Chapter 8 Hash Functions
A hash function maps a variable-length message into a fixed-length hash value, or message digest.
Key Points

1. A hash function maps a variable-length message into a fixed-length hash value, or message digest.
2. Virtually all cryptographic hash functions involve the iterative use of a compression function.
1. A hash function maps a variable-length message into a fixed-length hash value, or message digest.

2. Virtually all cryptographic hash functions involve the iterative use of a compression function.

3. The compression function used in secure hash algorithms falls into one of two categories: a function specifically designed for the hash function or an algorithm based on a symmetric block cipher. SHA and Whirlpool are examples of the two approaches, respectively.
Definition

A hash function $H$ accepts a variable-length block of data $M$ as input and produces a fixed-length hash value $h = H(M)$. A ‘good’ hash function has the property that the results of applying the function to a large set of inputs will produce outputs that are evenly distributed and apparently random.

A hash function $H$ accepts a variable-length block of data $M$ as input and produces a fixed-length hash value $h = H(M)$. A ‘good’ hash function has the property that the results of applying the function to a large set of inputs will produce outputs that are evenly distributed and apparently random.

In general terms, the principle object of the hash function is data integrity. A change to any bit or bits of $M$ results (with high probability) in a change to the hash code.
The kind of hash function needed for security applications is referred to as a **cryptographic hash function**. A cryptographic hash function is an algorithm for which it is computationally infeasible (because no attack is significantly better than brute force) to find either

(a) a data object that maps to a pre-specified hash result (the one-way property); or

(b) two data objects that map to the same hash result (the collision-free property).
The kind of hash function needed for security applications is referred to as a cryptographic hash function. A cryptographic hash function is an algorithm for which it is computationally infeasible (because no attack is significantly better than brute force) to find either

(a) a data object that maps to a pre-specified hash result (the one-way property); or

(b) two data objects that map to the same hash result (the collision-free property).

Because of the characteristics, hash functions are often used to determine whether or not data has changed.
Typically, the input is padded out to an integer multiple of some fixed length (i.e. 1024 bits), and the padding includes the value of the length of the original message in bits. The length field is a security measure to increase the difficulty for an attacker to produce an alternative message with the same hash value.

Message of data block $M$ (variable length) $L$

$H$

$h$

Hash Value $h$ (Fixed Length)
Message Authentication

Message authentication is a mechanism or service used to verify the integrity of the message. Message authentication assures that data received are exactly as sent. In many cases, there is a requirement that the authentication mechanism assures that alleged identity of a sender is valid. When a hash function is used to provide message authentication, the hash function value is often referred to as a message digest.
The message plus concatenation (link in a series or chain) hash code is encrypted using symmetric encryption. Because only A and B share the secret key, the message has not been altered. The hash code provides the structure for redundancy required to achieve authentication. Because encryption is applied to the entire message plus the hash code, confidentiality is also provided.
Hash Functions for Message Authentication

Source A

\[ E(K, [M||H(M)]) \]
Hash Functions for Message Authentication

Source A

\[ \text{Source A} \]

\[ M \xrightarrow{H} E(K, [M||H(M)]) \]

Destination B

\[ \text{Destination B} \]

\[ E(K, [M||H(M)]) \xrightarrow{K} \]

\[ H(M) \xrightarrow{\text{Compare}} \]
Only the hash code is encrypted, using a symmetric encryption. This reduces the processing burden for those applications that do not require confidentiality.
Hash Functions for Message Authentication

Source A

\[ M \rightarrow H \rightarrow || \rightarrow E \rightarrow K \rightarrow E(K, H(M)) \]

Destination B
Hash Functions for Message Authentication

Source A

\[ M \rightarrow H \rightarrow E \left( K, H(M) \right) \rightarrow M \]

Destination B

\[ M \rightarrow H \left( E(K, H(M)) \right) \rightarrow D \rightarrow \text{Compare} \]

\[ E(K, H(M)) \rightarrow K \]
It is possible to use a hash function but no encryption for message authentication. The technique assumes that the two communicating parties share a common secret value $S$. A computes the hash value over the concatenation of $M$ and $S$ and appends the resulting hash value to $M$. Because B possesses $S$, he can recompute the hash value to verify. Because the secret value itself is not sent, an opponent cannot modify an intercepted message and cannot generate a false message.
Hash Functions for Message Authentication

Source A

\[ H(M || S) \]
Hash Functions for Message Authentication

Source A

\[ H(\mathbf{M} \| \mathbf{S}) \]

Destination B

\[ H(\mathbf{M} \| \mathbf{S}) \]

Compare
An Example

\[ E(K, H(M)) \] is a function of variable length message \( M \) and a secret key \( K \).
Confidentially can be added to the approach of (c) by encrypting the entire message plus the hash code.
Hash Functions for Message Authentication

\[ E(K, [M || H(M || S)]) \]
Hash Functions for Message Authentication

Source A

\[ E(K, [M || H(M || S)]) \]

\[ E(K, [M || H(M || S)]) \]  

\[ H(M || S) \]

Destination B

\[ E(K, [M || H(M || S)]) \]  

\[ H(M || S) \]  

Compare
One Final Note On These Methods

When confidentiality is not required, method (b) has an advantage over methods (a) and (d), which decrypt the entire message, in that less computation is required.
Why Avoid Encryption?

There has been growing interest in techniques that avoid encryption. Reasons for this are

1. Encryption software is relatively slow. Even though the amount of data encrypted per message may be small, there may be a steady stream of messages into and out of a system.
Why Avoid Encryption?

There has been growing interest in techniques that avoid encryption. Reasons for this are

1. Encryption software is relatively slow. Even though the amount of data encrypted per message may be small, there may be a steady stream of messages into and out of a system.

2. Encryption hardware costs are not negligible. Low cost chip implementations of DES are available, but the cost adds up if all nodes in a network must have the capability.
Why Avoid Encryption?

There has been growing interest in techniques that avoid encryption. Reasons for this are

1. Encryption software is relatively slow. Even though the amount of data encrypted per message may be small, there may be a steady stream of messages into and out of a system.

2. Encryption hardware costs are not negligible. Low cost chip implementations of DES are available, but the cost adds up if all nodes in a network must have the capability.

3. Encryption hardware is optimized towards large data samples. For small blocks of data, a high proportion of the time is spent in the initialization/invocation (help support) overhead.
Why Avoid Encryption?

There has been growing interest in techniques that avoid encryption. Reasons for this are

1. Encryption software is relatively slow. Even though the amount of data encrypted per message may be small, there may be a steady stream of messages into and out of a system.

2. Encryption hardware costs are not negligible. Low cost chip implementations of DES are available, but the cost adds up if all nodes in a network must have the capability.

3. Encryption hardware is optimized towards large data samples. For small blocks of data, a high proportion of the time is spent in the initialization/invocation (help support) overhead.

4. Encryption algorithms may be covered by patents, and there is a cost associated with licensing their use.
A more commonly used message authentication is achieved by using message authentication code (MAC), also known as a keyed hash function.
A more commonly used message authentication is achieved by using message authentication code (MAC), also known as a keyed hash function.

Typically, MAC’s are used between parties that share a secret key to authenticate information exchanged. A MAC function takes as input a secret key and a data block and produces a hash value, referred to as the MAC. This can be transmitted or stored with the protected message.
A more commonly used message authentication is achieved by using message authentication code (MAC), also known as a keyed hash function.

Typically, MAC’s are used between parties that share a secret key to authenticate information exchanged. A MAC function takes as input a secret key and a data block and produces a hash value, referred to as the MAC. This can be transmitted or stored with the protected message.

If the integrity needs to be checked, the MAC function can be applied to the message and the result compared to the stored MAC value. An attacker altering the message will be unable to alter the MAC value without knowledge of the secret key.
A more commonly used message authentication is achieved by using message authentication code (MAC), also known as a keyed hash function.

Typically, MAC’s are used between parties that share a secret key to authenticate information exchanged. A MAC function takes as input a secret key and a data block and produces a hash value, referred to as the MAC. This can be transmitted or stored with the protected message.

If the integrity needs to be checked, the MAC function can be applied to the message and the result compared to the stored MAC value. An attacker altering the message will be unable to alter the MAC value without knowledge of the secret key.

In practice, specific MAC algorithms are designed that are generally more efficient than an encryption algorithm.
The operation of the digital signature is similar to that of the MAC. In the case of the digital signature, the hash value of the message is encrypted with a user’s private key. Anyone who knows the user’s public key can verify the integrity of the message that is associated with the digital signature. In this case, an attacker who wishes to alter the message would need to know the user’s private key.
This (simplified) view of a hash code is used to provide a digital signature.
Our Visual

This (simplified) view of a hash code is used to provide a digital signature.

Source A

Extra Encryption If Privacy Is Required
Our Visual Destination B

\[ E(\text{PR}_a, H(M)) \]

\[ PU_a \]

Compare
Hash functions are commonly used to create a **one-way password file**: hash of a password is stored by an operating system rather than the password itself. This way, the actual password is not retrievable by a hacker who gains access to the password file. That is, when a user enters a password, the hash of the password is compared to the stored hash value for verification. This approach to password verification is used by most operating systems.

Hash functions can be used for intrusion detection or virus detection. Store $H(F)$ for each file on a system and secure the hash files (for example, on a thumb drive that is kept secure). One can later determine if a file has been modified by recomputing $H(F)$. An intruder would need to change $F$ without changing $H(F)$. To get a feel for the security considerations involved in cryptographic hash functions, we will look at two simple insecure hash functions.
Hash functions are commonly used to create a one-way password file: hash of a password is stored by an operating system rather than the password itself. This way, the actual password is not retrievable by a hacker who gains access to the password file. That is, when a user enters a password, the hash of the password is compared to by the stored hash value for verification. This approach to password verification is used by most operating systems.

Hash functions can be used for intrusion detection or virus detection. Store $H(F)$ for each file on a system and secure the hash files (for example, on a thumb drive that is kept secure). One can later determine if a file has been modified by recomputing $H(F)$. An intruder would need to change $F$ without changing $H(F)$. 
Other Applications

Hash functions are commonly used to create a one-way password file: hash of a password is stored by an operating system rather than the password itself. This way, the actual password is not retrievable by a hacker who gains access to the password file. That is, when a user enters a password, the hash of the password is compared to by the stored hash value for verification. This approach to password verification is used by most operating systems.

Hash functions can be used for intrusion detection or virus detection. Store $H(F)$ for each file on a system and secure the hash files (for example, on a thumb drive that is kept secure). One can later determine if a file has been modified by recomputing $H(F)$. An intruder would need to change $F$ without changing $H(F)$.

To get a feel for the security considerations involved in cryptographic hash functions, we will look at two simple insecure hash functions.
All hash functions use the following basic principles:

- The input (message, file, etc.) is viewed as a sequence of $n$-bit blocks.
All hash functions use the following basic principles:

- The input (message, file, etc.) is viewed as a sequence of $n$-bit blocks.
- The input is processed one block at a time in an iterative fashion to produce an $n$-bit hash function.
One of the simplest hash functions is the bit-by-bit exclusive-or (XOR) of every block. This can be expressed as

\[ c_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{im} \]
One of the simplest hash functions is the bit-by-bit exclusive-or (XOR) of every block. This can be expressed as

\[ c_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{im} \]

- \( c_i \) is the \( i^{th} \) bit of the hash code, \( 1 \leq i \leq n \)
- \( m \) is the number of \( n \)-bit blocks in the input
- \( b_{ij} \) is the \( i^{th} \) bit in the \( j^{th} \) block
One of the simplest hash functions is the bit-by-bit exclusive-or (XOR) of every block. This can be expressed as

\[ c_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{im} \]

- \( c_i \) is the \( i^{th} \) bit of the hash code, \( 1 \leq i \leq n \)
- \( m \) is the number of \( n \)-bit blocks in the input
- \( b_{ij} \) is the \( i^{th} \) bit in the \( j^{th} \) block

This operation produces a simple parity for each bit position and is known as a longitudinal redundancy check. Each \( n \)-bit hash value is equally likely. So, the probability that a data error will result in an unchanged hash value is \( 2^{-n} \).
Parity and *XOR*

One of the simplest hash functions is the bit-by-bit exclusive-or (XOR) of every block. This can be expressed as

\[ c_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{im} \]

- \(c_i\) is the \(i^{th}\) bit of the hash code, \(1 \leq i \leq n\)
- \(m\) is the number of \(n\)-bit blocks in the input
- \(b_{ij}\) is the \(i^{th}\) bit in the \(j^{th}\) block

This operation produces a simple parity for each bit position and is known as a longitudinal redundancy check. Each \(n\)-bit hash value is equally likely. So, the probability that a data error will result in an unchanged hash value is \(2^{-n}\).

With more predictably formatted data, the function is less effective. For example, in normal text files, the higher order bit of each octet is always 0. So if a 128 bit hash value is used, instead of an effectiveness of \(2^{-128}\), the hash function on this type of data has an effectiveness of \(2^{-112}\).
A simple way to improve matters is to perform one-bit circular shift, or rotation, on the hash value after each block is processed. The procedure can be summarized as

1. Initially set the $n$-bit hash value to 0
A simple way to improve matters is to perform one-bit circular shift, or rotation, on the hash value after each block is processed. The procedure can be summarized as

1. Initially set the $n$-bit hash value to 0
2. Process each successive $n$-bit block as
   1. Rotate the current hash value to the left by one bit

This has the effect of 'randomizing' the input more completely and overcoming any regularities that appear in the input.
A simple way to improve matters is to perform a one-bit circular shift, or rotation, on the hash value after each block is processed. The procedure can be summarized as

1. Initially set the $n$-bit hash value to 0
2. Process each successive $n$-bit block as
   1. Rotate the current hash value to the left by one bit
   2. XOR the block into the hash value

This has the effect of 'randomizing' the input more completely and overcoming any regularities that appear in the input.
A simple way to improve matters is to perform one-bit circular shift, or rotation, on the hash value after each block is processed. The procedure can be summarized as

1. Initially set the $n$-bit hash value to 0
2. Process each successive $n$-bit block as
   1. Rotate the current hash value to the left by one bit
   2. XOR the block into the hash value

This has the effect of ‘randomizing’ the input more completely and overcoming any regularities that appear in the input.
Note

Although the second procedure provides a good measure of data integrity, it is virtually useless for data security when an encrypted hash code is used with a plaintext message because it is easy to produce a new message when given an original message that yields the same hash code; simply prepare the alternate message and then append an $n$-bit block that forces the new message plus the block to yield the desired hash code.
Although simple XOR or rotated XOR (RXOR) is insufficient if only the hash code is encrypted, you may still feel that such a simple function could be useful when the message together with the hash code is encrypted. But you must be careful. A technique originally proposed by the National Bureau of Standards used a simple XOR applied to 64-bit blocks of the message and then encryption of the entire message that used the cipher block chaining (CBC) mode.
We can define the scheme as follows:

- Given a message $M$ consisting of a sequence of 64-bit blocks $P_1, P_2, \ldots, P_N$, define the hash code $h = H(M)$ as the block-by-block XOR of all blocks and append the hash code as the final block:

$$h = P_{N+1} = P_1 \oplus P_2 \oplus \ldots \oplus P_N$$
We can define the scheme as follows:

- Given a message $M$ consisting of a sequence of 64-bit blocks $P_1, P_2, \ldots, P_N$, define the hash code $h = H(M)$ as the block-by-block XOR of all blocks and append the hash code as the final block:

  $$h = P_{N+1} = P_1 \oplus P_2 \oplus \ldots \oplus P_N$$

- Next, encrypt the entire message plus the hash code using CBC mode to produce the encrypted message $C_1, C_2, \ldots, C_{N+1}$. There are several ways the ciphertext can be manipulated in such a way that it is not detectable by the hash code.
CBC Example

By the definition of CBC,

\[
CBC : C_j = E(K, [C_{j-1} \oplus P_j])
\]
By the definition of CBC,

\[ CBC : C_j = E(K, [C_{j-1} \oplus P_j]) \]

We have

\[ P_1 = IV \oplus D(K, C_1) \]
\[ P_i = C_{i-1} \oplus D(K, C_i) \]
\[ P_{N+1} = C_N \oplus D(K, C_{N+1}) \]
But, $P_{N+1}$ has the hash code.

\[
P_{N+1} = P_1 \oplus P_2 \oplus \ldots \oplus P_N \\
= [IV \oplus D(K, C_1)] \oplus [C_1 \oplus D(K, C_2)] \oplus \ldots \oplus [Y_{C+1} \oplus D(K, C_N)]
\]
But, $P_{N+1}$ has the hash code.

$$P_{N+1} = P_1 \oplus P_2 \oplus \ldots \oplus P_N$$

$$= [IV \oplus D(K, C_1)] \oplus [C_1 \oplus D(K, C_2)] \oplus \ldots \oplus [Y_{C+1} \oplus D(K, C_N)]$$

Because the terms in the preceding equation can be XOR’ed in any order, it follows that the hash code would not change if the ciphertext blocks were permuted.
For a hash value $h = H(x)$, we say that $x$ is the preimage of $h$. That is, $x$ is the data block whose hash function, using the function $H$, is $h$. Because $H$ is a many-to-one mapping, for any given hash value, there will in general be many preimages. A collision occurs if we have $x \neq y$ and $H(x) = H(y)$. Because we are using hash functions for data integrity, collisions are clearly undesirable.
For a hash value $h = H(x)$, we say that $x$ is the preimage of $h$. That is, $x$ is the data block whose hash function, using the function $H$, is $h$.

Because $H$ is a many-to-one mapping, for any given hash value, there will in general be many preimages. A collision occurs if we have $x \neq y$ and $H(x) = H(y)$. Because we are using hash functions for data integrity, collisions are clearly undesirable.
How many preimages are there for a given hash value? (This is a measure of the potential collisions for a given hash value.)
How many preimages are there for a given hash value? (This is a measure of the potential collisions for a given hash value.)

Suppose the length of the hash block is $n$ bits, and the function $H$ takes as input messages or data blocks with length $b$ bits with $b > n$. Then, the total number of possible messages is $2^b$ and the total number of possible hash values is $2^n$. 
How many preimages are there for a given hash value? (This is a measure of the potential collisions for a given hash value.)

Suppose the length of the hash block is $n$ bits, and the function $H$ takes as input messages or data blocks with length $b$ bits with $b > n$. Then, the total number of possible messages is $2^b$ and the total number of possible hash values is $2^n$.

If $H$ tends to uniformly distribute hash values then each hash value will have close to $2^\frac{b}{n}$ images.
How many preimages are there for a given hash value? (This is a measure of the potential collisions for a given hash value.)

Suppose the length of the hash block is $n$ bits, and the function $H$ takes as input messages or data blocks with length $b$ bits with $b > n$. Then, the total number of possible messages is $2^b$ and the total number of possible hash values is $2^n$.

If $H$ tends to uniformly distribute hash values then each hash value will have close to $2^{b/n}$ images.

If we now allow inputs of arbitrary length, not just fixed length of some number of bits, then the number of preimages per hash value is arbitrarily large. However the security risks in the use of hash functions are not as severe as they might appear.
To understand this better, the security implications of cryptographic hash functions, we need to precisely define their security requirements.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Variable input size</td>
<td>$H$ can be applied to a block of any size</td>
</tr>
</tbody>
</table>
To understand this better, the security implications of cryptographic hash functions, we need to precisely define their security requirements.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Variable input size</td>
<td>$H$ can be applied to a block of any size</td>
</tr>
<tr>
<td>2. Fixed output size</td>
<td>It produces a fixed length output</td>
</tr>
</tbody>
</table>
To understand this better, the security implications of cryptographic hash functions, we need to precisely define their security requirements.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Variable input size</td>
<td>$H$ can be applied to a block of any size</td>
</tr>
<tr>
<td>2. Fixed output size</td>
<td>It produces a fixed length output</td>
</tr>
<tr>
<td>3. Efficiency</td>
<td>$H(x)$ is relatively easy to compute for given $x$, making both hardware and software implementation practical</td>
</tr>
</tbody>
</table>
To understand this better, the security implications of cryptographic hash functions, we need to precisely define their security requirements.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Variable input size</td>
<td>$H$ can be applied to a block of any size</td>
</tr>
<tr>
<td>2. Fixed output size</td>
<td>It produces a fixed length output</td>
</tr>
<tr>
<td>3. Efficiency</td>
<td>$H(x)$ is relatively easy to compute for given $x$, making both hardware and software implementation practical</td>
</tr>
</tbody>
</table>

These first three properties are requirements for the practical application of a hash function.
<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Preimage resistant</td>
<td>For any given hash value, it is computationally infeasible to find y ∈ H(y) = h</td>
</tr>
<tr>
<td>(one-way property)</td>
<td></td>
</tr>
</tbody>
</table>

| 5. Second preimage resistant       | For any given block x, it is weak collision resistant computationally infeasible to find y ≠ x with H(y) = H(x) |
| 6. Collision resistant            | It is computationally infeasible strong collision resistant to find any pair (x, y) such that H(x) = H(y) |
| 7. Pseudorandomness               | Output of H meets standard tests for pseudorandomness.                       |
## Security Requirements

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Preimage resistant</td>
<td>For any given hash value, it is computationally infeasible to find $y \ni H(y) = h$</td>
</tr>
<tr>
<td>(one-way property)</td>
<td></td>
</tr>
<tr>
<td>5. Second preimage resistant</td>
<td>For any given block $x$, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$</td>
</tr>
<tr>
<td>(weak collision resistant)</td>
<td></td>
</tr>
</tbody>
</table>
## Security Requirements

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Preimage resistant</td>
<td>For any given hash value, it is computationally infeasible to find $y \ni \text{H}(y) = h$</td>
</tr>
<tr>
<td>(one-way property)</td>
<td></td>
</tr>
<tr>
<td>5. Second preimage resistant</td>
<td>For any given block $x$, it is computationally infeasible to find $y \neq x$ with $\text{H}(y) = \text{H}(x)$</td>
</tr>
<tr>
<td>(weak collision resistant)</td>
<td></td>
</tr>
<tr>
<td>6. Collision resistant</td>
<td>It is computationally infeasible to find any pair $(x, y)$ such that $\text{H}(x) = \text{H}(y)$</td>
</tr>
<tr>
<td>(strong collision resistant)</td>
<td></td>
</tr>
</tbody>
</table>
## Security Requirements

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Preimage resistant (one-way property)</td>
<td>For any given hash value, it is computationally infeasible to find $y \ni H(y) = h$</td>
</tr>
<tr>
<td>5. Second preimage resistant (weak collision resistant)</td>
<td>For any given block $x$, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$</td>
</tr>
<tr>
<td>6. Collision resistant (strong collision resistant)</td>
<td>It is computationally infeasible to find any pair $(x, y)$ such that $H(x) = H(y)$</td>
</tr>
<tr>
<td>7. Pseudorandomness</td>
<td>Output of $H$ meets standard tests for pseudorandomness</td>
</tr>
</tbody>
</table>
The fourth property, preimage resistant, is the one-way property. It is easy to generate a code given a message, but virtually impossible to generate a message given a code.
The fourth property, **preimage resistant**, is the one-way property. It is easy to generate a code given a message, but virtually impossible to generate a message given a code.

This property is important if the authentication technique involves the use of a secret value (our third scenario). The secret value is not sent.
The fourth property, **preimage resistant**, is the one-way property. It is easy to generate a code given a message, but virtually impossible to generate a message given a code.

This property is important if the authentication technique involves the use of a secret value (our third scenario). The secret value is not sent. However, if the hash function is not one-way, an attacker can easily discover the secret value: if the attacker can interrupt or observe a transmission, the attacker obtains the message $M$ and the hash code $h = H(M||S)$. The attacker then inverts the hash function to obtain

$$M||S = H^{-1}(MD_M)$$

Because the attacker now has both $M$ and $M||S$, it is a trivial matter to recover $S$. 

The fifth property, **second preimage resistant**, guarantees that it is impossible to find an alternative message with the same hash value as the given message. This prevents forgery when an encrypted hash code is issued (our no. 2 and 3). If this property were not true, an attacker would be capable of the following sequence:

1. Observe or intercept a message plus its encrypted hash code
2. Generate an unencrypted hash code from the message
3. Generate an alternate message with the same hash code
The fifth property, second preimage resistant, guarantees that it is impossible to find an alternative message with the same hash value as the given message.

This prevents forgery when an encrypted hash code is issued (our no. 2 and 3).
The fifth property, second preimage resistant, guarantees that it is impossible to find an alternative message with the same hash value as the given message.

This prevents forgery when an encrypted hash code is issued (our no. 2 and 3).

If this property were not true, an attacker would be capable of the following sequence:

1. Observe or intercept a message plus its encrypted hash code
The fifth property, second preimage resistant, guarantees that it is impossible to find an alternative message with the same hash value as the given message.

This prevents forgery when an encrypted hash code is issued (our no. 2 and 3).

If this property were not true, an attacker would be capable of the following sequence:

1. Observe or intercept a message plus its encrypted hash code
2. Generate an unencrypted hash code from the message
The fifth property, \textit{second preimage resistant}, guarantees that it is impossible to find an alternative message with the same hash value as the given message.

This prevents forgery when an encrypted hash code is issued (our no. 2 and 3).

If this property were not true, an attacker would be capable of the following sequence:

1. Observe or intercept a message plus its encrypted hash code
2. Generate an unencrypted hash code from the message
3. Generate an alternate message with the same hash code
A hash function that satisfies these first five properties is referred to as a weak hash function.
A hash function that satisfies these first five properties is referred to as a weak hash function.

If the 6th property, collision resistant is also satisfied, then it is referred to as a strong hash function.
A hash function that satisfies these first five properties is referred to as a weak hash function.

If the 6th property, collision resistant is also satisfied, then it is referred to as a strong hash function.

A strong hash function protects against an attack in which one party generates a message for another party to sign. For example, suppose Bob writes an IOU message, sends it to Alice and she signs it. Bob finds two messages with the same hash, one which requires Alice to pay a small amount and one a large amount. Alice signs the first message and then Bob is able to claim the second is authentic.
This figure gives the relationships among the three resistance properties. A function that is collision resistant is also second preimage resistant, but the reverse is not necessarily true. It is possible to be preimage resistant and not second preimage resistant, and vice versa.
This figure gives the relationships among the three resistance properties. A function that is collision resistant is also second preimage resistant, but the reverse is not necessarily true.
This figure gives the relationships among the three resistance properties. A function that is collision resistant is also second preimage resistant, but the reverse is not necessarily true.

It is possible to be preimage resistant and not second preimage resistant, and vice versa.
The final requirement, pseudorandomness, has not traditionally been listed as a requirement for cryptographic hash functions but is more or less implied.
The final requirement, **pseudorandomness**, has not traditionally been listed as a requirement for cryptographic hash functions but is more or less implied.

Cryptographic hash functions are commonly used for key derivation and pseudorandom number generation, and that in message integrity applications, the three resistance properties depend on the output of the hash functions appearing to be random. Thus, it makes sense to verify that in fact a given hash function produces pseudorandom output.
Brute Force Attacks v. Cryptanalysis

As with encryption algorithms, there are two categories of all attacks on hash functions: brute force and cryptanalysis.
As with encryption algorithms, there are two categories of all attacks on hash functions: brute force and cryptanalysis.

A brute-force attack does not depend on the specific algorithm but depends only on bit length. In the case of a hash function, a brute force attack depends only on the bit length of the hash value.
As with encryption algorithms, there are two categories of all attacks on hash functions: brute force and cryptanalysis.

A brute-force attack does not depend on the specific algorithm but depends only on bit length. In the case of a hash function, a brute force attack depends only on the bit length of the hash value.

Cryptanalysis, in contrast, is an attack based on weaknesses in a particular cryptographic algorithm.
For these attacks, an adversary wishes to find a $y$ such that $H(y)$ is equal to a given hash value $h$. 
For these attacks, an adversary wishes to find a $y$ such that $H(y)$ is equal to a given hash value $h$.

The brute force method is to pick values of $y$ at random and try each value until a collision occurs. For an $m$-bit hash value, the levels of effort is proportional to $2^m$. Specifically, the adversary would have to try, on average, $2^{m-1}$ values of $y$ to find one that generates a given hash value.
For a collision resistant attack, an adversary wishes to find two messages or data blocks, $x$ and $y$, that yield the same hash function, $H(x) = H(y)$. 
Collision Resistant Attacks

For a collision resistant attack, an adversary wishes to find two messages or data blocks, $x$ and $y$, that yield the same hash function, $H(x) = H(y)$.

This turns out to require significantly less effort than a preimage or a second preimage attack. The effort required is explained by the birthday paradox.
Collision Resistant Attacks

For a collision resistant attack, an adversary wishes to find two messages or data blocks, \( x \) and \( y \), that yield the same hash function, \( H(x) = H(y) \).

This turns out to require significantly less effort than a preimage or a second preimage attack. The effort required is explained by the birthday paradox.

If we choose random variables from a uniform distribution in the range \([0, N - 1]\), then the probability that a repeated element is encountered exceeds .5 after \( \sqrt{N} \) choices have been made.
Collision Resistant Attacks

For a collision resistant attack, an adversary wishes to find two messages or data blocks, $x$ and $y$, that yield the same hash function, $H(x) = H(y)$.

This turns out to require significantly less effort than a preimage or a second preimage attack. The effort required is explained by the birthday paradox.

If we choose random variables from a uniform distribution in the range $[0, N - 1]$, then the probability that a repeated element is encountered exceeds .5 after $\sqrt{N}$ choices have been made.

Thus, for an $m$-bit hash value, if we pick data blocks at random, we can expect to find two data blocks with the same hash value within $\sqrt{2^m} = 2^{m/2}$ attempts.
The source, A, is prepared to sign a legitimate message $x$ by appending the appropriate $m$-bit hash code and encryption with A’s private key.
The source, A, is prepared to sign a legitimate message \( x \) by appending the appropriate \( m \)-bit hash code and encryption with A’s private key.

The opponent generates \( 2^{\frac{m}{2}} \) variations \( x' \) of \( x \), all of which convey essentially the same meaning and stores the messages and their hash values.
Exploiting the Birthday Paradox in a Collision Resistant Attack

1. The source, A, is prepared to sign a legitimate message $x$ by appending the appropriate $m$-bit hash code and encryption with A’s private key.

2. The opponent generates $2^{m/2}$ variations $x'$ of $x$, all of which convey essentially the same meaning and stores the messages and their hash values.

3. The opponent prepares a fraudulent message $y$ for which A’s signature is desired.
The opponent generates minor variations $y'$ of $y$, all of which convey essentially the same meaning. For each $y'$, the opponent computes $H(y')$, checks for matches with any of the $H(x')$ values and continues until a match is found.
The opponent generates minor variations $y'$ of $y$, all of which convey essentially the same meaning. For each $y'$, the opponent computes $H(y')$, checks for matches with any of the $H(x')$ values and continues until a match is found.

The opponent offers the valid variation to A for signature. The signature can then be attached to the fraudulent variation for transmission to the intended recipient. Because the two variations have the same hash code, they will produce the same signature; the opponent is assured of success even though the encryption key is not known.
The generations of many variations that convey the same meaning is not difficult.

- The opponent could insert a number of ‘space-space-backspace’ characters between pairs of words throughout the document.
Generating the Variations

The generations of many variations that convey the same meaning is not difficult.

- The opponent could insert a number of ‘space-space-backspace’ characters between pairs of words throughout the document.
- Variations could then be generated by substituting ‘space-backspace-space’ in selected instances.
The generations of many variations that convey the same meaning is not difficult.

- The opponent could insert a number of ‘space-space-backspace’ characters between pairs of words throughout the document.
- Variations could then be generated by substituting ‘space-backspace-space’ in selected instances.
- The opponent could simply reword the message but retain the meaning.
As with encryption algorithms, cryptanalysis attacks on hash functions seek to exploit some property of the algorithm to perform some attack other than an exhaustive search. The way to measure the resistance of a hash algorithm to cryptanalysis is to compare its strength to the effort required for a brute force attack. That is, an ideal hash algorithm will require a cryptanalysis effort greater than a brute force effort.
As with encryption algorithms, cryptanalysis attacks on hash functions seek to exploit some property of the algorithm to perform some attack other than an exhaustive search. The way to measure the resistance of a hash algorithm to cryptanalysis is to compare its strength to the effort required for a brute force attack. That is, an ideal hash algorithm will require a cryptanalysis effort greater than a brute force effort.

In recent years, there has been considerable effort and some successes in developing cryptanalytic attacks on hash functions. To understand these we need to look at the overall structure of a typical hash function. The structure is referred to as an iterated hash function and is the structure of most hash functions today.
Structure of an Iterative Hash Function

$IV = \text{initial value}$

$CV_i = \text{chaining variable}$

$Y_i = i^{th} \text{ input attack}$

$f = \text{compression algorithm}$

$L = \text{number of input boxes}$

$n = \text{length of hash code}$

$b = \text{length of input block}$
How This Function Works

- The hash function takes an input message and partitions it into $L$ fixed-sized blocks of $b$ bits each.
How This Function Works

- The hash function takes an input message and partitions it into $L$ fixed-sized blocks of $b$ bits each.
- If necessary, the final block is padded to $b$ bits.
How This Function Works

- The hash function takes an input message and partitions it into $L$ fixed-sized blocks of $b$ bits each.
- If necessary, the final block is padded to $b$ bits.
- The final block also includes the value of the total length of the input to the hash function.
The hash function takes an input message and partitions it into $L$ fixed-sized blocks of $b$ bits each.

If necessary, the final block is padded to $b$ bits.

The final block also includes the value of the total length of the input to the hash function.

The inclusion of the length makes the job of the opponent more difficult. Either the opponent must find two messages of equal length that hash to the same value or two messages of differing lengths that hash to the same value.
How This Function Works

- The hash algorithm involves repeated use of a compression function, $f$, that takes two inputs ($n$ bits from the previous step called the chaining variable and a $b$-bit block) and produces an $n$-bit output.
How This Function Works

- The hash algorithm involves repeated use of a compression function, $f$, that takes two inputs ($n$ bits from the previous step called the chaining variable and a $b$-bit block) and produces an $n$-bit output.
- At the start of hashing, the chaining variable has an initial value that is specified as part of the algorithm.
- The final value of the chaining variable is the hash value.
How This Function Works

- The hash algorithm involves repeated use of a compression function, \( f \), that takes two inputs (\( n \) bits from the previous step called the chaining variable and a \( b \)-bit block) and produces an \( n \)-bit output.
- At the start of hashing, the chaining variable has an initial value that is specified as part of the algorithm.
- The final value of the chaining variable is the hash value.

The hash function can be summarized as

\[
CV_0 = IV, \text{ the initial } n\text{-bit value}
\]
The hash algorithm involves repeated use of a compression function, $f$, that takes two inputs ($n$ bits from the previous step called the chaining variable and a $b$-bit block) and produces an $n$-bit output.

At the start of hashing, the chaining variable has an initial value that is specified as part of the algorithm.

The final value of the chaining variable is the hash value.

The hash function can be summarized as

\[
CV_0 = IV, \text{ the initial } n\text{-bit value} \\
CV_i = f(CV_{i-1}, Y_{i-1}), \ 1 \leq i \leq L
\]
How This Function Works

- The hash algorithm involves repeated use of a compression function, $f$, that takes two inputs ($n$ bits from the previous step called the chaining variable and a $b$-bit block) and produces an $n$-bit output.
- At the start of hashing, the chaining variable has an initial value that is specified as part of the algorithm.
- The final value of the chaining variable is the hash value.

The hash function can be summarized as

$$CV_0 = IV, \text{ the initial } n\text{-bit value}$$

$$CV_i = f(CV_{i-1}, Y_{i-1}), 1 \leq i \leq L$$

$$H(M) = CV_L$$
How This Function Works

- The hash algorithm involves repeated use of a compression function, $f$, that takes two inputs ($n$ bits from the previous step called the chaining variable and a $b$-bit block) and produces an $n$-bit output.
- At the start of hashing, the chaining variable has an initial value that is specified as part of the algorithm.
- The final value of the chaining variable is the hash value.

The hash function can be summarized as

\[
CV_0 = IV, \text{ the initial } n\text{-bit value} \\
CV_i = f(CV_{i-1}, Y_{i-1}), \ 1 \leq i \leq L \\
H(M) = CV_L
\]

where the input to the hash function is a message $M$ consisting of blocks $Y_0, Y_1, \ldots, Y_{L-1}$. 
Cryptanalysis of hash functions focuses on the internal structure of $f$ and is based on attempts to find efficient techniques for producing collisions for a single execution of $f$. Once that is done, the attack must take into account the fixed value of $IV$. The attack on $f$ depends on exploiting the internal structure.

Cryptanalysis of hash functions focuses on the internal structure of $f$ and is based on attempts to find efficient techniques for producing collisions for a single execution of $f$. Once that is done, the attack must take into account the fixed value of $IV$. The attack on $f$ depends on exploiting the internal structure.

Typically, as with symmetric block ciphers, $f$ consists of a series of rounds of processing so that the attack involves analysis of the pattern of bit changes from round to round.
Cryptanalysis of hash functions focuses on the internal structure of $f$ and is based on attempts to find efficient techniques for producing collisions for a single execution of $f$. Once that is done, the attack must take into account the fixed value of $IV$. The attack on $f$ depends on exploiting the internal structure.

Typically, as with symmetric block ciphers, $f$ consists of a series of rounds of processing so that the attack involves analysis of the pattern of bit changes from round to round.

Keep in mind that for any hash function, there must exist collisions, because we are mapping a message of length at least equal to twice the block size $b$ (because we must append a length field) into a hash code of length $n$, where $b \geq n$. What is required is that it is computationally infeasible to find collisions.
A number of proposals have been made for hash functions based in using a cipher block chaining technique, but without using a secret key.

One of the first proposals was by Rabin in a 1978 article. Divide a message $M$ into fixed sized blocks $M_1, M_2, \ldots, M_N$ and use a symmetric encryption system such as DES to compute the hash code $G$ as

$$
H_0 = \text{initial value} \\
H_i = E(M_i, H_{i-1}) \\
G = H_N
$$

This is similar to the CBC technique, but in this case, there is no secret key. As with any hash code, this scheme is subject to a birthday attack, and if the encryption algorithm is DES and only a 64-bit hash code is produced, then the system is vulnerable.
Hash Functions Based on Cipher Block Chaining

A number of proposals have been made for hash functions based in using a cipher block chaining technique, but without using a secret key.

One of the first proposals was by Rabin in a 1978 article. Divide a message $M$ into fixed sized blocks $M_1, M_2, \ldots M_N$ and use a symmetric encryption system such as DES to compute the hash code $G$ as

\[
\begin{align*}
H_0 &= \text{initial value} \\
H_i &= E(M_i, H_{i-1}) \\
G &= H_N
\end{align*}
\]
Hash Functions Based on Cipher Block Chaining

A number of proposals have been made for hash functions based in using a cipher block chaining technique, but without using a secret key.

One of the first proposals was by Rabin in a 1978 article. Divide a message $M$ into fixed sized blocks $M_1, M_2, \ldots, M_N$ and use a symmetric encryption system such as DES to compute the hash code $G$ as

$$H_0 = \text{initial value}$$
$$H_i = E(M_i, H_{i-1})$$
$$G = H_N$$

This is similar to the CBC technique, but in this case, there is no secret key. As with any hash code, this scheme is subject to a birthday attack, and if the encryption algorithm is DES and only a 64-bit hash code is produced, then the system is vulnerable.
Another version of the birthday attack can be used even if the opponent has access to only one messages its and its valid signature and cannot obtain multiple settings.
Another Birthday Attack

Another version of the birthday attack can be used even if the opponent has access to only one messages its and its valid signature and cannot obtain multiple settings.

This form of attack is known as a meet-in-the-middle attack.
Another Birthday Attack

Another version of the birthday attack can be used even if the opponent has access to only one messages its and its valid signature and cannot obtain multiple settings.

This form of attack is known as a meet-in-the-middle attack.

Here is the scenario: we assume the opponent intercepts a message with a signature in the form of an encrypted hash code and that the unencrypted hash code is $m$ bits long.
How the Attack Works

1. Use the previous algorithm to calculate the unencrypted hash code $G$. 
How the Attack Works

1. Use the previous algorithm to calculate the unencrypted hash code $G$.
2. Construct any desired message in the form $Q_1, Q_2, \ldots, Q_{n-2}$. 

3. Compute $H_i = E(Q_i, H_{i-1})$ for $1 \leq i \leq (N-2)$.

4. Generate $2m^2$ random blocks for each block $X$, compute $E(X, H_{N-2})$. Generate an additional $2m^2$ random blocks; for each block $Y$, compute $D(Y, G)$, where $D$ is the decryption function corresponding to $E$.

5. Based on the birthday paradox, with high probability there will be an $X$ and $Y$ such that $E(X, H_{N-2}) = D(Y, G)$.

6. Form the message $Q_1, Q_2, \ldots, Q_{N-2}, X, Y$. The message has the hash code $G$ and can therefore be used with the intercepted encrypted signature.
How the Attack Works

1. Use the previous algorithm to calculate the unencrypted hash code $G$.
2. Construct any desired message in the form $Q_1, Q_2, \ldots, Q_{n-2}$.
3. Compute $H_i = E(Q_i, H_{i-1})$ for $1 \leq i \leq (N - 2)$.
How the Attack Works

1. Use the previous algorithm to calculate the unencrypted hash code $G$.
2. Construct any desired message in the form $Q_1, Q_2, \ldots, Q_{n-2}$.
3. Compute $H_i = E(Q_i, H_{i-1})$ for $1 \leq i \leq (N - 2)$.
4. Generate $2^m$ random blocks for each block $X$, compute $E(X, H_{N-2})$. Generate an additional $2^m$ random blocks; for each block $Y$, compute $D(Y, G)$, where $D$ is the decryption function corresponding to $E$.
How the Attack Works

1. Use the previous algorithm to calculate the unencrypted hash code $G$.

2. Construct any desired message in the form $Q_1, Q_2, \ldots, Q_{n-2}$.

3. Compute $H_i = E(Q_i, H_{i-1})$ for $1 \leq i \leq (N - 2)$.

4. Generate $2^{m/2}$ random blocks for each block $X$, compute $E(X, H_{N-2})$. Generate an additional $2^{m/2}$ random blocks; for each block $Y$, compute $D(Y, G)$, where $D$ is the decryption function corresponding to $E$.

5. Based on the birthday paradox, with high probability there will be an $X$ and $Y$ such that $E(X, H_{N-2}) = D(Y, G)$. 


How the Attack Works

1. Use the previous algorithm to calculate the unencrypted hash code $G$.

2. Construct any desired message in the form $Q_1, Q_2, \ldots, Q_{n-2}$.

3. Compute $H_i = E(Q_i, H_{i-1})$ for $1 \leq i \leq (N - 2)$.

4. Generate $2^{m^2}$ random blocks for each block $X$, compute $E(X, H_{N-2})$. Generate an additional $2^{m^2}$ random blocks; for each block $Y$, compute $D(Y, G)$, where $D$ is the decryption function corresponding to $E$.

5. Based on the birthday paradox, with high probability there will be an $X$ and $Y$ such that $E(X, H_{N-2}) = D(Y, G)$.

6. Form the message $Q_1, Q_2, \ldots, Q_{N-2}, X, Y$. The message has the hash code $G$ and therefore can be used with the intercepted encrypted signature.
A number of researchers have proposed refinements intended to strengthen the basic block chaining approach.
A number of researchers have proposed refinements intended to strengthen the basic block chaining approach.

For example, Davies and Price proposed in 1989 a variation

$$H_i = E(M_i, H_{i-1}) \oplus H_{i-1}$$
A number of researchers have proposed refinements intended to strengthen the basic block chaining approach.

For example, Davies and Price proposed in 1989 a variation

\[ H_i = E(M_i, H_{i-1}) \oplus H_{i-1} \]

Another variation proposed in 1988 by Meyer and Schilling is

\[ H_i = E(H_{i-1}, M_i) \oplus M_i \]
A number of researchers have proposed refinements intended to strengthen the basic block chaining approach.

For example, Davies and Price proposed in 1989 a variation

\[ H_i = E(M_i, H_{i-1}) \oplus H_{i-1} \]

Another variation proposed in 1988 by Meyer and Schilling is

\[ H_i = E(H_{i-1}, M_i) \oplus M_i \]

However, both of these schemes have been shown to be vulnerable to a variety of attacks. More generally, it can be shown that some form of the birthday attack will succeed against any hash scheme involving the use of CBC without a secret key, provided that either the resulting hash code is small enough (64 bits or less) or that a larger hash code can be decomposed into independent subcodes.
In recent years, the most widely used hash function has been the SHA because virtually every other widely used hash function has been found to have substantial cryptanalytic weaknesses. SHA was the last standing standardized hash algorithm by 2005.
In recent years, the most widely used hash function has been the SHA because virtually every other widely used hash function has been found to have substantial cryptanalytic weaknesses. SHA was the last standing standardized hash algorithm by 2005.

Possible project topic?