Lecture 6: SHA 256

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Secure Hash Algorithm

- Goal is to compute a unique hash value for any input "message", where a "message" can be anything.
- SHA-256 (widely used) returns a 256-bit hash value (a.k.a. message digest or strong checksum)

"The quick brown fox jumps over the lazy dog" → SHA-256 → 2fd4e1c6 7a2d28fc ed849ee1 …

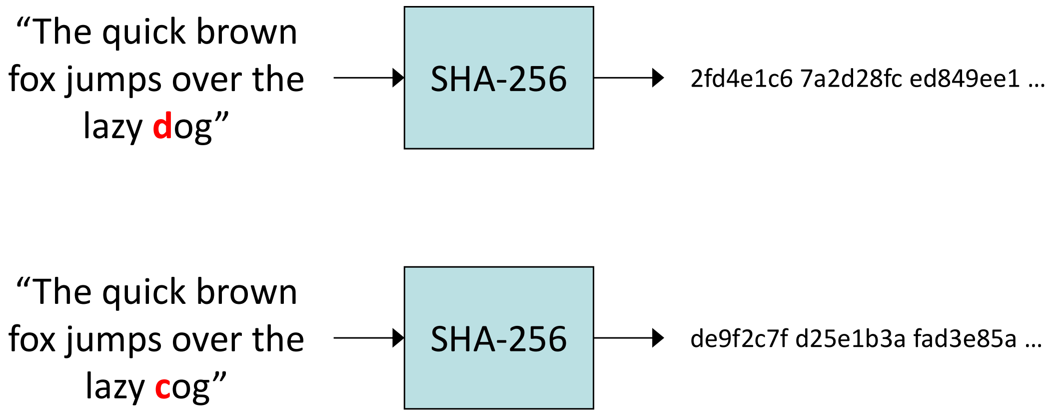
256-bits = eight32-bit words

file: avatar.avi → SHA-256 → some 256-bit value

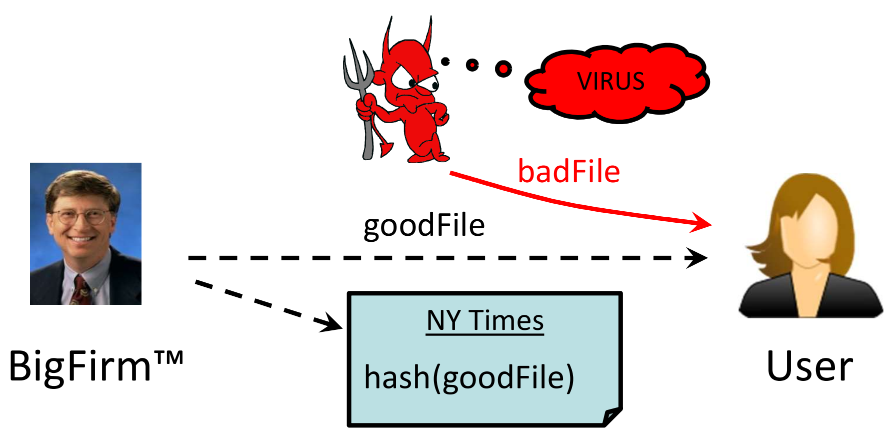
file: chopin.mp3 → SHA-256 → some 256-bit value

SHA-256

- Just a small change, e.g. from “dog” to “cog”, will completely change the hash value

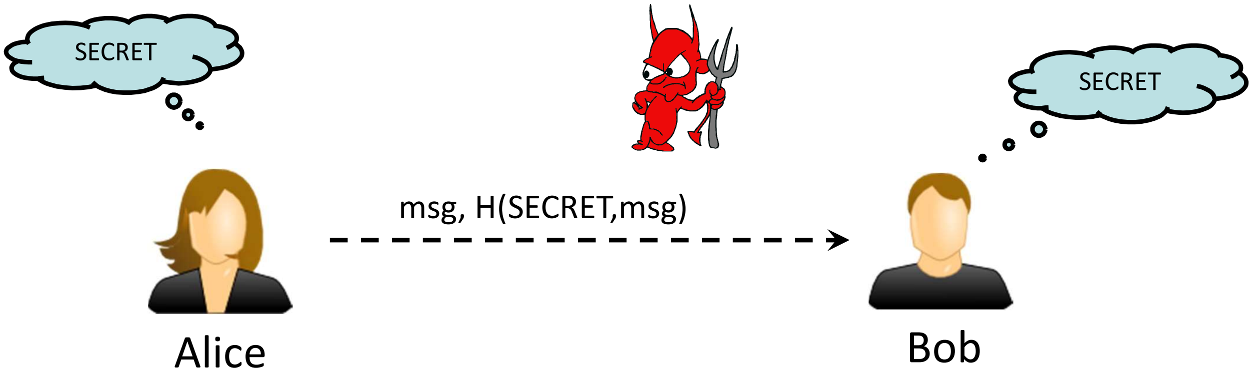


Verifying File Integrity



- Software manufacturer wants to ensure that the executable file is received by users without modification …
- Sends out the file to users and publishes its hash in NY Times
- The goal is <u>integrity</u>, not secrecy
- Idea: given goodFile and hash(goodFile), very hard to find badFile such that hash(goodFile)=hash(badFile)

Authentication



Alice wants to ensure that nobody modifies message in transit (both integrity and authentication)

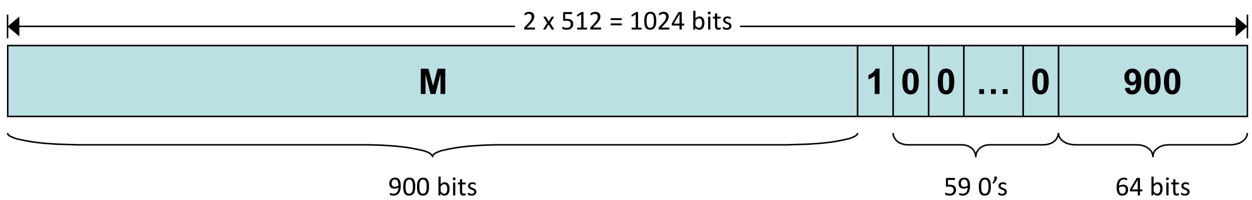
Idea: given msg,
very hard to compute H(SECRET, msg) without SECRET;
easy with SECRET

General Logic

- Input message must be < 2^{64} bits
  - not really a problem
- Message is processed in 512-bit blocks sequentially
- Message digest is 256 bits

SHA-256 Algorithm

- Step 1: Padding bits
  - A **b**-bit message **M** is padded in the following manner:
    - Add a single “1” to the end of **M**
    - Then pad message with “0’s” until the length of message is congruent to 448, modulo 512 (which means pad with 0’s until message is 64-bits less than some multiple of 512).
- Step 2: Appending length as 64 bit unsigned
  - A 64-bit representation of **b** is appended to the result of Step 1.
    - The resulting message is a multiple of 512 bits
    - e.g. suppose b = 900



SHA-256 Algorithm

- Step 3: Buffer initiation – initialize message digest (MD) to these eight 32-bit words

H_0 = 6a09e667
H_1 = bb67ae85
H_2 = 3c6ef372
H_3 = a54ff53a
H_4 = 510e527f
H_5 = 9b05688c
H_6 = 1f83d9ab
H_7 = 5be0cd19

SHA-256 Algorithm

- Step 4: Processing of the message (the algorithm)
  - Divide message M into 512-bit blocks, M_0, M_1, … M_j, …
  - Process each M_j sequentially, one after the other
  - Input:
    - W_t : a 32-bit word from the message
    - K_t : a constant array
    - H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7 : current MD
  - Output:
    - H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7 : new MD

SHA-256 Algorithm

- Step 4: Cont'd
  - At the beginning of processing each M_j, initialize
    (A, B, C, D, E, F, G, H) = (H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7)
  - Then 64 processing rounds of 512-bit blocks
  - Each step t ($0 \le t \le 63$): Word expansion for W_t
    - If $t < 16$
      - W_t = t^{th} 32-bit word of block M_j
    - If $16 \le t \le 63$
      - s_0 = (W_{t-15} **rightrotate** 7) **xor** (W_{t-15} **rightrotate** 18) **xor** (W_{t-15} **rightshift** 3)
      - s_1 = (W_{t-2} **rightrotate** 17) **xor** (W_{t-2} **rightrotate** 19) **xor** (W_{t-2} **rightshift** 10)
      - $W_t = W_{t-16} + s_0 + W_{t-7} + s_1$

SHA-256 Algorithm

- Step 4: Cont'd
  - K_t constants

    K [0..63] = 0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174, 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da, 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85, 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070, 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

SHA-256 Algorithm

- Step 4: Cont'd
  - Each step t ($0 \le t \le 63$):

    S_0 = (A **rightrotate** 2) **xor** (A **rightrotate** 13) **xor** (A **rightrotate** 22)

    maj = (A **and** B) **xor** (A **and** C) **xor** (B **and** C)

    $t_2 = S_0 + maj$

    S_1 = (E **rightrotate** 6) **xor** (E **rightrotate** 11) **xor** (E **rightrotate** 25)

    ch = (E **and** F) **xor** ((**not** E) **and** G)

    $t_1 = H + S_1 + ch + K_t + W_t$

    $(A, B, C, D, E, F, G, H) = (t_1 + t_2, A, B, C, D + t_1, E, F, G)$

SHA-256 Algorithm

- Step 4: Cont'd
  - Finally, when all 64 steps have been processed, set

    $H_0 = H_0 + A$

    $H_1 = H_1 + B$

    $H_2 = H_2 + C$

    $H_3 = H_3 + D$

    $H_4 = H_4 + E$

    $H_5 = H_5 + F$

    $H_6 = H_6 + G$

    $H_7 = H_7 + H$

SHA-256 Algorithm

- Step 5: Output
  - When all M_j have been processed, the 256-bit hash of M is available in H_0, H_1, H_2, H_3, H_4, H_5, H_6, and H_7

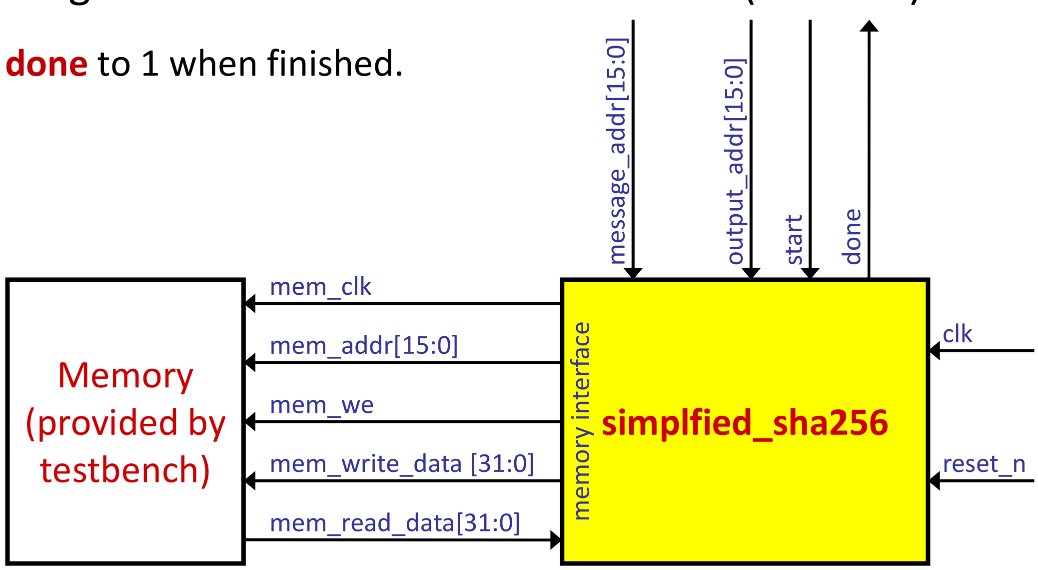
SHA-256 Algorithm

- More information can be found in the Wikipedia page

  https://en.wikipedia.org/wiki/SHA-2

Module Interface

- Wait in idle state for **start**, read message starting at **message_addr** and write final hash **{H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7}** in 8 words to memory starting at **output_addr**. **message_addr** and **output_addr** are word addresses.
- Message size is "hardcoded" to 20 words (640 bits).
- Set **done** to 1 when finished.



Module Interface

- Write the final hash **{H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7}** in 8 words to memory starting at **output_addr** as follows:

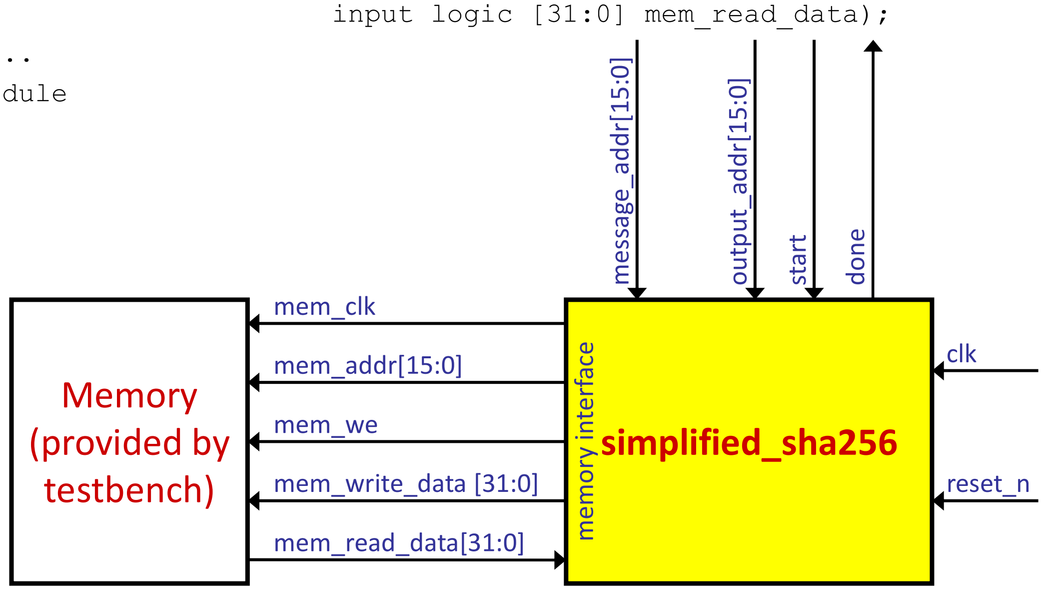
```
mem_addr <= output_addr;
mem_write_data <= H0;

mem_addr <= output_addr + 1;
mem_write_data <= H1;

...

mem_addr <= output_addr + 7;
mem_write_data <= H7;
```

output_addr	H_0
output_addr + 1	H_1
output_addr + 2	H_2
output_addr + 3	H_3
output_addr + 4	H_4
output_addr + 5	H_5
output_addr + 6	H_6
output_addr + 7	H_7

Module Interface

- Your assignment is to design the yellow box:

```
module simplified_sha256(input logic clk, reset_n, start,
                          input logic [15:0] message_addr, output_addr,
                         output logic done, mem_clk, mem_we,
                         output logic [15:0] mem_addr,
                         output logic [31:0] mem_write_data,
                          input logic [31:0] mem_read_data);
   ...
endmodule
```



Hints

- Since message size is hardcoded to 20 words, then there will be exactly 2 blocks.
- First block:
  - w[0]…w[15] correspond to first 16 words in memory
- Second block:
  - w[0]…w[3] correspond to remaining 4 words in memory
  - w[4] <= 32’80000000 to put in the “1” delimiter
  - w[5]…w[14] <= 32’00000000 for the “0” padding
  - w[15] <= 32’d640, since 20 words = 640 bits

Hints

- You must use “clk” as the “mem_clk”.

  assign mem_clk = clk

- Using “negative” phase of “clk” for “mem_clk” is not allowed.

Hints: Parameter Arrays

- Declare SHA256 K array like this:

```
// SHA256 K constants
parameter int sha256_k[0:63] = '{
   32'h428a2f98, 32'h71374491, 32'hb5c0fbcf, 32'he9b5dba5, 32'h3956c25b, 32'h59f111f1, 32'h923f82a4, 32'hab1c5ed5,
   32'hd807aa98, 32'h12835b01, 32'h243185be, 32'h550c7dc3, 32'h72be5d74, 32'h80deb1fe, 32'h9bdc06a7, 32'hc19bf174,
   32'he49b69c1, 32'hefbe4786, 32'h0fc19dc6, 32'h240ca1cc, 32'h2de92c6f, 32'h4a7484aa, 32'h5cb0a9dc, 32'h76f988da,
   32'h983e5152, 32'ha831c66d, 32'hb00327c8, 32'hbf597fc7, 32'hc6e00bf3, 32'hd5a79147, 32'h06ca6351, 32'h14292967,
   32'h27b70a85, 32'h2e1b2138, 32'h4d2c6dfc, 32'h53380d13, 32'h650a7354, 32'h766a0abb, 32'h81c2c92e, 32'h92722c85,
   32'ha2bfe8a1, 32'ha81a664b, 32'hc24b8b70, 32'hc76c51a3, 32'hd192e819, 32'hd6990624, 32'hf40e3585, 32'h106aa070,
   32'h19a4c116, 32'h1e376c08, 32'h2748774c, 32'h34b0bcb5, 32'h391c0cb3, 32'h4ed8aa4a, 32'h5b9cca4f, 32'h682e6ff3,
   32'h748f82ee, 32'h78a5636f, 32'h84c87814, 32'h8cc70208, 32'h90befffa, 32'ha4506ceb, 32'hbef9a3f7, 32'hc67178f2
};
```

- Use it like this:

```
tmp <= g + sha256_k[i];
```


Hints: Right Rotation

- Right rotate by 1

  {x[30:0], x[31]}

  ((x >> 1) | (x << 31))

- Right rotate by r

  ((x >> r) | (x << (32-r))

Hints: Right Rotation

```
// right rotation
function logic [31:0] rightrotate(input logic [31:0] x,
                                  input logic [ 7:0] r);
   rightrotate = (x >> r) | (x << (32-r));
endfunction
```


Testing

- Testbench: **tb_simplified_sha256.sv**