Information Security
(WBCS004-05)

Fatih Turkmen

Some slides are borrowed from Dr. Frank B. Brokken

Today

- Hashing:
  - Concept and requirements
  - Collisions
  - Use of hashing in cryptography
  - CRC: a non-cryptographic hash
  - MD5, SHA-*x*, Tiger
  - Sharing Secrets / Key Escrow
  - Information Hiding
  - E-mail peculiarities

Hashing

Yes, the alchemists worshipped the antimatter, vowing dark, agile actions while announcing algebra…

Hashing

- Why do we hash?
  - Verification of the integrity of a message
  - Authentication
  - Message fingerprinting
  - Digital Signatures

Requirements

- *Compression:* A hashing function *h* computes a small string/number matching a large piece of information (e.g., a message).

- *Efficiency:* Computing *h(x)* must be easy/fast.

- *Trap-door:* Given *h(x)*, *x* cannot be "easily" retrieved

- *Collision resistant:* Infeasible to find ***y*** for which *h(x)* == *h(y)* (x is given or freely selectable)
  - Collisions *do* exist: If *h()* results in *N* bits and if *x* consists of *M* bits (M > N) then there must exist …. **2^{M-N} collisions.** How many collisions?

Hashing Algorithm

Try 1: Simple addition: If x_i are bytes, $h(x) = \sum x_i \; (mod\ 256)$

✓ Compresses, easy to compute, cannot be inverted.

✘ Unfortunately: many **collisions**

ASCII [1]

Allowing blanks, here are some for hashes

icy porn net □ (69 63 79 20 70 6f 72 6e 20 6e 65 74) = X + 20 (mod256)

inept crony □ (69 6e 65 70 74 20 63 72 6f 6e 79) = X (mod 256)

intern copy □ (69 6e 74 65 72 6e 20 63 6f 70 79) = X (mod 256)

no inept cry □ (6e 6f 20 69 6e 65 70 74 20 63 72 79) = X + 20 (mod 256)

[1] https://commons.wikimedia.org/wiki/File:ASCII-Table-wide.svg

Hashing

Try 2: Modification: If x_i are bytes, multiply the values wi
index $h(x) = \sum i * x_i \ (mod\ 256)$

✓ Compresses, easy to compute, cannot be inverted.

✓ Fewer collisions (i.e., better distribution over the ”has

✘ it's still easy to construct collisions:

	1 * a + 2 * b = h()
@0	1 * 64 + 2 * 48 = 160
>1	1 * 62 + 2 * 49 = 160

❗ Given this scheme, what are the hashes of “@0” and “>1”?

ASCII printable characters							
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Collisions

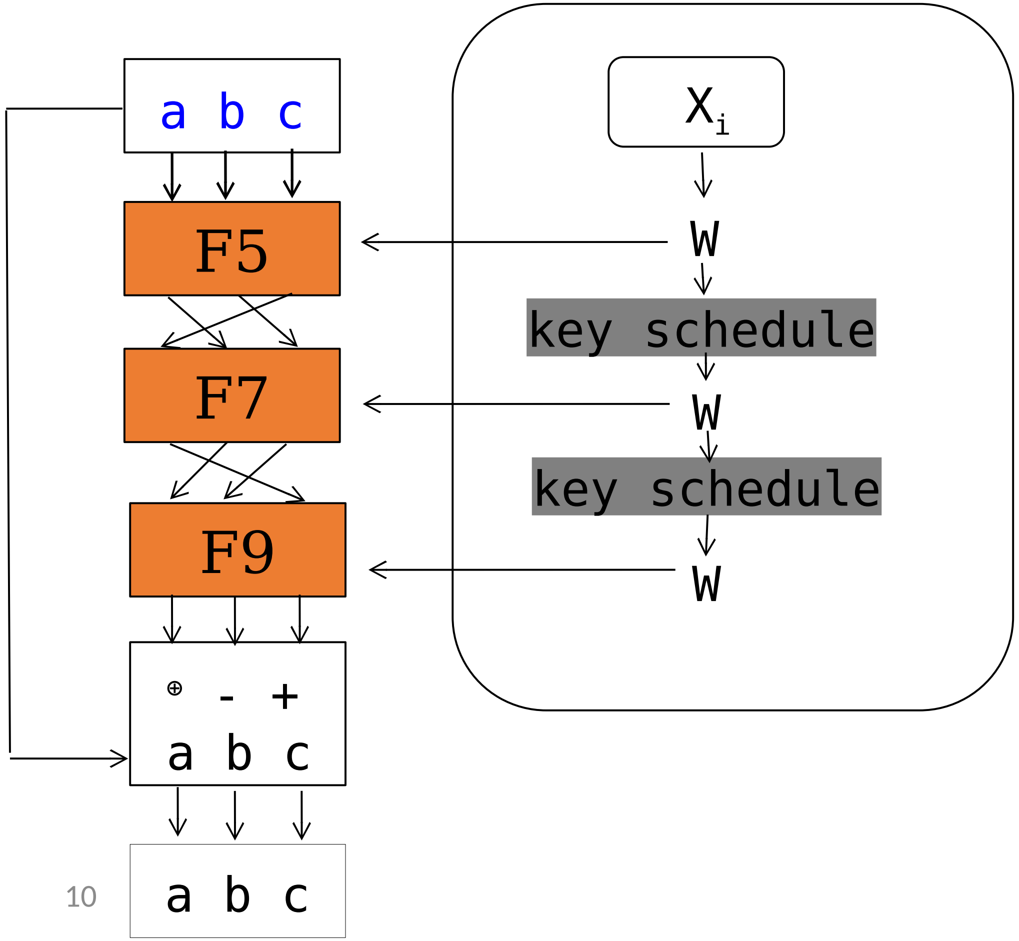
Read about the **Birthday Paradox** in the book!

Tiger Hash

- Developed (1995) by Ross Anderson & Eli Biham.
- Resembles block ciphers
- Operates on blocks of 512 bits (padding may be applied if necessary)
- Resulting hash value (digest): 192 bits (works well with 64-bit processors)
- 4 S-boxes mapping 8 bits to 64 bits
- Uses a *key schedule*, using the input blocks as key.
- Tiger applies one *outer round* on each 512-bit block.

Tiger Hash (Outer Rounds)

- Input is X = $(X_0, X_1, \ldots, X_{n-1})$
- Tiger's **outer round:** Applied to each 512-bit block (i.e. X_i):

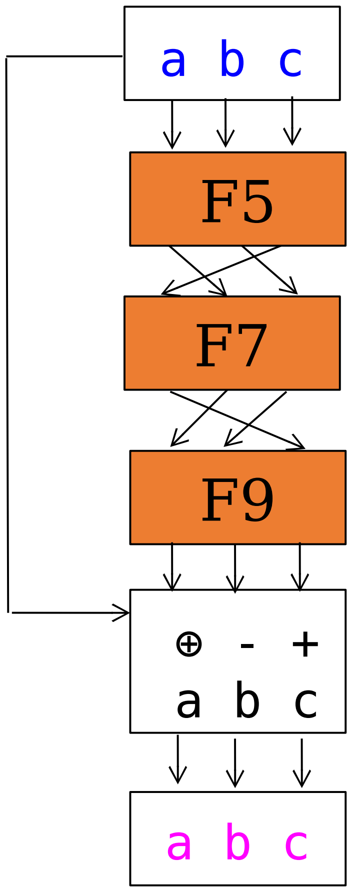


Note: The message itself is used as a key, since there is no key!

- There are n iterations of the outer round
- Initial a, b, c have fixed values (e.g., a is 0x123456789abcdef)

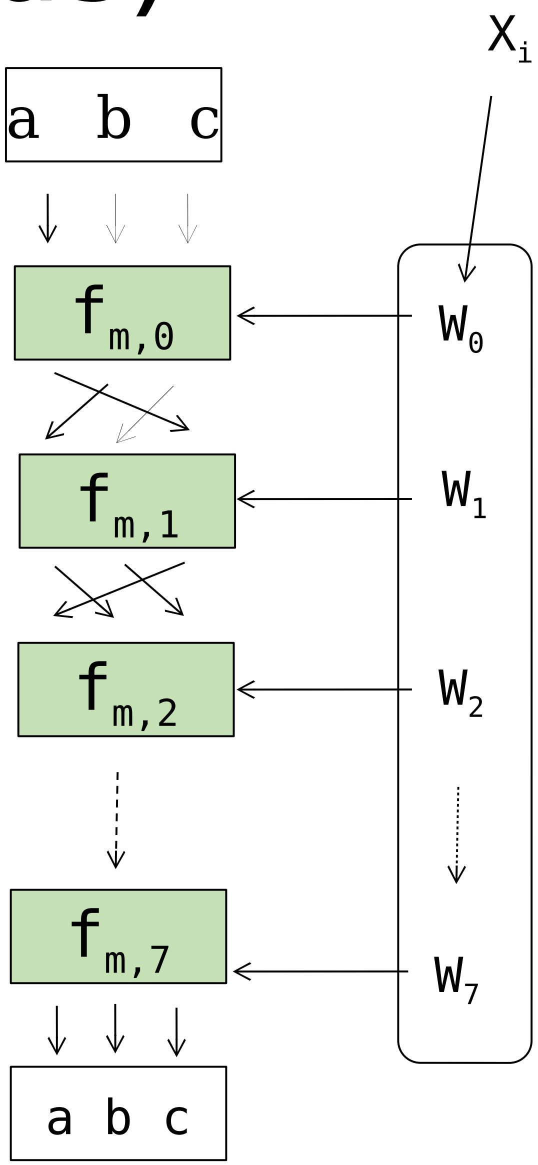
Tiger Hash (Outer Rounds)

- Three outer round functions F_5, F_7, F_9
- a, b, c: each 64 bits
- a leaving F5 becomes b of F7, b leaving F7 becomes a of F9 etc…
- *Final* a, b, c is the hash value, thus the final output is 192 bits



Tiger Hash (Inner Rounds)

- Each F_m consists of 8 *inner rounds where* $m \in \{5,7,9\}$:
  - Each w_i is a 64-bit section of a 512 bit input block, i.e. $W = (w_0, w_1, .., w_{n-1})$
  - Each $f_{m,i}$ receives a permutation of the a,b,c output by $f_{m,i-1}$.
    E.g., *(abc)*, *(bca)*, *(cab)*:
    - $f_{m,0}$ to $f_{m,1}$: output b becomes input a



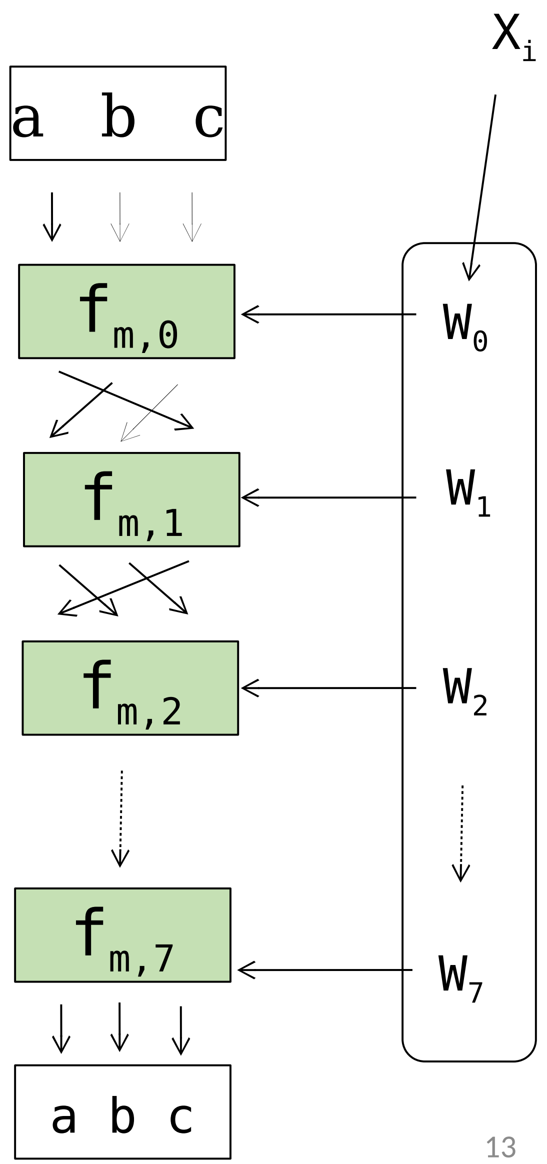
Tiger Hash (Inner Rounds)

An S-box
element

- Final step in inner rounds
  - The 64 c-bits are split in 8 bits
    (c_0 ... c_7):

```
c ⊕= wi
a -= S[0][c0] ⊕ S[1][c2] ⊕ S[2][c4] ⊕
     S[3][c6]
b += S[3][c1] ⊕ S[2][c3] ⊕ S[1][c5] ⊕
     S[0][c7]
b *= m
```

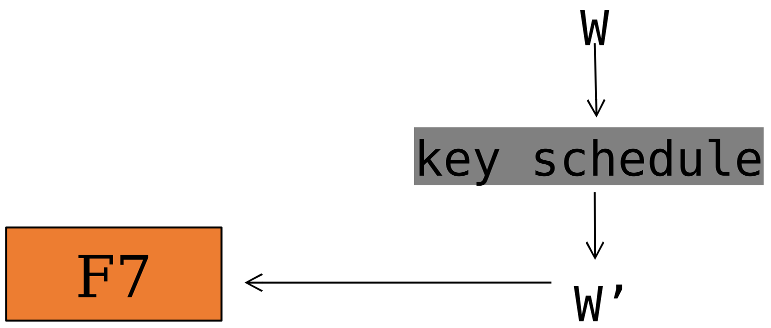
  - The *key schedule* recomputes
    w_0 to w_7 between the f_m boxes (how? Next slide!)
    - cf. Stamp, p. 132.



Tiger Hash

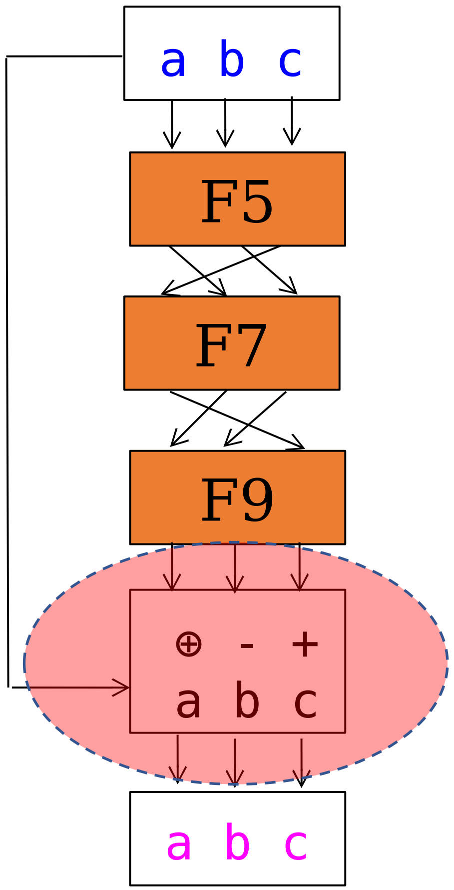
- Tiger's key schedule, simplifying Table 5.1 (see Stamp, p. 93 for more details):

```
w0 -= w7 ⊕ 0xa5a5a5a5a5a5a5a5;
w1 ⊕= w0;
w2 += w1;
w3 -= w2 ⊕ (~w1 << 19);
w4 ⊕= w3;
w5 += w4;
w6 -= w5 ⊕ (~w4 >> 23);
w7 ⊕= w6;
w0 += w7;
w1 -= w0 ⊕ (~w7 << 19);
w2 ⊕= w1;
w3 += w2;
w4 -= w3 ⊕ (~w2 >> 23);
w5 ⊕= w4;
w6 += w5;
w7 -= w6 ⊕ 0x0123456789abcdef;
```



Tiger Hash

- The final step is called feedforward
- The results, say a’, b’, c’ of F9 are XORed, subtracted and added with the initial a, b, c respectively.



- For the original proposal see Tiger Hash paper: https://link.springer.com/content/pdf/10.1007/3-540-60865-6_46.pdf
- For the Cryptanalysis see: https://iacr.org/archive/asiacrypt2007/48330539/48330539.pdf

Refresher on Message Integrity

- Use of cryptography for “Unauthorized Modification” (not about unauthorized reading!) of the plain text
- Message Authentication Code (MAC) (chapter 3.4)

What “principal” is this about?

- How does MAC work?
  - Symmetric encryption, i.e., the same encryption key is used
  - It works in **<u>CBC mode</u>**, i.e., blocks of messages M_0, M_1,..,M_{n-1}

$C_0 = E(M_0 \oplus IV, \mathbf{K})$, $C_1 = E(M_1 \oplus C_0, \mathbf{K})$, …, $C_{n-1} = E(M_{n-1} \oplus C_{n-2}, \mathbf{K})$

- C_{n-1} , also called as CBC residue, serves as the MAC. The rest is discarded for the case of “integrity”.

(Example) Uses of Hashing: Integrity

- Verification of the **integrity** of a message
  - With MAC $\{C_0, C_1,..,C_{n-1}\}$ + C_{n-1} is sent for confidentiality + integrity.
- Requirements:
  - *M* and *h(M)*: changing *M* changes *h(M)* and v.v.
  - *M* and *h(M)* will be sent together since we are interested in integrity…

HMAC

- Integrity with respective to original message “M” must be protected (i.e., the figure).
- Enter: **Hashed Message Authentication Code**.
  - Prevent the change of hash!
  - Hashing functions typically process blocks of bytes.
    - We could prepend (or append) a key K to the message M (i.e., start with the key, or start with the message block(s))

Bob

h(M’), M’

Trudy

Alice

h(M), M

HMAC (cont.)

- So, should we use $h(K, M)$ or $h(M, K)$?

- Let's consider the prepend: $h(K, M)$
  - Hash functions F tend to use *blocks* (*e.g.*, *Tiger uses ?*), e.g., $M = (B1, B2)$.
  - The common case is to use the hash of the previous block as input when computing the next block's hash:
    - $h(M) = F(F(Init, B1), B2)$, where **F** is similar to the outer round of Tiger

What is the block size?

512 bits

HMAC (cont.)

Given *h(K,M) = F(F(K, B1), B2) where M = (B1, B2),*

- Intruder Trudy (1) intercepts M;
  (2) appends a new block X to M (i.e., *M'=(M,X)*);
  and (3) sends the appropriate hash
  - Trudy doesn't know K.
  - Trudy knows M' , h(K, M) and needs to find h(K,M').

Bob

h(K, M'),
M'

?

What happens?

Trudy

Alice

h(K, M),
M

?

HMAC (cont.)

- Use $h(K, M)$? Bad idea…
- $h(K, M) = F(F(K, B1), B2)$
  - If Trudy appends X and sends M' not knowing K:
    - $h(K, M, X) =$

$$F(F(F(A, K), M), X) =$$
$$F(\boxed{h(K, M)}, X)$$

Intercepted by Trudy earlier

Set A=0 if you use Tiger which does not have key but fixed constants..

Bob

$h(K, M')$, M'

Trudy

HMAC (cont.)

$h(M1, K) = h(h(M1),K) = h(h(M2),K)$

- Use $h(M, K)$ instead.
- Less serious but (if there is) a known *collision* ($h(M1) == h(M2)$) renders the hash function *insecure*.
  - Note: M1 and M2 need to be a multiple of the block size.
  - This happened to MD5. SHA1 by now is also considered insecure.
  - Better to use HMAC described in RFC 2104: (https://www.ietf.org/rfc/rfc2104.txt)
  - K, unknown to Trudy, is required to finalize the hash computation

Look for a nice discussion here! : https://stackoverflow.com/questions/7885268/simple-enquiry-on-hash-algorithm

HMAC (cont.)

- *h(M,K)* is preferred over *h(K,M)*
- None of these solutions is complete safe!
  - RFC 2104 offers a solution, B the block length (i.e., 512/8=64)
    - Thoroughly mixing the key to the hash!

```
B: hash block size in bytes (e.g., B = 64)
define: ipad = 0x36 repeated B times
            opad = 0x5C bytes repeated B times.

HMAC(M,K) = h1(K ⊕ opad, h2(K ⊕ ipad, M))
```

  - (ipad and opad could be omitted)
  - Note: h_1 *and* h_2 *are the same hash function.* h_2 is the real work but it reduces the message to digest/hash so h_1 is quickly computed thereafter.
  - See also https://en.wikipedia.org/wiki/HMAC

Example Non-cryptographic Uses of Hashing: CRC

- Cyclic Redundancy Check (CRC)
  - Not a cryptographically-acceptable hash function.
  - Intended for networking applications: detecting transmission errors.
  - WEP uses (inappropriately) CRCs.

Computation of CRC

Given a divisor of size n;

- Add *n-1* 0-bits at the end of the dividend (i.e., data stream).
  - Once the first (leftmost) bit of the input stream is 1: *xor* the leftmost *n* bits of the input stream by the divisor:
  - Continue the process until the remainder is 0 or smaller than the divisor

  - The *remainder* (*n-1* bits) is the CRC.

```
bitstream: 101010110000
divisor:   10011
             11001
```

Computation of CRC

- The remainder (*n-1* bits) is the CRC:

`x | 0 ⊕ d = 0 | c`   *(⊕ d: d `slides' over x|0)*

- So:

`x|0 ⊕ d = c`

- Consequently: the CRC of a bitstream + its CRC equals 0:

`x|c ⊕ d = 0|0`

Computation of CRC

- Example:

The message: 10011100
The divisor: 10011
What is the CRC?

1100

```
bitstream: 101010110000
divisor:   10011
             11001
             10011
              10101
              10011
                11000
                10011
                 10110
                 10011
                   1010
```

```
101010111010
10011
  11001
  10011
   10101
   10011
     11010
     10011
      10011
      10011
          0
```

See this for an explanation of the division (in CRC):
https://www.youtube.com/watch?v=kscjEvjTVBI

Collisions in CRC

```
bitstream: 101010110000   101010111010
divisor:   10011          10011
             11001          11001
             10011          10011
              10101          10101
              10011          10011
                11000          11010
                10011          10011
                 10110          10011
                 10011          10011
                   1010              0
```

- It's easy to create CRC collisions
  - Look at the final (intermediate) value before the CRC bits are added at the end of the bit stream:

```
bitstream: 101010111010
                 ...
                  10101
                  10011
                    11010
```

  - Observation: Once 110 is the remainder of the division, the resulting CRC is 1010
  - Earlier bits are irrelevant, as long as the result, ignoring the remainder, equals 110.

Collisions in CRC

- Finding CRC collisions
  - Change the bitpattern *ad lib*, and turn the final #`divisor` bits into .-characters, then solve for the dots. Originally:

```
bitstream: 101010111010
                (...)
                  10101
                  10011
                    11010
```

Collisions in CRC

- Find a collision: *CRC = 1010*, *d = 10011*

```
original: 10101011

modified: 010.....1010
divisor:   10011
            0.....
             10011
               110
```

find 5 bits

```
bitstream: 101010110000   101010111010
divisor:   10011          10011
             11001          11001
             10011          10011
              10101          10101
              10011          10011
                11000          11010
                10011          10011
                 10110          10011
                 10011          10011
                   1010              0
```

Hashing

- Find a collision: *CRC = 10_d, d = 10011*

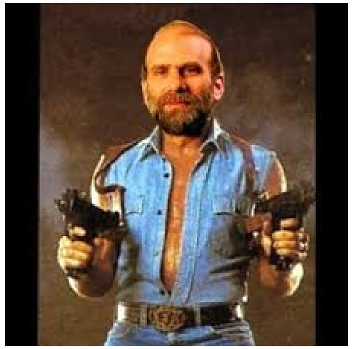
```
original: 10101011

modified: 010.....1010    010110011010
divisor:   10011           10011
            0.....          010101
             10011           10011
             00110             110
```

  - the required bits are all implied and easy to find.

Hashing

- MD5, SHA-1
  - Both MD5 and SHA-1 were extremely popular
  - MDx (128 bit hash) hashes are now considered *insecure*, as collisions can be found.
  - SHA-1 (180 bit) is an improvement, but is in fact by now superseded by SHA-256.
    (cf. http://csrc.nist.gov/groups/ST/hash/policy.html),
    Software computing SHA-x is widely available

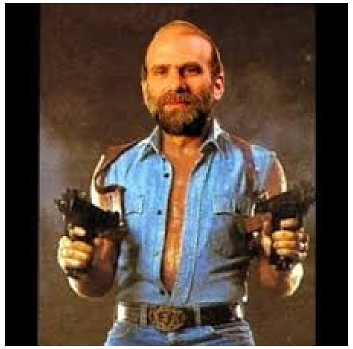
Hashing



- How insecure is SHA-1?

- *Schneier* reports that in approx. $\mathbf{2^{74}}$ computer cycles a SHA-1 collision is found

Hashing

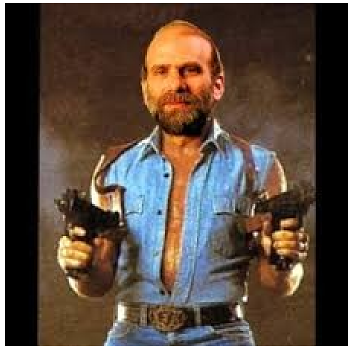


- How insecure is SHA-1?

- *Schneier* reports that in approx. $\mathbf{2^{74}}$ computer cycles a SHA-1 collision is found

- In 2016 a *core* ran at approx. 2^{33} cycles/sec. Assume a *processor* has 8 cores, and a multi-processor *server* 4 processors; then a server did $\mathbf{2^{33+3+2}} = \mathbf{2^{38}}$ cycles/second.

Hashing

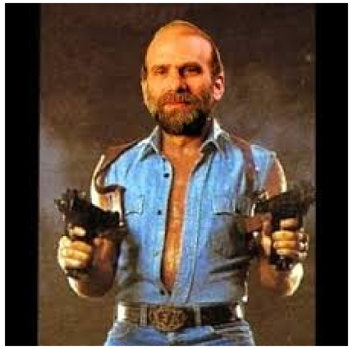


- How insecure is SHA-1?

Schneier reports that in approx. **2^{74}** computer cycles a SHA-1 collision is found
In 2016 a *core* ran at approx. 2^{33} cycles/sec. Assume a *processor* has 8 cores, and a multi-processor *server* 4 processors; then a server did **2^{38}** cycles/second.

In a year there are approx. **2^{25}** seconds. A server-year (s-y) does **2^{63}** cycles, so collisions after (74-63): 2^{11} s-y.

Hashing

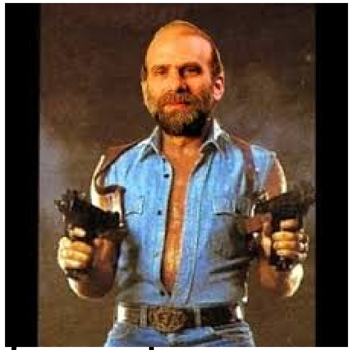


- How insecure is SHA-1?

- *Schneier reports that in approx.* $\mathbf{2^{74}}$ *computer cycles a SHA-1 collision is found*
- In 2016 a *core* ran at approx. 2^{33} cycles/sec. Assume a *processor* typically has 8 cores, and a multi-processor *server* 4 processors; then a server did $\mathbf{2^{38}}$ cycles/second.
- *In a year there are approx.* $\mathbf{2^{25}}$ *seconds. A server-year (s-y) does* $\mathbf{2^{63}}$ *cycles, so*
  collisions after (74-63): 2^{11} *s-y.*

- Using *Moore's law* (computing power doubles every 18 months):
  In 2019: 3/1.5 = 2 doublings in computer power (2^2): 2^{65} cycles, so
  collisions after: 2^9 s-y.
  In 2022: 6/1.5 = 4 doublings (2^4): 2^{67} cycles, collisions after: 2^7 s-y.

Hashing



- How insecure is SHA-1?

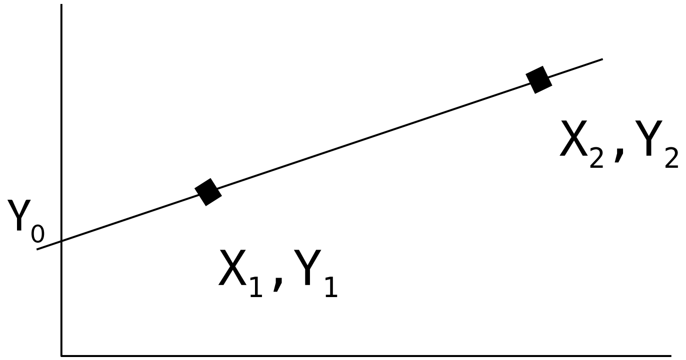
- *Schneier* reports that in approx. **2^{74}** computer cycles a SHA-1 collision is found
- A core runs at approx. 2^{33} cycles/sec. In 2016 processors typically had 8 cores, a multi-processor server had 4 processors, so a server did **2^{38}** cycles/second.
- In a year there are approx. **2^{25}** seconds. A server-year (s-y) did **2^{63}** cycles,
  collision after: 2^{11} s-y.
- Using *Moore's law* (computing power doubles every 18 months):
  In 2019: 3/1.5 = 2 doublings in computer power (2^2): 2^{65} cycles, so
  collisions after: 2^9 s-y.
  In 2022: 6/1.5 = 4 doublings (2^4): 2^{67} cycles, collisions after: 2^7 s-y.

- Renting a server costs approx. €250/yr = approx. €2^8/yr, multiply by #s-y for a collision: a collision attack in 2016 costs approx. €2^{19}, approx. €500k,
  in 2019 2^{17} (€130k), in 2022 2^{15} (€33k).

https://www.schneier.com/blog/archives/2012/10/when_will_we_se.html

Sharing Secrets

- Simple ways to share a secret
  - Basic idea: *polynomials fitting*
    - E.g., straight line - polynomial of degree 1
      - Given two points, the line's equation can be determined



How do we calculate Y_0?

$$Y_0 = Y_1 - X_1 * (Y_2 - Y_1) / (X_2 - X_1)$$

- In general: *n + 1* points are required to determine a polynomial of degree *n*.

Sharing Secrets

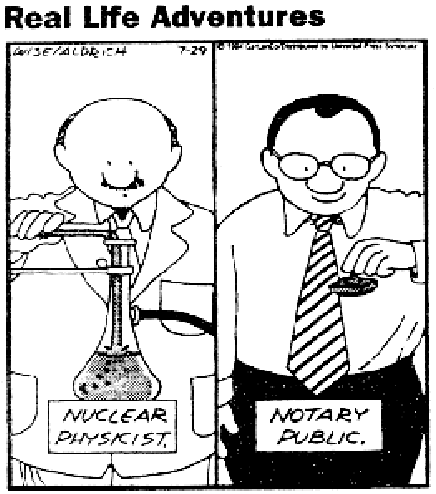
- Simple ways to share a secret
  - Select a polynomial of your choice
    - E.g., $Y = aX + b$
  - A polynomial of degree 1: Alice and Bob each receive one point on this line as the secret info.
  - Using only their own point neither Alice nor Bob can determine the secret.
  - The secret could be, e.g., the Y coordinate for X = 0

Sharing Secrets

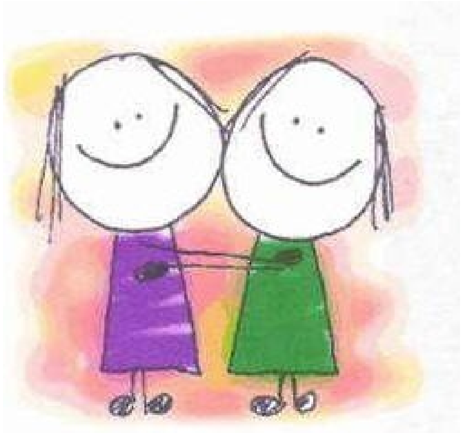
- Simple ways to share a secret
  - E.g., $Y = aX + b$
  - Two parties each receive the coordinates of one point on this line.
  - The secret S is defined as point $(0, S)$, so $b == S$
  - Select a and two points X_a and X_b.
  - Alice gets (X_a,Y_a), where $Y_a = a*X_a + S$,
    Bob gets (X_b,Y_b), where $Y_b = a*X_b + S$.

Key Escrow

- An example use of secret sharing
- Key Escrow
  - Store your secret (i.e., key) with a *trusted* party.
    - A notary? A good friend?



Jobs in which nobody understands what you do.



    - Who do ***you*** trust??

Key Escrow

- Key Escrow
  - Example: The *clipper chip*, announced in 1993 and abandoned by 1996.
    - To be built into all electronic devices offering cryptography
    - Used a symmetric encryption algorithm (*Skipjack*) comparable to DSA
    - *Key escrow* by the US Government…
    - Can the Government be trusted?

Key Escrow

- Key Escrow
  - Alternative:
    - use polynomial key-splitting, requiring *n* people to work together to determine your secret, which may be the *passphrase* to unlock your *file of secrets* or to access your *encrypted file system*.

Key Escrow

- Key Escrow
  - Subtle modification:
    - A polynomial of order *n-1* may be determined if *n* points are provided.
    - Provide *m* points (*m*>*n*) to *m* people, thus implementing an

      $\boxed{\textit{n out of m}}$

      key escrow: any *n* people may join to obtain the secret.

Steganography

- Information Hiding (steganography)
  - Hide information in unlikely places
    - *Yes, the alchemists worshipped the antimatter, vowing dark, agile actions while announcing algebra...*
  - The problem is of course Kerckhoffs principle
  - Unused places can be used to hide information in
    - cf. Stamp's low order bits of a html-file's color attribute
  - *Collusion attacks* (i.e., use *diff)* can be used to reveal hidden information.

What did we learn today?

- Topics this lecture:
  - Concept and requirements
  - Collisions
  - Use of hashing in cryptography
  - CRC: a non-cryptographic hash
  - MD5, SHA-x, Tiger
  - Sharing Secrets / Key Escrow
  - Information Hiding
  - E-mail peculiarities

FAQ

- What is m in Tiger hash?
  - It is a constant value (denoting the round index, i.e., 5,7,9)
- Does Tiger hash work with chaining, meaning for instance the resulting a,b,c, values of hashing X0 would be used in hashing X1, given M={X0,…,Xn}?
  - Yes, the result of the hashing of Xi is the final hash value or the initial value for the next message block Xi+1.
- Why does h(M,K) prevent length extension attacks?
  - TBC

That’s all for today.