Integrity Checking and Secure Hash Functions

Dr. Demetrios Glinos
University of Central Florida

CIS3360 - Security in Computing

Readings

- "Computer Security: Principles and Practice", 3rd Edition, by William Stallings and Lawrie Brown
    - Section 21.1

Outline

- Integrity Checking
- Checksums
- Cyclic Redundancy Checks (CRCs)
- Hash Functions
- Secure Hash Functions
- The SHA Family of Algorithms

Integrity Checking

- Integrity checking is all about
  - *detecting changes* in files and messages
- Basic idea
  - Compute a numeric value for each message/file
  - Later, compute a fresh value to check for any unauthorized modifications
- Applications:
  - digital signatures
  - message authentication
  - file system integrity
- Numeric value computation
  - Checksum
  - Cyclic Redundancy Check (CRC)
  - Simple Hash Function
  - Cryptographic Hash function

Checksum

- Generally, a small value that is used for detecting errors or changes
- Many different algorithms can be used

- We will use the **Modular sum** algorithm:
  - Divide input into uniform words (e.g., 2 byte blocks) of the checksum size
  - **Add** all the words as **unsigned binary numbers**, **discarding any overflow bits**
  - **I**nterpret the **result** as a **two's complement number**
  - Use the **negation** of the **result** as the checksum value
  - **Append** the checksum value to the end of the input

  - **To validate**:
    - Divide into words and add them up in same manner, **including the checksum**
    - If result is not a word full of zeroes, then a change has occurred
    - **NOTE:** a single-bit error will be detected, but liklihood of 2 errors in the same column being undetected is 1/n, where n = #bits in word.

4-bit Checksum Example (1): message = B37F19

B		1	0	1	1
+ 3		0	0	1	1
=		**1**	**1**	**1**	**0**
+ 7		0	1	1	1
=	**1**	**0**	**1**	**0**	**1**
+ F		1	1	1	1
=	**1**	**0**	**1**	**0**	**0**
+ 1		0	0	0	1
=		**0**	**1**	**0**	**1**
+ 9		1	0	0	1
=		**1**	**1**	**1**	**0**

Note: we ignore the overflow bits as we go along (shown in red)

Checksum is two's complement of 1110, which is 0001 + 1 = 0010 = 2_{16}

So, what will be transmitted is the message, plus the checksum: **B37F192**

4-bit Checksum Example (2): validate B37F192

B		1	0	1	1
+ 3		0	0	1	1
=		1	1	1	0
+ 7		0	1	1	1
=	1	0	1	0	1
+ F		1	1	1	1
=	1	0	1	0	0
+ 1		0	0	0	1
=		0	1	0	1
+ 9		1	0	0	1
=		1	1	1	0
+ 2		**0**	**0**	**1**	**0**
=	1	**0**	**0**	**0**	**0**

← checksum value
← final result

Note: we ignore the overflow bits as we go along (shown in red)

Note: Since final result (ignoring overflow) is all zeroes, the message with checksum is validated

Cyclic Redundancy Check (CRC)

- A **Cyclic Redundancy Check (CRC)** is a checksum that is the remainder on division of the entire message by a polynomial.

- The coefficients of the polynomial are expressed as a binary value

- Examples
  - CRC-**8**: $x^{8} + x^5 + x^4 + x^2 + 1$ → 1 0 0 1 1 0 1 0 1
  - CRC-**4**: $x^{4} + x^2 + x$ → 1 0 1 1 0
  - CRC-**3**: $x^{3} + x^2 + 1$ → 1 1 0 1

- **NOTE:** the ***number*** of coefficients (i.e., number of bits) for the polynomial is always ***one more*** than the CRC word size (4 bits for CRC-3, 9 bits for CRC-8, etc.)

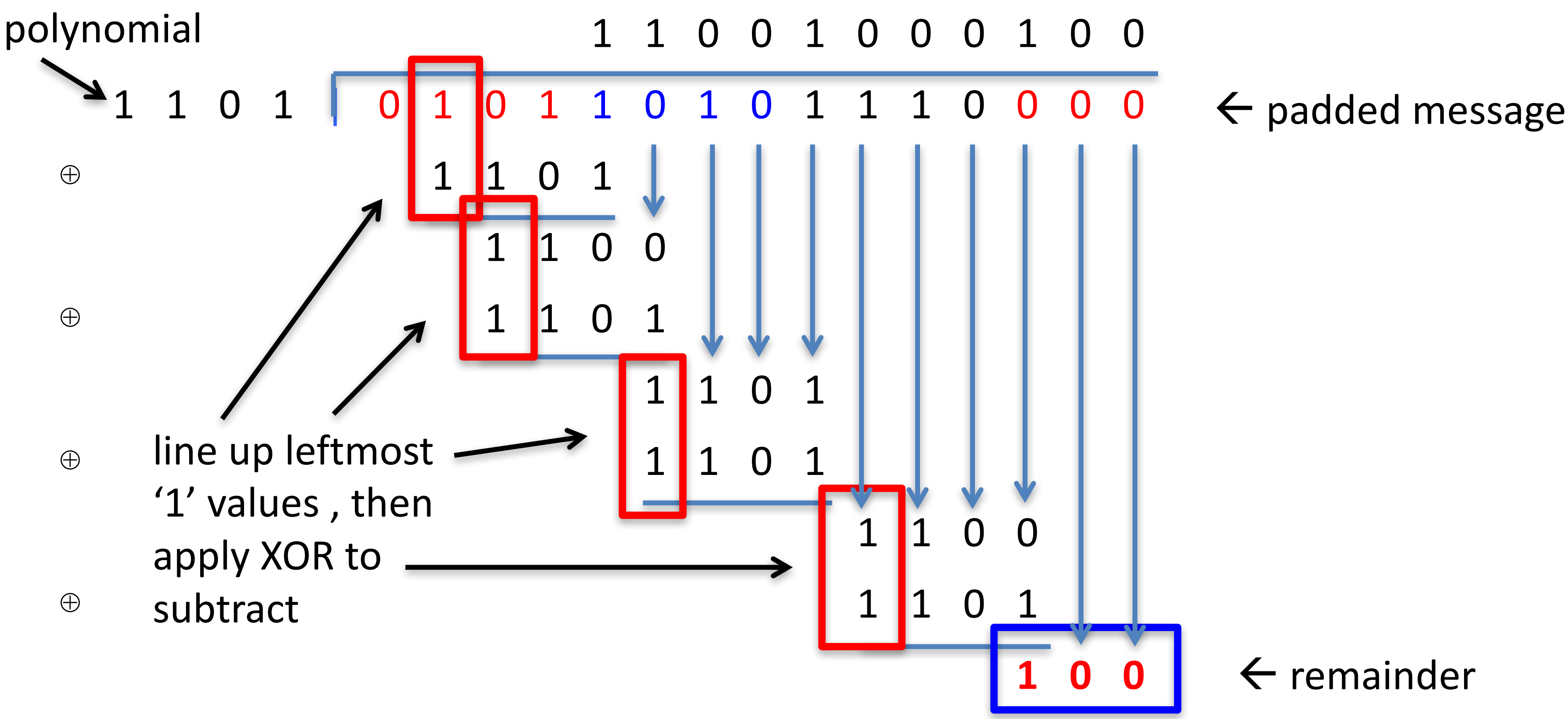
CRC Procedure

- To **generate a CRC:**
  1. **Pad** the message with **zeroes** (3 for CRC-3, 4 for CRC-4, etc.).
  2. **Divide** by the coefficients of the selected **polynomial**, using the **binary exclusive-or (XOR)** operator.
  3. **The remainder** is the CRC code.

p	q	p XOR q
1	1	0
1	0	1
0	1	1
0	0	0

- To **validate a CRC**:
  - Compute the CRC on the message **without** the checksum.
  - If the computed value is the **same** as the value received, no change has occurred.

CRC-3 Example: message=5AE, $x^3 + x^2 + 1$



Since we are doing CRC-3 and final result remainder is only 3 bits, we must ***pad*** it with a ***leading*** zero to get a 4-bit value. Our final CRC code for this message is **0100** which is $\mathbf{4}_{16}$, so the message with CRC appended is: $\mathbf{5AE4}_{16}$

Hash Functions

- A **hash function** is
  - An algorithm much like a cipher, **but**:
    - Takes **any size** data (such as a file) as input; and
    - Separates the input message/file into fixed-size blocks
    - Processes the input one block at a time, in order
    - Produces a **fixed size** output (the **hash value** or **hash code**)
      - – size of hash code is same as block size
      - – typically much smaller than the entire input message/file
  - hash value also commonly called the **message digest,** or simply the **digest**
- Properties
  - Easy to compute: **M → H(M), where H(M) is the digest of M.**

A Simple Hash Function

- **Bit-by-bit exclusive-OR (XOR)**
  - produces a simple parity for each bit position ($C_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{im}$)
  - also known as a longitudinal redundancy check
  - values for each bit position are computed separately
  - reasonably effective for random data
    - somewhat less effective for non-random data (e.g., text, where msb = 0)

	Bit 1	Bit 2	• • •	Bit n
Block 1	b_{11}	b_{21}		b_{n1}
Block 2	b_{12}	b_{22}		b_{n2}
	• • •	• • •	• • •	• • •
Block m	b_{1m}	b_{2m}		b_{nm}
Hash code	C_1	C_2		C_n

source: Fig.21.1

Bit-by-bit XOR Example: message = B37F19

To compute the bit-by-bit XOR of BE7F19 using a block size of 4 bits:

B	1	0	1	1
⊕ 3	0	0	1	1
=	**1**	**0**	**0**	**0**
⊕ 7	0	1	1	1
=	**1**	**1**	**1**	**1**
⊕ F	1	1	1	1
=	**0**	**0**	**0**	**0**
⊕ 1	0	0	0	1
=	**0**	**0**	**0**	**1**
⊕ 9	1	0	0	1
=	**1**	**0**	**0**	**0**

Hash value is 1000 = $\mathbf{8_{16}}$

So, what will be transmitted is the message, plus the hash value: **B37F198**

Secure Hash Functions

- A **secure hash function** (also called a **cryptographic hash function**) is a hash function that has these **additional** properties:

  - It is, for all practical purposes, **one-way**:

    - easy to compute **Y = H(M)**, but hard to find **M** given only **Y**.
    - that is, it is ***Infeasible,*** as a practical matter, to generate a message with a given hash value

  - It is, for all practical purposes, **collision-resistent:**

    - hard to find two messages, **M** and **N**, such that **H(M) = H(N)**.
    - i.e., ***Highly unlikely*** to find two different messages with the same hash value
    - that is, it is ***Infeasible,*** as a practical matter, to modify a message without changing the hash value

  - Examples: MD5, SHA family of algorithms

SHA Family of Algorithms

- **Secure Hash Algorithm (SHA)**
  - Developed by NIST, published in 1993 and revised as SHA-1 in 1995
  - 3 additional versions added in 2002: SHA-256, SHA-384, SHA-512
  - All versions share same basic structure
  - SHA-3 in development, presumably with different architecture

source: **Table 21.1 Comparison of SHA Parameters**

	SHA-1	SHA-256	SHA-384	SHA-512
Message digest size	160	256	384	512
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block size	512	512	1024	1024
Word size	32	32	64	64
Number of steps	80	64	80	80
Security	80	128	192	256

Notes: 1. All sizes are measured in bits.
2. Security refers to the fact that a birthday attack on a message digest of size n produces a collision with a work factor of approximately $2^{n/2}$.

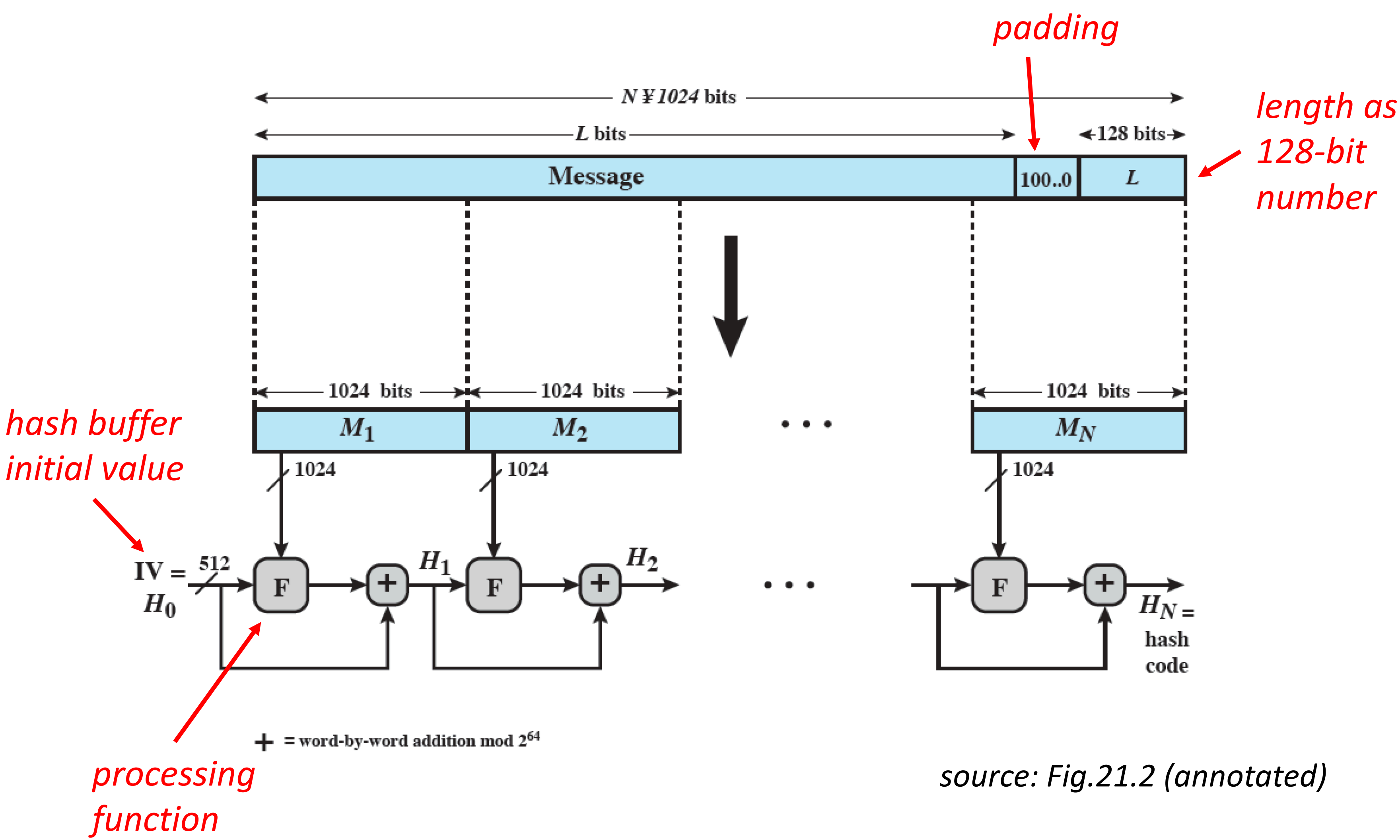
SHA-512 Message Digest Generation (1)

- SHA-512 computes a digest in 5 steps

1. Append padding bits so message length is congruent to 896 modulo 1024
2. Append length as a 128-bit unsigned binary number
3. Initialize a 512-bit hash buffer to certain fixed values
4. Process message in 1024-bit blocks
5. The message digest is the output after the last input block is processed

Note: the other SHA algorithms use a similar structure

SHA-512 Message Digest Generation (2)



source: Fig.21.2 (annotated)

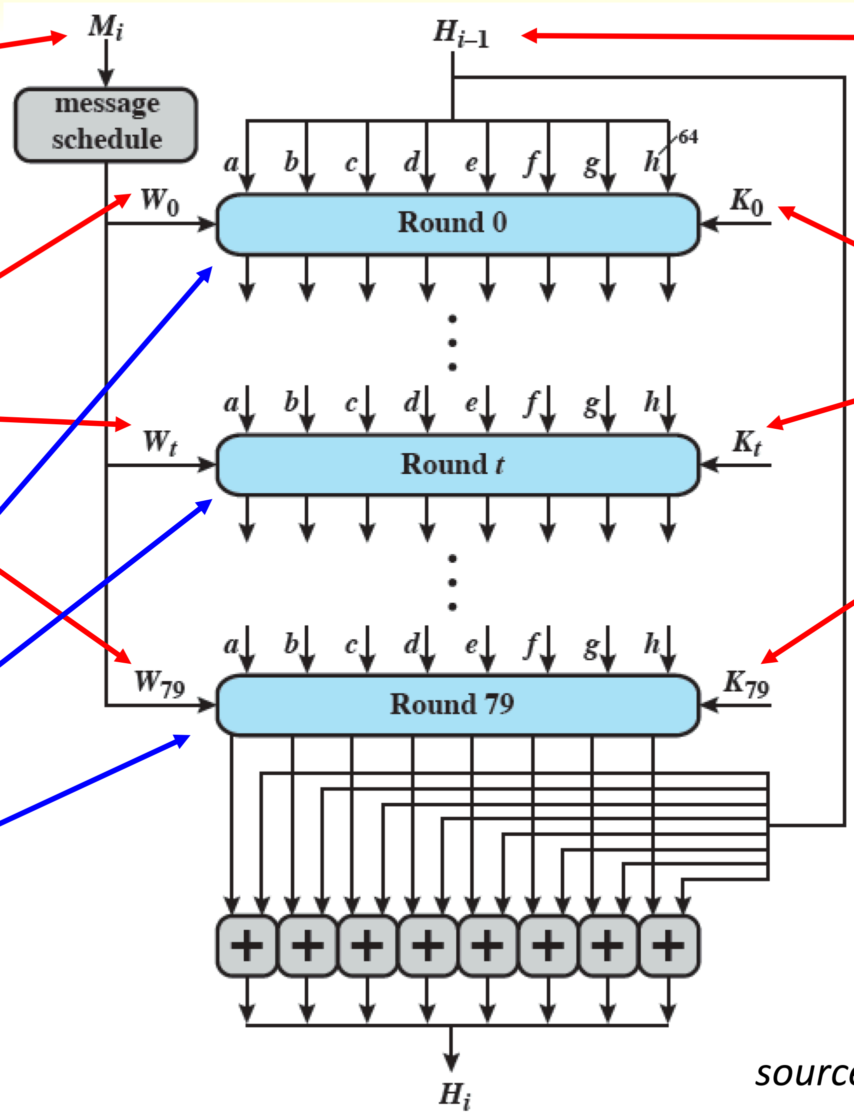
SHA-512 80-round Processing Function

1024-bit block

input buffer from previous round

64-bit values derived from M_i

The first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers



processing in each round consist of circular shifts and primitive Boolean AND, OR, NOT and XOR operations

Output of last round is added to input buffer as 8 separate additions mod 2^{64}

processing function

source: Fig.21.3 (annotated)