# Secure Hash Algorithms

Another basic tool for cryptography is a secure hash algorithm. Unlike encryption, given a variable-length message $x$, a secure hash algorithm computes a function $h(x)$ which has a fixed and often smaller number of bits. So it is usually not possible to recover $x$ from its hash value. The hash function is secure if it is hard to get information about the $x$'s that hash to a particular value. The properties we would like from a secure hash function are:

1. A hash function $h(x)$ is said to be *one-way* if given $y$ it is hard to find an $x$ such that $h(x) = y$.
2. A hash function $h(x)$ is said to be *weakly collision-free* if given a message $x_1$ it is hard to find another message $x_2$ such that $h(x_1) = h(x_2)$.
3. A hash function $h(x)$ is said to be *strongly collision-free* if it is hard to find any pair of messages $x_1$, $x_2$ such that $h(x_1) = h(x_2)$.

and as the terminology suggests, a function which is strongly collision-free is also weakly collision-free.

One of the most popular secure hash algorithms is called MD5. It was invented by Ron Rivest, the R in RSA. The details of MD5 aren't very enlightening, and we wont go over it here. It is completely described in the Wayner book. At high level, it processes 512-bit blocks of data and produces 128-bit hash values. Thus it reduces the length of a large message by a factor of 4. To arrive at a hash value of fixed size, you iterate the process until the output is a single 128-bit value. As far as is known, MD5 is one-way and strongly collision-free.

Hash functions are very much like the fingerprint functions we used earlier. But one difference is that the simple modular functions we used for string matching are very easy to fool - given a string and its fingerprint, it is easy to generate other strings with the same fingerprint. That must not be the case for a secure hash function.

An important property of secure hash functions, like any hash function, is that they should uniformly cover their range. That is, if you place a uniform distribution on the inputs, the output probabilities from the hash function should be uniform. But we typically hope for much more. Namely that for any "reasonable" probability distribution on the inputs, the output probabilities should still be uniform. e.g. if the inputs consists of ASCII representation of normal english text (which is very non-uniform compared to random binary strings) the output distribution from the hash function should still be uniform. We will henceforth assume that is true.

# Digital Signatures

The purpose of a digital signature is similar to a physical signature. That is, you endorse some document in a way that: (i) others can verify that you signed that particular document (ii) it is difficult for someone else to forge your signature. The simplest signature schemes allow anyone to verify both the document you signed, and who you are.

Here is a simple signature scheme. It assumes that you, the signer, have a published RSA key $e$, and a corresponding private key $d$. Given a document $x$, first compute a hash of it using MD5, $y = h(x)$. Your signature will be the RSA encryption of $y$ by your *secret key*, which we write as $R(y, d)$ (we wont distinguish between encryption and decryption functions for RSA since they are the same, i.e. exponentiation by the key mod n). That is, your signature is

$$s \;=\; R(h(x), d)$$

Now anyone else that has access to the document $x$ and your signature $s$ can verify that you have signed it by computing $R(s, e)$ using your public key $e$. Since $e$ and $d$ are inverses,

$$R(R(h(x), d), e) \;=\; h(x)$$

Now the verifier (the person checking your signature) compares this value with the hash value that they compute directly from the document using MD5, which will also be $h(x)$.

Because MD5 is a secure hash function, it is impractical to construct another document with the same hash value. So the verifier knows that only this document could have been signed. Furthermore, because it is difficult for anyone else to discover your private key, the fact that you were able to compute $R(h(x), d)$ convinces the verifier that you know what $d$ is. That is, you are the person you claim to be, and that you intended to sign this document.

### Limitations of RSA

The basic RSA scheme has some limitations. The most obvious is vulnerability of small messages. Suppose a user sends a small ($k$-bit) message M using a public key $e$. A cracker may see the encrypted message. Although they cant invert it directly, they could instead enumerate all $k$-bit messages and encrypt them until they find one matching the intercepted message.

There are ways of dealing with this problems, but they are addressed "naturally" in a different crypto-system called Diffie-Hellman/El-Gamal (El-Gamal evolved from Diffie-Hellman which was originally a key exchange protocol).

# Diffie-Hellman/El-Gamal encryption

The El-Gamal crypto-system relies on the difficulty of the discrete logarithm problem. If $p$ is prime, then $\mathbb{Z}_p^*$ is cyclic. Let $g$ be a generator of $\mathbb{Z}_p^*$. The discrete logarithm of an integer $m$ is

the minimum exponent $e$ such that $m = g^e (\mathrm{mod}\ \ p)$. Computing discrete logs is believed to be intractable, although just like factoring this has never been proved. The El-Gamal scheme has the following components:

**Key Generation**
Alice picks a strong prime $p$, and knows the factorization of $p-1$. She also chooses a generator $g$ of $\mathbb{Z}_p^*$ (she can test if random elements are generators using the factorization of $p-1$). She choose a secret key $s$ and computes $h = g^s$. She publishes $(p, g, h)$ as her public key.

**Encryption**
Bob wants to send a message $M$ to Alice. He chooses a random "one-time" key $r$, and sends $(A, B) = (g^r, Mh^r)$ to Alice.

**Decryption**
Alice receives Bob's message $(A, B)$. She computes $BA^{-s}$ which is $Mh^r(g^r)^{-s} = Mh^r(g^{-s})^r = Mh^r h^{-r} = M$.

Clearly, El-Gamal can't work unless discrete log is hard. The public key contains both $g$ and $h = g^s$, and if an observer could compute the discrete log of $h$ wrt $g$, they would get the secret key immediately. The difficulty of El-Gamal is as problematic as RSA. It is not provably equivalent to the discrete log problem (which is not provably hard anyway). But all of these problems have closely related complexity. Improvements in factoring have generally led to improvements in discrete log and vice-versa. The acceptable key sizes for RSA and El-Gamal have tracked closely over the years. The current acceptable key size is 1024 bits, although 2048-bit keys are being considered and may become standard in a few years.

One advantage of El-Gamal over basic RSA is that it includes a "security parameter" which protects small messages. An encrypted message is defined by both the input message $M$ and the sender's one-time key $r$. Seeing different encryptions of the same message doesnt help an observer figure out what the message is.

**Key exchange**

One interesting property of this scheme is that two agents that know each other's public keys automatically know a shared secret. Suppose Alice has private key $x$ and publishes $(p, g, g^x)$ as her public key. Bob has a private key $y$ and publishes $(p, g, g^y)$ as his public key. Bob can look up Alice's public key and compute $S = (g^x)^y$ from it using his private key. Similarly, Alice can look up Bob's public key and compute $S = (g^y)^x$ from it using her private key. Then Alice and Bob both know $S$, but no-one else does. They could use $S$ (for instance) to send private-key encrypted messages which are much faster than public key encryption.

This scheme was originally proposed for key-exchange by Diffie and Hellman. In this setting, the private keys $x$ and $y$ are short-term "session keys" for communication between Alice and Bob.

Such communication is susceptible to a “man in the middle” (MIM) attack by a third party who intercepts and changes the communications between Alice and Bob. If there is instead a trusted authority who stores public keys for Alice and Bob, they can use this information to generate the shared key without ever communicating with each other. They will be protected from MIM attacks, so long as they know in advance how to recognize the authority.