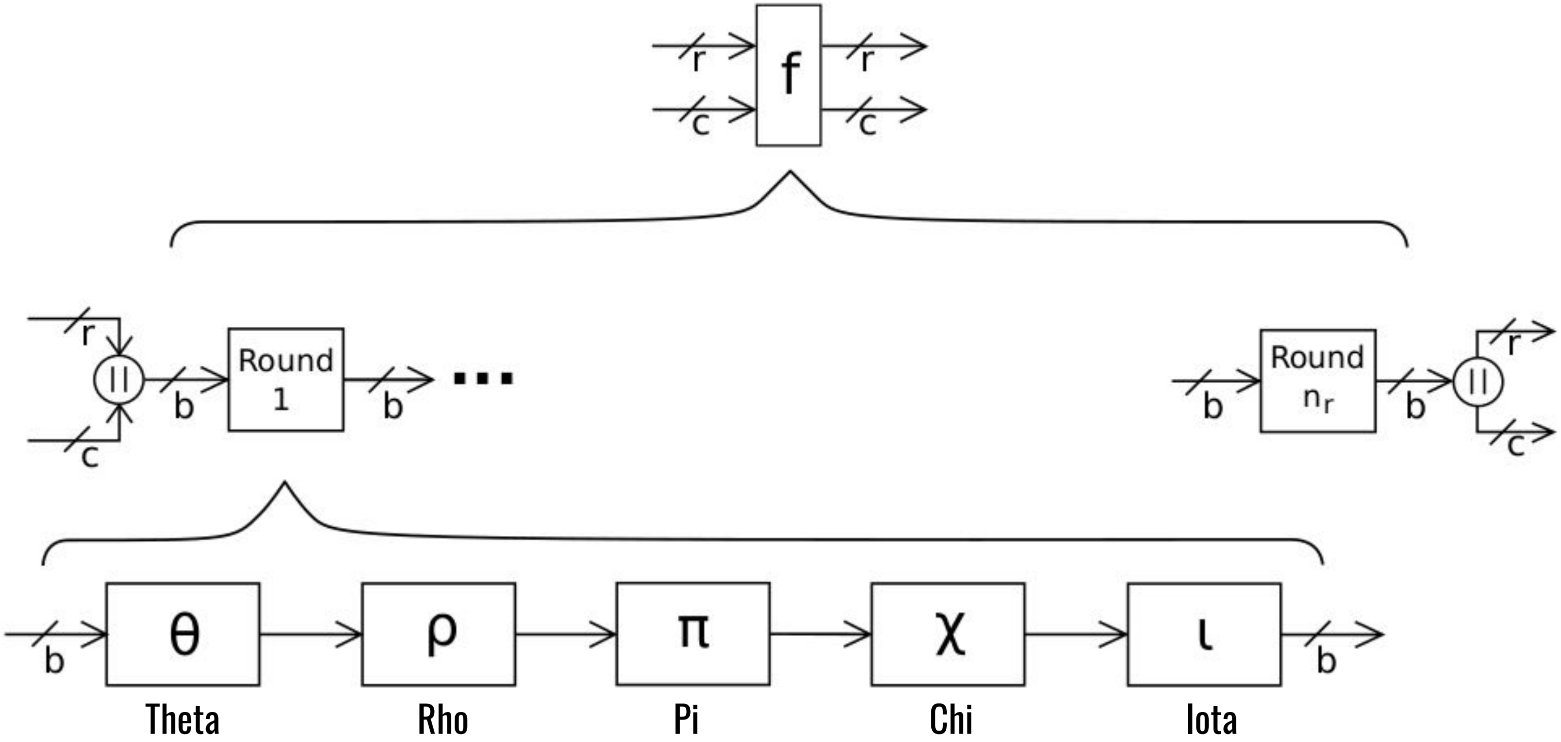
SHA-3: overview



After absorbing phase, we continue applying Keccak only if we need more output bits, otherwise we stop.

Chapter 11 of Understanding Cryptography by Christof Paar and Jan Pelzl

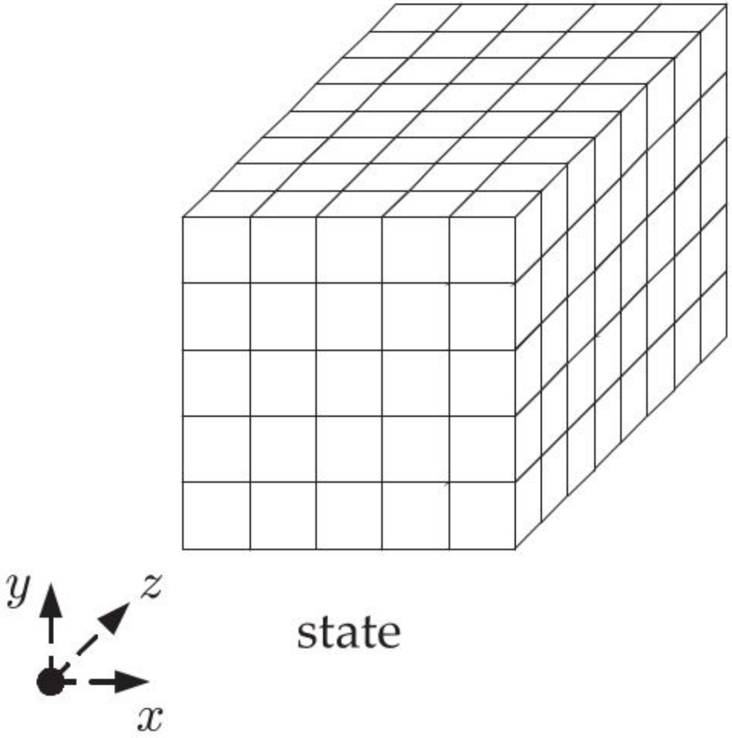
SHA-3: function f



Chapter 11 of Understanding Cryptography by Christof Paar and Jan Pelzl

SHA-3: state

The state b can seen as 5x5x64 cube with 1600 bits (64 layers in SHA-3). Each layer is 5x5=25 matrix.



Chapter 11 of Understanding Cryptography by Christof Paar and Jan Pelzl

SHA-3: Theta step

Each of the 1600 bits is replaced by the xor sum of 10 bits “in its neighborhood” and the original bit itself.

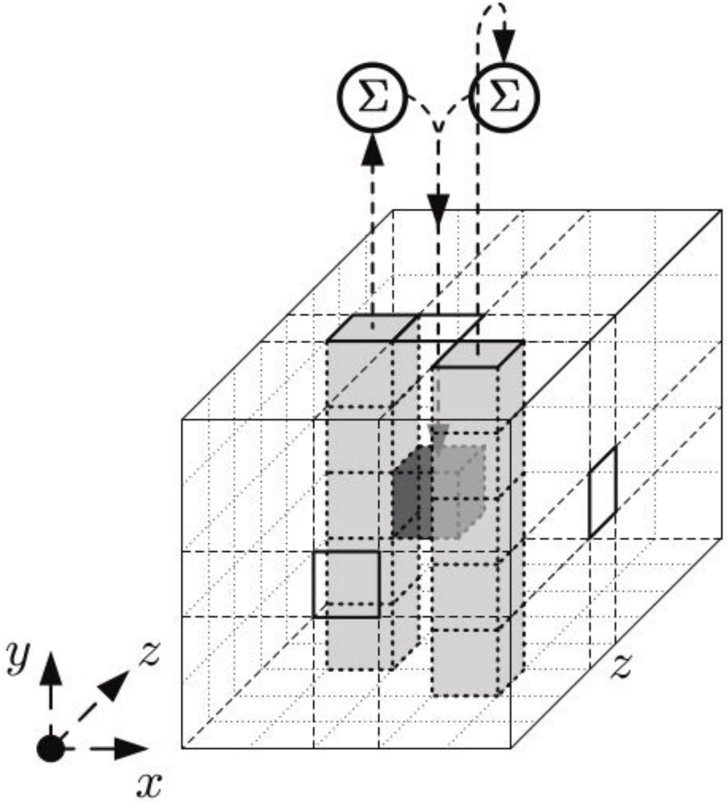
Original bit

XOR

5-bit column to left of the bit

XOR

5-bit column to the right and one position to the front of the bit

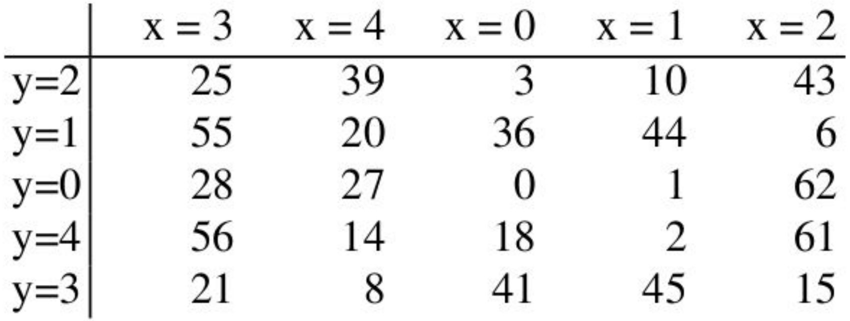


Chapter 11 of Understanding Cryptography by Christof Paar and Jan Pelzl

SHA-3: Rho and Pi steps

We consider words A(x,y) of length 64: we take all 64 bits on the z-axis given a (x, y) position

Rho step: temp[x, y] = rotate(A[x,y], r[x,y]) where r[x,y] is given by



	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

Pi step: permutations of the words based on B[y, 2x+3y] = temp[x, y] (B is the new state)

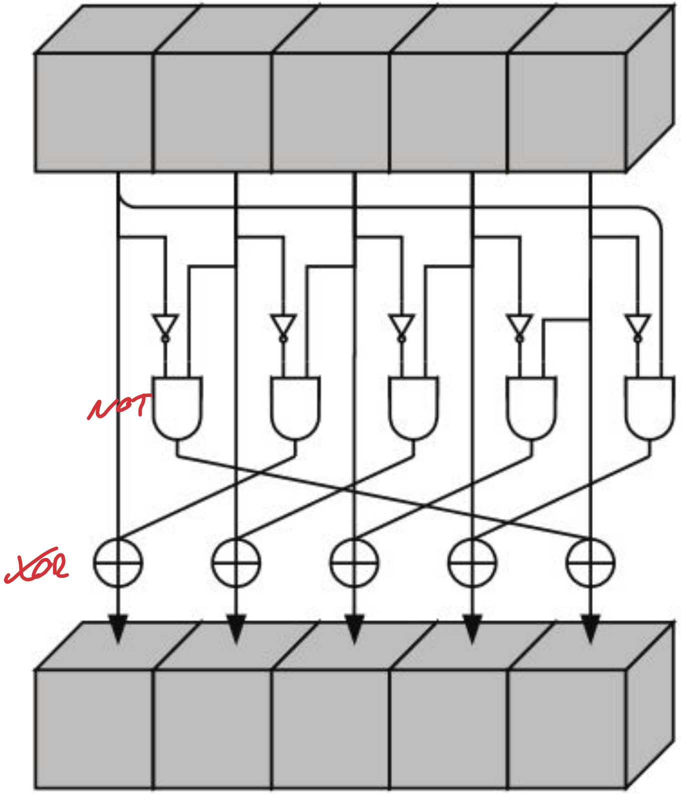
E.g., temp[3,1] = rot(A[3,1], 55) then B[1, 9] = temp[3,1]

Chapter 11 of Understanding Cryptography by Christof Paar and Jan Pelzl

SHA-3: Chi step

$$A[x,y] = B[x,y] \oplus ((\bar{B}[x+1,y]) \wedge B[x+2,y])$$

where $\bar{B}$ is the bitwise complement

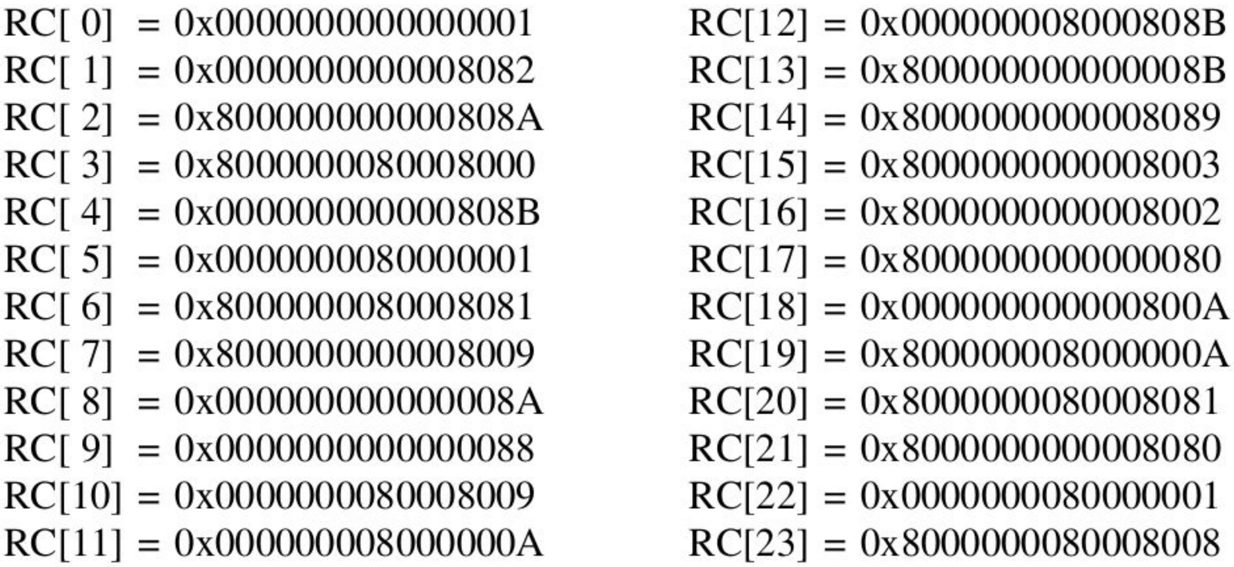


Chapter 11 of Understanding Cryptography by Christof Paar and Jan Pelzl

SHA-3: Iota step

$$A[0,0] = A[0,0] \oplus RC[i]$$

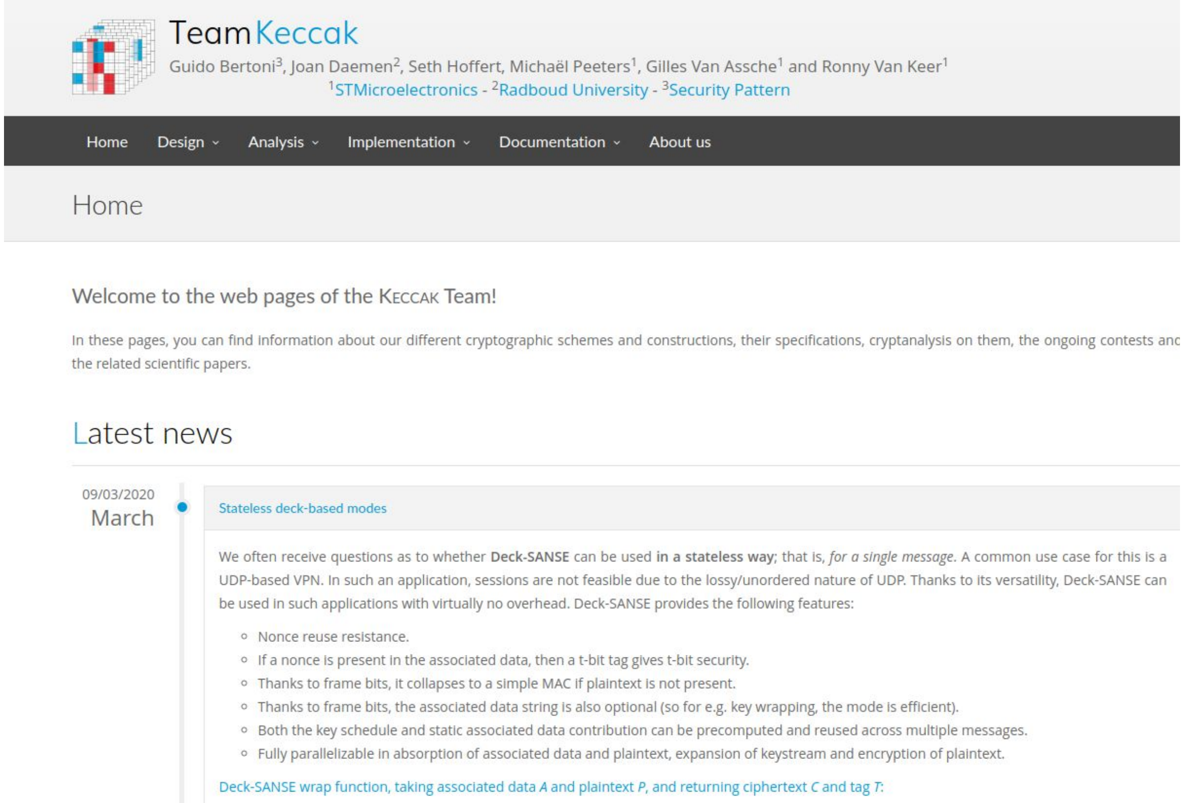
where RC[i] is given by:



RC[0]	=	0x0000000000000001
RC[1]	=	0x0000000000008082
RC[2]	=	0x800000000000808A
RC[3]	=	0x8000000080008000
RC[4]	=	0x000000000000808B
RC[5]	=	0x0000000080000001
RC[6]	=	0x8000000080008081
RC[7]	=	0x8000000000008009
RC[8]	=	0x000000000000008A
RC[9]	=	0x0000000000000088
RC[10]	=	0x0000000080008009
RC[11]	=	0x000000008000000A
RC[12]	=	0x000000008000808B
RC[13]	=	0x800000000000008B
RC[14]	=	0x8000000000008089
RC[15]	=	0x8000000000008003
RC[16]	=	0x8000000000008002
RC[17]	=	0x8000000000000080
RC[18]	=	0x000000000000800A
RC[19]	=	0x800000008000000A
RC[20]	=	0x8000000080008081
RC[21]	=	0x8000000000008080
RC[22]	=	0x0000000080000001
RC[23]	=	0x8000000080008008

Chapter 11 of Understanding Cryptography by Christof Paar and Jan Pelzl

Many useful resources at https://keccak.team



Guido Bertoni[3], Joan Daemen[2], Seth Hoffert, Michaël Peeters[1], Gilles Van Assche[1] and Ronny Van Keer[1]
[1]STMicroelectronics - [2]Radboud University - [3]Security Pattern

Home

Welcome to the web pages of the Keccak Team!

In these pages, you can find information about our different cryptographic schemes and constructions, their specifications, cryptanalysis on them, the ongoing contests anc the related scientific papers.

Latest news

09/03/2020
March

Stateless deck-based modes

We often receive questions as to whether **Deck-SANSE** can be used **in a stateless way**; that is, *for a single message*. A common use case for this is a UDP-based VPN. In such an application, sessions are not feasible due to the lossy/unordered nature of UDP. Thanks to its versatility, Deck-SANSE can be used in such applications with virtually no overhead. Deck-SANSE provides the following features:

- Nonce reuse resistance.
- If a nonce is present in the associated data, then a t-bit tag gives t-bit security.
- Thanks to frame bits, it collapses to a simple MAC if plaintext is not present.
- Thanks to frame bits, the associated data string is also optional (so for e.g. key wrapping, the mode is efficient).
- Both the key schedule and static associated data contribution can be precomputed and reused across multiple messages.
- Fully parallelizable in absorption of associated data and plaintext, expansion of keystream and encryption of plaintext.

Deck-SANSE wrap function, taking associated data *A* and plaintext *P*, and returning ciphertext *C* and tag *T*:

Credits

These slides are based on material from:

- Slides of Prof. D’Amore from CNS 2019-2020
- Christof Paar and Jan Pelzl. Understanding Cryptography: A Textbook for Students and Practitioners. Springer. http://www.crypto-textbook.com/
- Wikipedia (english version)