Message Authentication Methods, Functions, and Implementations

Çetin Kaya Koç
Cryptographic Hash Functions

- A cryptographic hash function provides message integrity and authentication.
- A function is used to compute a short `fingerprint' of the data; if the data is modified, the fingerprint will not be valid.
- \( h \) is the hash function, and \( x \) is the data.
- The fingerprint is defined as \( y=h(x) \).
- The fingerprint \( y \) is called as `message digest'; it is fairly short, e.g., 160 bits.
Cryptographic Hash Functions

- We assume that $y$ is stored in a safe place, but $x$ is publicly accessible.
- If $x$ is changed to $x'$, we hope that $h(x')$ is different from $y$.
- Therefore, we can detect the change in $x$ by re-computing the message digest of $x'$ and checking if $h(x) \neq h(x')$.
- Message digest functions are also used in digital signatures.
Keyed Hash Functions

- Keyed hash functions are also useful; they are often called message authentication codes (MAC).
- A & B are sharing a secret key K, used as an index in the keyed hash function to compute the hash value of x as $y = h(K,x)$.
- The pair $(x,y)$ can be sent by A to B; B verifies the authenticity $y = h(K,x)$ and becomes confident that neither is changed provided that the hash function is secure.
Unkeyed and Keyed Hash Functions

- The assurance of data integrity are different
- Unkeyed hash functions: the message digest value $y$ must be securely stored so that it cannot be changed by an unauthorized party
- Keyed hash function: the key $K$ must be kept secret; $x$ & $y$ can be sent over on insecure channel
Security of Hash Functions

- Given a hash function $y = h(x)$, the following security requirements are desired due to their applications in cryptographic protocols:
  - One-way or Preimage Resistance
  - Second Preimage Resistance
  - Collision Resistance
One-way or Preimage Resistance

- Given a message digest $y$, the problem of preimage computation is the computation of $x$ such that $y=h(x)$
- A function for which a preimage problem cannot be efficiently solved is called a one-way function or a preimage resistant function
Second Preimage Resistance

- Given a message $x$, the problem of second preimage computation is the computation of $x'$ (which is not equal to $x$) such that $h(x') = h(x)$
- A hash function for which the second preimage computation cannot be efficiently done is called second preimage resistant
Collision Resistance

- The problem of collision is the computation of a pair of $x$ & $x'$ (which are not equal) such that $h(x') = h(x)$
- If such a valid pair is found, then we have detected a collision: $(x,y)$ is valid pair so is $(x',y)$
- A hash function for which the collision problem cannot be efficiently solved is called *collision resistant*
Generic Attacks to Hash Functions

- These attacks depend only on the bit size of the hash value $y$, and are independent of the specific properties of the algorithm.

- It is generally assumed that the hash function approximates a random function, otherwise these attacks will be even more successful.

  - Random Second Preimage Attack
  - Birthday Attack
Random Second Preimage Attack

- The attacker selects a random message $x'$ and hopes that the given hash value is hit: $h(x') = y$
- The probability of success is $2^{(-n)}$ if the bit size of $y$ is $n$
- The attack can be carried out off-line and in parallel
- Therefore, the bit size $n$ should be sufficiently large to circumvent this attack: 64, 80, 128, 160, 256, etc
Birthday Attack

- This attack attempts to find any two $x$ and $x'$ such that their hash values are equal $y = h(x) = h(x')$.
- This problem is related to finding two people with the same birthday (any year; for example, two people born on October 9 albeit different years).
- The probability of success is much higher than $1/365$. 
Birthday Attack Probability

- The probability that the birthday of the first person in a specific day of the year is equal to $1/365$
- The probability that the birthday of the second person is NOT the same as the first person is $(1 - 1/365) = 364/365$
- If the birthdays of the first two people are different, the probability that birthday of the third person is different from the first two people is $(1 - 2/365) = 363/365$
- Therefore, the probability that all $T$ people have different birthdays is $P = (364/365) \times (363/365) \times \cdots \times ((366-T)/365)$
Birthday Attack Probability

- The probability of having two people with the same birthday in a room of T people as $1 - P$
- For example, if there are 10 people in the room, this probability is found as 0.12
- If there are 23 people, the probability of having two people with the same birthday is found to be 0.51 > 50%
- Intuitively, one expects lower probability, however, there $23 \times 22 / 2 = 253$ pairs of people
An adversary generates $r$ variations on a bogus message and $r$ variations on a genuine message.

The probability of finding a bogus message and a genuine message which have the same hash value is given as

$$1 - \exp(-r^2/2^n)$$

where $2^n$ is the number of hash values.

If $r = 2^{(n/2)}$, then this probability is found as

$$1 - e^{-1} = 63\%$$

Therefore, hash functions with output hash value of less than 128 bits are not secure: $2^{64}$ tries.
Message Digest Functions

There are essentially three classes of MDC functions:
- MDC functions based on block ciphers
- MDC functions based on algebraic structures
- Custom-designed MDC functions
MDCs based on Block Ciphers

- For historical reasons (due to DES), such MDC were used and continued to be used
- Minimal design effort (block cipher is already available)
- Existing hardware/software can be used
- Trust in the block cipher is a factor
- DES-based systems do not offer long-term security due to the bit length – succumbs to birthday attack
MDCs based on Algebra

- Modular arithmetic is a popular choice.
- Ad-hoc schemes without security proofs (its security does not reduce to a known intractable problem).
- Such systems are more efficient, but many proposed schemes are broken.
- Provable schemes are based on RSA, modular squaring etc.
- There are also knapsack or lattice based schemes.
Custom-Designed MDC Functions

- Based on the iterative application of a compression function
- Employs 32-bit arithmetic/logic operations for speed on software
- Several efficient systems have been proposed
- The surviving members are found in the current US and European standards
Design of Hash Functions

- There is a widely accepted general model of hash functions based on iterative application of a compression function.
- Compression function has fixed input size and process every block the same way.
- The iterated hash function repeatedly uses the compression function in order to produce the final hash value.
- The message is broken into equal size blocks, each of which is applied to the compression function (message is also padded).
General Hash Function Model

\[ H_0 = IV \]
\[ H_i = f(x_i, H_{i-1}) \]
\[ h(x) = g(H_t) \]

- IV: a fixed initial value, same for all messages
- f: the compression function (round function)
- g: Output transformation
Some Security Considerations

- The choice of IV is important
  - IV should be defined as part of the description of the hash function

- The choice of padding rule is important
  - Padding rule should be unambiguous
  - At the end, one should append the length of the message

- Deviations from these rules will make the hash function less secure
MD5, SHA-1 → new SHAs

- MD5 was proposed by Rivest, part of RSA Security PKCS
  - MD5: 128-bit message digest function
- SHA-1 was proposed by NIST, together with DSA
  - SHA-1: 160-bit message digest function
- MD5 and SHA are based on the same principles
- MD5 may not be considered secure anymore due to its length: 128 bits
MD5, SHA-1 → new SHAs

- SHA-1 is 160 bits .. still fine
- NIST introduced 3 new SHA functions
  - SHA-256, SHA-384, and SHA-512
- They are not direct generalizations of SHA
- Based on some new methods and constructs
- Standardized on Aug 2002 (FIPS 180-2)
  - SHA-1, SHA-256, SHA-384, SHA-512
- Some security issues
  - More security analyses may be needed
  - Usage of truncated hashes needs clarification
MD5, SHA-1 \rightarrow new SHAs

Properties of SHA functions

<table>
<thead>
<tr>
<th></th>
<th>Message</th>
<th>Block</th>
<th>Word</th>
<th>Digest</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>$2^{64}$</td>
<td>512</td>
<td>32</td>
<td>160</td>
<td>80</td>
</tr>
<tr>
<td>SHA-256</td>
<td>$2^{64}$</td>
<td>512</td>
<td>32</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>SHA-384</td>
<td>$2^{128}$</td>
<td>1024</td>
<td>64</td>
<td>384</td>
<td>192</td>
</tr>
<tr>
<td>SHA-512</td>
<td>$2^{128}$</td>
<td>1024</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
</tbody>
</table>
Attacks on Iterated Hash Functions

- **Meet-in-the-middle attack**
  - A variation of the birthday attack
  - We compare intermediate chaining variables instead of the final hash value
  - More advanced versions of the attack were also developed: p-fold iterated schemes

- **Fixed point attack**
  - Tries to look for intermediate values such that
    \[ f(x_i, H_{i-1}) = H_{i-1} \]
  - It is possible to insert an arbitrary number of data blocks without modifying the final hash
A Security Analysis - Case Study

- An iterated hash function employed in a popular product
- I was asked to provide a security analysis
- Iteration uses two compression functions
  - F: input size 60x1024 bits
    output size 16 bits
  - G: input size 60x1024 bits
    output size 32 bits
- Message is broken into 60kb blocks
A Case Study

- First, each message block is independently hashed using F to obtain

\[ M_1M_2...M_L \] message

\[ A_1A_2...A_L \] such that \( A_i = F(M_i) \)

- Then, chaining is applied using G

\[ C_0 = \text{a fixed initial 32-bit value} \]
\[ C_i = G(C_{i-1}, M_i) \text{ for } i = 1,2,...,L \]

Final hash value: \( A_1A_2A_3...A_LC_L \)
A Case Study

- Even though the final hash value is $16L + 32$ bits long, where $L$ the number of message blocks, the hash function is not secure because a meet-in-the-middle attacks can be devised.

- Consider the composite compression function $Z$ such that

$$A_i | C_i = Z(C_{i-1}, M_i)$$

- The input size of $Z$ is $32 + 60k$ bits while the output size is $16 + 32 = 48$ bits.
A Case Study

- Given a message of $L$ blocks, it is possible to modify the first block without changing the hash values:

  $$M_1M_2\ldots M_L \quad A_1A_2\ldots A_LC_L$$

  $$M'_1M_2\ldots M_L \quad A_1A_2\ldots A_LC_L$$

  $$A_1 \mid C_1 = Z(C_0, M_1) = Z(C_0, M'_1)$$

- Finding a collision in the 48-bit compression function of $Z$ is quite an easy task; by trying $2^{24}$ different messages, we have a success probability of 63%
SHA-256 on Pentium 4

- Optimization Techniques:
  - Loop unrolling and renaming registers
  - Redefining Boolean functions
  - Pre-fetching data
  - Replacing rotation instructions
  - Using SIMD instructions
  - Instruction scheduling
  - Reducing data dependencies
  - Using simple instructions
  - Reducing memory accesses
  - Avoiding unnecessary work
Loop Unrolling

- Eliminates:
  - Loop overhead
  - Index calculations
  - Shifting registers in each iteration

- Allows us to rename the registers instead of shifting them in each iteration

- Elimination of shifting phase:
  - Main computational gain
  - Needs unrolling the loops by a factor of the number of state variables

- Full unrolling releases one register and eliminates a branch miss prediction
Loop Unrolling

However, there is a tradeoff:
- reduced computation vs. increased code size
  - Increased code size causes more cache misses and page faults.
  - Most papers we have read on SHA software implementations claim that loop unrolling gives better results

Truth:
- It is better not to unroll the loops. We can prevent latencies caused by shifting registers by doing careful instruction scheduling
Redefining Boolean Functions

- We can reduce the number of operations!!

<table>
<thead>
<tr>
<th>Boolean Functions</th>
<th>Original Definition</th>
<th>Modified Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Ch(x, y, z) )</td>
<td>((x \land y) \oplus (\neg x \land z))</td>
<td>((z \oplus (x \land (y \oplus z))))</td>
</tr>
<tr>
<td>( Maj(x, y, z) )</td>
<td>((x \land y) \oplus (x \land z) \oplus (y \land z))</td>
<td>((y \land (x \lor z)) \lor (x \land z))</td>
</tr>
<tr>
<td>( Parity(x, y, z) )</td>
<td>(x \oplus y \oplus z)</td>
<td>SAME</td>
</tr>
</tbody>
</table>

- We save one operation for each boolean function
Pre-fetching Data

- Reduces the effect of data transfer latencies by overlapping hash computations
- PREFETCHh instruction:
  - Loads data from memory to a selected level of cache
  - Introduced in Pentium 3 processors as a part of SSE instruction set
Replacing Rotation Instructions

Rol/Ror instruction has a longer latency on the Pentium 4 processor than on previous Pentium processor generations

mov reg2, reg1
shl reg1, imm
shr reg2, 32-imm is better than rol reg1, imm
or reg1, reg2
Using SIMD Instructions

- Pentium 4 CPU has Single Instruction Multiple Data (SIMD) instruction sets: MMX, SSE and SSE2
- We can use SIMD instructions to improve the performance of SHA by hashing more than one stream simultaneously
- The result of our research: The throughput of SHA can be improved on P4 by 130% using SIMD instructions

*
Well-Known Optimization Techniques

- Instruction scheduling
- Reducing data dependencies
- Using simple instructions
- Reducing memory accesses
- Avoiding unnecessary work
Instruction Scheduling (IS)

Example:

```
add ecx, eax
add ebx, 4
add ecx, esi
cmp ebx, 256
mov ds:[0], ecx
jl LoopBegin
```

is better than

```
add ecx, eax
add ecx, esi
mov ds:[0], ecx
add ebx, 4
cmp ebx, 256
jl LoopBegin
```

We reduced the data dependencies between the first 3 lines and between lines 4 and 5.
Using Simple Instructions

- Complex instructions reduce the performance.
- Instead of using a complex instruction, using a few simple instructions gives more room for IS and usually increases the performance.

```
mov   eax, ds:[0]
add   eax, ebx   ✡ is better than ✡ add ds:[0], ebx
mov   ds:[0], eax
```
Reducing Memory Accesses

- Example:
  
  ```c
  tmp = (unsigned long *) msg;
  i = (unsigned long)(tmp + 512);
  for(; tmp < (unsigned long *)i; tmp+=16){
    rct_SHA256_transform(sha256_session.state, tmp);
  }
  ```

  is better than

  ```c
  for (i=0; i<512; i+=16) {
    for (j=0; j<16; ++j) {
      block[ j ] =msg[ i + j ];
    }
    rct_SHA256_transform(sha256_session.state, block);
  }
  ```
Performance Results

<table>
<thead>
<tr>
<th></th>
<th>From Main Memory</th>
<th>From Hard Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>87.12 MB/s</td>
<td>67.08 MB/s</td>
</tr>
<tr>
<td>SHA-256</td>
<td>48.88 MB/s</td>
<td>37.64 MB/s</td>
</tr>
<tr>
<td>SHA-512</td>
<td>38.26 MB/s</td>
<td>30.60 MB/s</td>
</tr>
</tbody>
</table>

- **Platform:**
  - 2.4 GHz Pentium-4
  - 256 MB of main memory
  - Windows XP
## Performance Results (SIMD)

<table>
<thead>
<tr>
<th>Number of Streams</th>
<th>Architecture</th>
<th>SHA-1</th>
<th>SHA-256</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput</td>
<td>Speedup</td>
<td>Throughput</td>
</tr>
<tr>
<td>1</td>
<td>32-bit</td>
<td>87.12 MB/s</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>64-bit (MMX)</td>
<td>128.46 MB/s</td>
<td>1.474</td>
</tr>
<tr>
<td>4</td>
<td>128-bit (XMM)</td>
<td>143.09 MB/s</td>
<td>1.642</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Streams</th>
<th>Architecture</th>
<th>SHA-384 &amp; SHA-512</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Throughput</td>
<td>Speedup</td>
</tr>
<tr>
<td>1</td>
<td>32-bit</td>
<td>38.26 MB/s</td>
</tr>
<tr>
<td>1</td>
<td>64-bit (MMX)</td>
<td>55.71 MB/s</td>
</tr>
<tr>
<td>2</td>
<td>128-bit (XMM)</td>
<td>88.02 MB/s</td>
</tr>
</tbody>
</table>
References