Cryptography II: Hash Functions

Computer Security Lecture 3

David Aspinall

School of Informatics
University of Edinburgh

21st January 2013

Outline

Varieties of hash function

Properties of hash functions

Building hash functions

Standard hash functions

Conclusion

Outline

Varieties of hash function

Properties of hash functions

Building hash functions

Standard hash functions

Conclusion

Hash function basics

- A **hash function** is a *computationally efficient* function $h : \{0,1\}^* \rightarrow \{0,1\}^k$ which *compresses* any arbitrary length binary string to a fixed size k-length binary *hash value* (or *hash* for short).

Hash function basics

- A **hash function** is a *computationally efficient* function $h : \{0,1\}^* \to \{0,1\}^k$ which *compresses* any arbitrary length binary string to a fixed size k-length binary *hash value* (or *hash* for short).
- A good hash function distributes values uniformly: the probability that a randomly chosen string s gets mapped to a particular hash y is $\frac{1}{2^k}$

Hash function basics

- A **hash function** is a *computationally efficient* function $h : \{0,1\}^* \rightarrow \{0,1\}^k$ which *compresses* any arbitrary length binary string to a fixed size k-length binary *hash value* (or *hash* for short).
- A good hash function distributes values uniformly: the probability that a randomly chosen string s gets mapped to a particular hash y is $\frac{1}{2^k}$
- A **cryptographic hash function** must satisfy some further properties, e.g.:
  1. it should be difficult to invert;
  2. it should be difficult to find a second input that hashes to the same value as another input;
  3. it should be difficult to find any two inputs that hash to the same value.

Hash function uses and non-uses

- **Integrity**: Alice sends $m, h(m)$ (or alternatively, $E_K(m||h(m))$) to Bob.
- Protects against *malicious* modification.
- **Confidentiality**: An *Authentication Server* stores a user's password p as $h(p)$.
- Other uses: confirming knowledge (e.g. password) without revealing, deriving keys, pseudo-random numbers. A piece of "cryptographic glue".
- On their own, hash functions don't protect against
  - Malicious repetition of data, e.g., repeating a £100 bank deposit. (**Ex.** how could you do that?)
  - Dishonest repudiation, e.g., denying sending a hashed email message with a correct hash.
- Nor do they support message recovery, i.e., recovering the original message after tampering

Properties of cryptographic hash functions

Preimage Resistance (One-way)

h is **preimage resistant** if given a hash value y, it is computationally infeasible to find an x such that $h(x) = y$.

Properties of cryptographic hash functions

Preimage Resistance (One-way)

h is **preimage resistant** if given a hash value y, it is computationally infeasible to find an x such that $h(x) = y$.

2nd Preimage Resistance (Weak Collision Resistance)

h is **2nd preimage resistant** if given a value x_1 and its hash $h(x_1)$, it is computationally infeasible to find another x_2 such that $h(x_2) = h(x_1)$.

Properties of cryptographic hash functions

Preimage Resistance (One-way)

h is **preimage resistant** if given a hash value y, it is computationally infeasible to find an x such that $h(x) = y$.

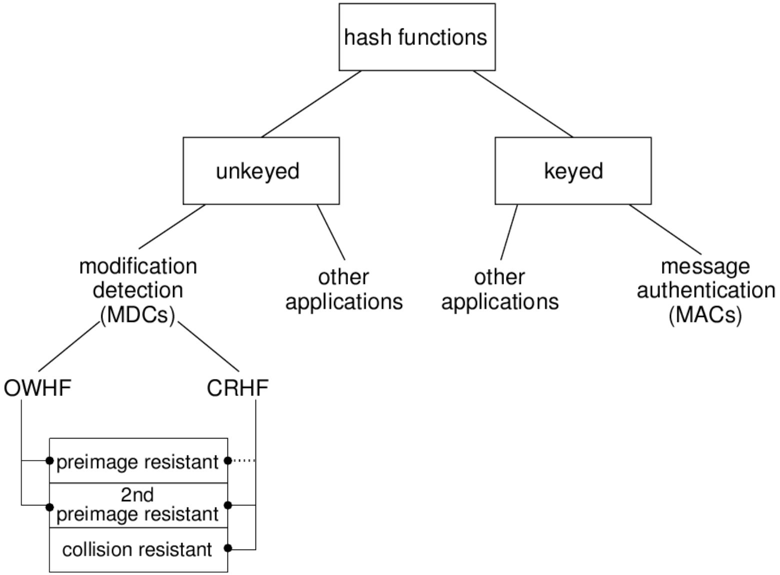
2nd Preimage Resistance (Weak Collision Resistance)

h is **2nd preimage resistant** if given a value x_1 and its hash $h(x_1)$, it is computationally infeasible to find another x_2 such that $h(x_2) = h(x_1)$.

(Strong) Collision Resistance

h is **collision resistant** if it is computationally infeasible to find *any* two inputs x_1 and x_2 such that $h(x_1) = h(x_2)$.

Hash function Classification [HAC]



Modification Detection Codes

- The main application of hash functions is as **Modification Detection Codes** to provide **data integrity**.
- A hash $h(x)$ provides a short *message digest*, a "fingerprint" of some possibly large data x. If the data is altered, the digest should become invalid.
  - This allows the data (but not the hash!) to be stored in an unsecured place.
  - If x is altered to x', we hope $h(x) \neq h(x')$, so it can be detected.
- This is useful especially where *malicious* alteration is a concern, e.g., software distribution.
- Ordinary hash functions such as CRC-checkers produce *checksums* which are not 2nd preimage resistant: an attacker could produce a hacked version of a software product and ensure the checksum remained the same.

Varieties of MDCs

- A **one-way hash function (OWHF)** is a hash function that satisfies preimage resistance and 2nd-preimage resistance.

Varieties of MDCs

- A **one-way hash function (OWHF)** is a hash function that satisfies preimage resistance and 2nd-preimage resistance.
- A **collision resistant hash function (CRHF)** is a hash function that satisfies 2nd-preimage resistance and collision resistance.

Varieties of MDCs

- A **one-way hash function (OWHF)** is a hash function that satisfies preimage resistance and 2nd-preimage resistance.
- A **collision resistant hash function (CRHF)** is a hash function that satisfies 2nd-preimage resistance and collision resistance.
- In practice, CRHF usually satisfies preimage resistance.

Varieties of MDCs

- A **one-way hash function (OWHF)** is a hash function that satisfies preimage resistance and 2nd-preimage resistance.
- A **collision resistant hash function (CRHF)** is a hash function that satisfies 2nd-preimage resistance and collision resistance.
- In practice, CRHF usually satisfies preimage resistance.
- CRHFs are harder to construct than OWHFs and have longer length hash values.

Varieties of MDCs

- A **one-way hash function (OWHF)** is a hash function that satisfies preimage resistance and 2nd-preimage resistance.
- A **collision resistant hash function (CRHF)** is a hash function that satisfies 2nd-preimage resistance and collision resistance.
- In practice, CRHF usually satisfies preimage resistance.
- CRHFs are harder to construct than OWHFs and have longer length hash values.
- Choice between OWHF and CRHF depends on application:

Varieties of MDCs

- A **one-way hash function (OWHF)** is a hash function that satisfies preimage resistance and 2nd-preimage resistance.
- A **collision resistant hash function (CRHF)** is a hash function that satisfies 2nd-preimage resistance and collision resistance.
- In practice, CRHF usually satisfies preimage resistance.
- CRHFs are harder to construct than OWHFs and have longer length hash values.
- Choice between OWHF and CRHF depends on application:
  - If attacker can control input, CRHF required.

Varieties of MDCs

- A **one-way hash function (OWHF)** is a hash function that satisfies preimage resistance and 2nd-preimage resistance.
- A **collision resistant hash function (CRHF)** is a hash function that satisfies 2nd-preimage resistance and collision resistance.
- In practice, CRHF usually satisfies preimage resistance.
- CRHFs are harder to construct than OWHFs and have longer length hash values.
- Choice between OWHF and CRHF depends on application:
  - If attacker can control input, CRHF required.
  - Otherwise OWHF suffices

Varieties of MDCs

- A **one-way hash function (OWHF)** is a hash function that satisfies preimage resistance and 2nd-preimage resistance.
- A **collision resistant hash function (CRHF)** is a hash function that satisfies 2nd-preimage resistance and collision resistance.
- In practice, CRHF usually satisfies preimage resistance.
- CRHFs are harder to construct than OWHFs and have longer length hash values.
- Choice between OWHF and CRHF depends on application:
  - If attacker can control input, CRHF required.
  - Otherwise OWHF suffices
- **Ex**: which is needed for password file security?

Message Authentication Codes

- **Message Authentication Codes** are *keyed* hash functions, indexed with a secret key.
  - As well as data integrity, they provide **data-origin authentication**, because it is assumed that apart from the recipient, only the sender knows the secret key necessary to compute the MAC.
- A MAC is a key-indexed family of hash functions, $\{h_k \mid k \in \mathcal{K}\}$. MACs must satisfy a *computation resistance* property.

Computation Resistance

Given a set of pairs $(x_i, h_k(x_i))$ it is computationally infeasible to find any other text-MAC pair $(x, h_k(x))$ for a new input $x \neq x_i$.

Outline

Varieties of hash function

Properties of hash functions

Building hash functions

Standard hash functions

Conclusion

Relationships between properties

- **Collision resistance implies 2nd-preimage resistance**.

Relationships between properties

- **Collision resistance implies 2nd-preimage resistance**.
- Sketch proof [HAC]:

Relationships between properties

- **Collision resistance implies 2nd-preimage resistance**.
- Sketch proof [HAC]:
  - Let h be CR, but suppose it is not 2nd PI.

Relationships between properties

- **Collision resistance implies 2nd-preimage resistance**.
- Sketch proof [HAC]:
  - Let h be CR, but suppose it is not 2nd PI.
  - Fix some input x; compute $h(x)$.

Relationships between properties

- **Collision resistance implies 2nd-preimage resistance**.
- Sketch proof [HAC]:
  - Let h be CR, but suppose it is not 2nd PI.
  - Fix some input x; compute $h(x)$.
  - Since not 2nd PI, we can find an $x' \neq x$ with $h(x') = h(x)$.

Relationships between properties

- **Collision resistance implies 2nd-preimage resistance**.
- Sketch proof [HAC]:
  - Let h be CR, but suppose it is not 2nd PI.
  - Fix some input x; compute $h(x)$.
  - Since not 2nd PI, we can find an $x' \neq x$ with $h(x') = h(x)$.
  - But now (x, x') is a collision, so h cannot be CR.

Relationships between properties

- **Collision resistance implies 2nd-preimage resistance**.
- Sketch proof [HAC]:
  - Let h be CR, but suppose it is not 2nd PI.
  - Fix some input x; compute $h(x)$.
  - Since not 2nd PI, we can find an $x' \neq x$ with $h(x') = h(x)$.
  - But now (x, x') is a collision, so h cannot be CR.
- This and similar arguments (e.g., see Smart) can be made precise using the *Random Oracle Model*.

Relationships between properties

- **Collision resistance implies 2nd-preimage resistance**.
- Sketch proof [HAC]:
  - Let h be CR, but suppose it is not 2nd PI.
  - Fix some input x; compute $h(x)$.
  - Since not 2nd PI, we can find an $x' \neq x$ with $h(x') = h(x)$.
  - But now (x, x') is a collision, so h cannot be CR.
- This and similar arguments (e.g., see Smart) can be made precise using the *Random Oracle Model*.
- **Collision resistance does not imply preimage resistance**

Relationships between properties

- **Collision resistance implies 2nd-preimage resistance**.
- Sketch proof [HAC]:
  - Let h be CR, but suppose it is not 2nd PI.
  - Fix some input x; compute $h(x)$.
  - Since not 2nd PI, we can find an $x' \neq x$ with $h(x') = h(x)$.
  - But now (x, x') is a collision, so h cannot be CR.
- This and similar arguments (e.g., see Smart) can be made precise using the *Random Oracle Model*.
- **Collision resistance does not imply preimage resistance**
- Contrived counterexample:

$$h(x) = \begin{cases} 1 \mid\mid x & \text{if } x \text{ has length } n \\ 0 \mid\mid g(x) & \text{otherwise} \end{cases}$$

Collision Resistance and Birthday Attacks

- To satisfy (strong) collision resistance, a hash function must be large enough to withstand a **birthday attack**. (or *square root attack*).

Collision Resistance and Birthday Attacks

- To satisfy (strong) collision resistance, a hash function must be large enough to withstand a **birthday attack**. (or *square root attack*).
- Drawing random elements with replacement from a set of k elements, a repeat is likely after about $\sqrt{k}$ selections.

Collision Resistance and Birthday Attacks

- To satisfy (strong) collision resistance, a hash function must be large enough to withstand a **birthday attack**. (or *square root attack*).
- Drawing random elements with replacement from a set of k elements, a repeat is likely after about $\sqrt{k}$ selections.
- Mallory has two contracts, one for £1000, the other £100,000, to be signed with a 64-bit hash. He makes 2^{32} minor variations in each (e.g spaces/control chars), and finds a pair with the same hash. Later claims second document was signed, not first.

Collision Resistance and Birthday Attacks

- To satisfy (strong) collision resistance, a hash function must be large enough to withstand a **birthday attack**. (or *square root attack*).
- Drawing random elements with replacement from a set of k elements, a repeat is likely after about $\sqrt{k}$ selections.
- Mallory has two contracts, one for £1000, the other £100,000, to be signed with a 64-bit hash. He makes 2^{32} minor variations in each (e.g spaces/control chars), and finds a pair with the same hash. Later claims second document was signed, not first.
- An n-bit unkeyed hash function has **ideal security** if producing a preimage or 2nd-preimage each requires 2^n operations, and producing a collision requires $2^{n/2}$ operations.

Outline

Varieties of hash function

Properties of hash functions

Building hash functions

Standard hash functions

Conclusion

From one-way functions to MDCs

- **Multiplication of large primes** is a OWF

From one-way functions to MDCs

- **Multiplication of large primes** is a OWF
  - for appropriate choices of p and q, $f(p,q) = pq$ is a one-way function since *integer factorization [FACTORING]* is difficult.

From one-way functions to MDCs

- **Multiplication of large primes** is a OWF
  - for appropriate choices of p and q, $f(p,q) = pq$ is a one-way function since *integer factorization [FACTORING]* is difficult.
  - Not feasible to turn into an MD function, though. (**Ex**: why?)

From one-way functions to MDCs

- **Multiplication of large primes** is a OWF
  - for appropriate choices of p and q, $f(p,q) = pq$ is a one-way function since *integer factorization [FACTORING]* is difficult.
  - Not feasible to turn into an MD function, though. (**Ex**: why?)
- **Exponentiation in finite fields** is a OWF

From one-way functions to MDCs

- **Multiplication of large primes** is a OWF
  - for appropriate choices of p and q, $f(p, q) = pq$ is a one-way function since *integer factorization [FACTORING]* is difficult.
  - Not feasible to turn into an MD function, though. (**Ex**: why?)
- **Exponentiation in finite fields** is a OWF
  - for appropriate primes p and numbers α, $f(x) = \alpha^x \bmod p$ is a one-way function, since the *discrete logarithm problem [DLP]* is difficult.

From one-way functions to MDCs

- **Multiplication of large primes** is a OWF
  - for appropriate choices of p and q, $f(p,q) = pq$ is a one-way function since *integer factorization [FACTORING]* is difficult.
  - Not feasible to turn into an MD function, though. (**Ex**: why?)
- **Exponentiation in finite fields** is a OWF
  - for appropriate primes p and numbers α, $f(x) = \alpha^x \bmod p$ is a one-way function, since the *discrete logarithm problem [DLP]* is difficult.
  - Main problem with turning this into a realistic MD function is that it's too slow to calculate.

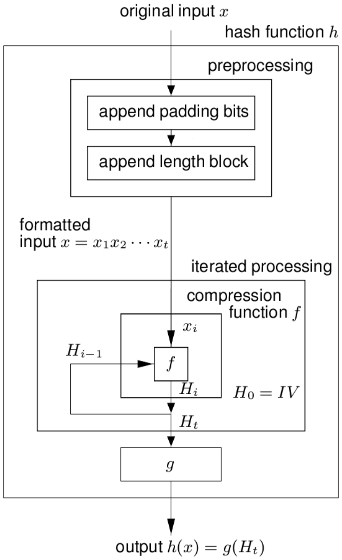
OWFs from block ciphers

- A block cipher is an encryption scheme which works on fixed length blocks of input text.
- We can construct a OWF from a block cipher such as DES, which is treated essentially as a random function:

  $$h(x) = E_k(x) \oplus x$$

  for fixed key k. This *can* be turned into a MD function, by iteration...

Iterated hash function construction [HAC]



Building up hash functions

- An **iterated hash function** is constructed using a *compression function* f which converts a $t+n$-bit input into an n-bit output.

Building up hash functions

- An **iterated hash function** is constructed using a *compression function* f which converts a $t+n$-bit input into an n-bit output.
  - The input x is split into blocks $x_1\, x_2, \ldots x_k$ of size t, appending padding bits and a *length block* indicating the original length.

$$H_0 = IV \qquad H_i = f(H_{i-1}, x_i),\ \ 1 \leq i \leq k \qquad h(x) = g(H_k).$$

Building up hash functions

- An **iterated hash function** is constructed using a *compression function* f which converts a $t+n$-bit input into an n-bit output.
  - The input x is split into blocks $x_1 x_2, \ldots x_k$ of size t, appending padding bits and a *length block* indicating the original length.

$$H_0 = IV \qquad H_i = f(H_{i-1}, x_i),\ \ 1 \leq i \leq k \qquad h(x) = g(H_k).$$

  - IV: an initialization vector; g: an output transformation (often identity).

Building up hash functions

- An **iterated hash function** is constructed using a *compression function* f which converts a $t+n$-bit input into an n-bit output.
  - The input x is split into blocks $x_1 x_2, \ldots x_k$ of size t, appending padding bits and a *length block* indicating the original length.

$$H_0 = IV \qquad H_i = f(H_{i-1}, x_i),\ \ 1 \leq i \leq k \qquad h(x) = g(H_k).$$

  - IV: an initialization vector; g: an output transformation (often identity).
- This is **Merkle's meta-method**

Building up hash functions

- An **iterated hash function** is constructed using a *compression function* f which converts a $t+n$-bit input into an n-bit output.
  - The input x is split into blocks $x_1 x_2, \ldots x_k$ of size t, appending padding bits and a *length block* indicating the original length.

$$H_0 = IV \qquad H_i = f(H_{i-1}, x_i),\ \ 1 \leq i \leq k \qquad h(x) = g(H_k).$$

  - IV: an initialization vector; g: an output transformation (often identity).
- This is **Merkle's meta-method**
  - Fact: any CR compression function f can be extended to a CRHF by the above construction, and

Building up hash functions

- An **iterated hash function** is constructed using a *compression function* f which converts a $t+n$-bit input into an n-bit output.
  - The input x is split into blocks $x_1 x_2, \ldots x_k$ of size t, appending padding bits and a *length block* indicating the original length.

$$H_0 = IV \qquad H_i = f(H_{i-1}, x_i),\ \ 1 \leq i \leq k \qquad h(x) = g(H_k).$$

  - IV: an initialization vector; g: an output transformation (often identity).
- This is **Merkle's meta-method**
  - Fact: any CR compression function f can be extended to a CRHF by the above construction, and
  - *padding*: the last block with 0s, adding a final extra block x_k which holds right-justified binary representation of $\mathit{length}(x)$ (this padding is called **MD strengthening**).

Building up hash functions

- An **iterated hash function** is constructed using a *compression function* f which converts a $t+n$-bit input into an n-bit output.
  - The input x is split into blocks $x_1 x_2, \ldots x_k$ of size t, appending padding bits and a *length block* indicating the original length.

$$H_0 = IV \qquad H_i = f(H_{i-1}, x_i),\ \ 1 \leq i \leq k \qquad h(x) = g(H_k).$$

  - IV: an initialization vector; g: an output transformation (often identity).
- This is **Merkle's meta-method**
  - Fact: any CR compression function f can be extended to a CRHF by the above construction, and
  - *padding*: the last block with 0s, adding a final extra block x_k which holds right-justified binary representation of $length(x)$ (this padding is called **MD strengthening**).
  - Set $IV = 0^n$, $g = id$, and compute $H_i = f(H_{i-1}, x_i)$.

Outline

Varieties of hash function

Properties of hash functions

Building hash functions

Standard hash functions

Conclusion

MD5

- Improvement of MD4; MD4 and MD5 designed by Ron Rivest.
  - Text processed in 512-bit blocks, as 16 32-bit sub-blocks. Output is four 32-bit blocks, giving a **128-bit** hash. Message padded with 1 and then 0s until last block is 448 bits long, then a 64-bit length.

MD5

- Improvement of MD4; MD4 and MD5 designed by Ron Rivest.
  - Text processed in 512-bit blocks, as 16 32-bit sub-blocks. Output is four 32-bit blocks, giving a **128-bit** hash. Message padded with 1 and then 0s until last block is 448 bits long, then a 64-bit length.
  - Main loop has four rounds, chaining 4 variables a, b, c, d. Each round uses a different operation (with a similar structure) 16 times, which computes a new value of one of the four variables using a non-linear function of the other three, chosen to preserve randomness properties of the input.

MD5

- Improvement of MD4; MD4 and MD5 designed by Ron Rivest.
  - Text processed in 512-bit blocks, as 16 32-bit sub-blocks. Output is four 32-bit blocks, giving a **128-bit** hash. Message padded with 1 and then 0s until last block is 448 bits long, then a 64-bit length.
  - Main loop has four rounds, chaining 4 variables a, b, c, d. Each round uses a different operation (with a similar structure) 16 times, which computes a new value of one of the four variables using a non-linear function of the other three, chosen to preserve randomness properties of the input.
  - For example, the first round uses the operation:

$$\begin{aligned} a &= (F(b,c,d) + x_i + t_j) <<< s \\ F(b,c,d) &= (b \wedge c) \vee (\neg b \wedge d) \end{aligned}$$

    where $<<< s$ is left-circular shift of s bits, x_i is the ith sub-block of the message. Constants t_j are the integer part of $2^{32} * \mathrm{abs}(\sin(i+1))$ where $0 \leq i \leq 63$ is in radians (for the 4 * 16 steps).

Secure Hash Algorithm SHA-1 (160)

SHA-1 is a NIST standard [FIPS 180] also based on MD4.
An attack strategy with cost 2^{51} was found in 2011.

- Five 32-bit blocks are chained; output is 160 bits. Message blocks 512 bits. Padding like MD5.

Secure Hash Algorithm SHA-1 (160)

SHA-1 is a NIST standard [FIPS 180] also based on MD4.
An attack strategy with cost 2^{51} was found in 2011.

- Five 32-bit blocks are chained; output is 160 bits. Message blocks 512 bits. Padding like MD5.
  - Main loop has four rounds of 20 operations, chaining 5 variables a, b, c, d, e, f. Five IVs and four constants are used:

$$A = \texttt{0x67452301}$$
$$B = \texttt{0xEFCDAB89}$$
$$C = \texttt{0x98BADCFE}$$
$$D = \texttt{0x10325476}$$
$$E = \texttt{0xC3D2E1F0}$$

$$K_0 = \texttt{0x5A827999}$$
$$K_1 = \texttt{0x6ED9EBA1}$$
$$K_2 = \texttt{0x8F1BBCDC}$$
$$K_3 = \texttt{0xCA62C1D6}$$

Secure Hash Algorithm SHA-1 (160)

SHA-1 is a NIST standard [FIPS 180] also based on MD4.
An attack strategy with cost 2^{51} was found in 2011.

- Five 32-bit blocks are chained; output is 160 bits. Message blocks 512 bits. Padding like MD5.
  - Main loop has four rounds of 20 operations, chaining 5 variables a, b, c, d, e, f. Five IVs and four constants are used:

$$\begin{aligned} A &= \texttt{0x67452301} \\ B &= \texttt{0xEFCDAB89} \\ C &= \texttt{0x98BADCFE} \\ D &= \texttt{0x10325476} \\ E &= \texttt{0xC3D2E1F0} \end{aligned} \qquad \begin{aligned} K_0 &= \texttt{0x5A827999} \\ K_1 &= \texttt{0x6ED9EBA1} \\ K_2 &= \texttt{0x8F1BBCDC} \\ K_3 &= \texttt{0xCA62C1D6} \end{aligned}$$

  - The message block undergoes an *expansion transformation* from 16*32-bit words x_i to 80*32-bit words, w_i by:

$$\begin{aligned} w_i &= x_i, & \text{for } 0 \leq i \leq 15. \\ w_i &= (w_{i-3} \oplus w_{i-8} \oplus \\ & \qquad w_{i-14} \oplus w_{i-16}) <<< 1, & \text{for } 16 \leq i \leq 79. \end{aligned}$$

SHA-1 (160) continued

- 80 steps in main loop, changing Ks and Fs 4 times

SHA-1 (160) continued

- 80 steps in main loop, changing Ks and Fs 4 times
  - Where $j = i/20$:

$$
\begin{aligned}
&\textbf{for}(\ i = 0;\ i < 80;\ i{+}{+}\ ) \quad \{ \\
&\quad tmp = (a <<< 5) + F_j(b, c, d) + e + w_i + K_j; \\
&\quad e = d; \\
&\quad c = b <<< 30; \\
&\quad b = a; \\
&\quad a = tmp; \\
&\}
\end{aligned}
$$

SHA-1 (160) continued

- 80 steps in main loop, changing Ks and Fs 4 times
  - Where $j = i/20$:

$$
\begin{array}{l}
\textbf{for}(\ i = 0;\ i < 80;\ i{+}{+}\ ) \qquad \{ \\
\qquad tmp = (a <<< 5) + F_j(b, c, d) + e + w_i + K_j; \\
\qquad e = d; \\
\qquad c = b <<< 30; \\
\qquad b = a; \\
\qquad a = tmp; \\
\}
\end{array}
$$

  - Each F_j combines three of the five variables:

$$
\begin{aligned}
F_0(X, Y, Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\
F_1(X, Y, Z) &= X \oplus Y \oplus Z \\
F_2(X, Y, Z) &= (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \\
F_3(X, Y, Z) &= X \oplus Y \oplus Z
\end{aligned}
$$

SHA-1 (160) continued

- 80 steps in main loop, changing Ks and Fs 4 times
  - Where $j = i/20$:

$$
\begin{array}{l}
\textbf{for}(\ i = 0;\ i < 80;\ i{+}{+}\ ) \quad \{ \\
\quad tmp = (a <<< 5) + F_j(b, c, d) + e + w_i + K_j; \\
\quad e = d; \\
\quad c = b <<< 30; \\
\quad b = a; \\
\quad a = tmp; \\
\}
\end{array}
$$

  - Each F_j combines three of the five variables:

$$
\begin{array}{rcl}
F_0(X, Y, Z) & = & (X \wedge Y) \vee (\neg X \wedge Z) \\
F_1(X, Y, Z) & = & X \oplus Y \oplus Z \\
F_2(X, Y, Z) & = & (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \\
F_3(X, Y, Z) & = & X \oplus Y \oplus Z
\end{array}
$$

  - Finally a, b, c, d, e are added to *tmp* (all addition is modulo 2^{32}).

SHA-1 (160) continued

- 80 steps in main loop, changing Ks and Fs 4 times
  - Where $j = i/20$:

$$
\begin{array}{l}
\textbf{for}(\ i = 0;\ i < 80;\ i{+}{+}\ ) \quad \{ \\
\quad tmp = (a <<< 5) + F_j(b, c, d) + e + w_i + K_j; \\
\quad e = d; \\
\quad c = b <<< 30; \\
\quad b = a; \\
\quad a = tmp; \\
\}
\end{array}
$$

  - Each F_j combines three of the five variables:

$$
\begin{array}{lcl}
F_0(X, Y, Z) & = & (X \wedge Y) \vee (\neg X \wedge Z) \\
F_1(X, Y, Z) & = & X \oplus Y \oplus Z \\
F_2(X, Y, Z) & = & (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \\
F_3(X, Y, Z) & = & X \oplus Y \oplus Z
\end{array}
$$

  - Finally a, b, c, d, e are added to *tmp* (all addition is modulo 2^{32}).
- **Exercise:** implement SHA-1 in your favourite language following this. Test against `sha1sum`.

Outline

Varieties of hash function

Properties of hash functions

Building hash functions

Standard hash functions

Conclusion

Current Status

- Hash functions are versatile and powerful primitive.
- However, difficult to construct and less researched than encryption schemes.
  - ideal hash function is a “random mapping” where knowledge of previous results doesn’t give knowledge of another.
  - practical fast iterative hash constructions fail this!
  - MD4 (1998), MD5 (1993/2005), SHA-1 (2005) are now **all considered broken**.
- The US National Institute of Standards and Technology (NIST) has standardised a set of newer hash functions.
  - Formerly called SHA-2, they are denoted by their output size: SHA-256, SHA-384, SHA-512.
  - However, since they are based upon the same *SHA* construction, they are not long-term solutions
  - In 2012, NIST awarded a new standard SHA-3 to the *Keccak* algorithm (a *sponge function* which has arbitrary output length).

References

- A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone, eds. *Handbook of Applied Cryptography*.
  CRC Press, 1997. Online: `http://www.cacr.math.uwaterloo.ca/hac`.

- Neils Ferguson and Bruce Schneier. *Practical Cryptography*.
  John Wiley & Sons, 2003.

- Douglas R Stinson. *Cryptography Theory and Practice*.
  CRC Press, second edition edition, 2002.

- Nigel Smart. *Cryptography: An Introduction*.
  McGraw-Hill, 2003. Third edition online: `http://www.cs.bris.ac.uk/~nigel/Crypto_Book/`

Recommended Reading

One of: Ch 9 of HAC (9.1–9.2); Ch. 10 of Smart 3rd Ed; 11.1–11.3 of Gollmann.