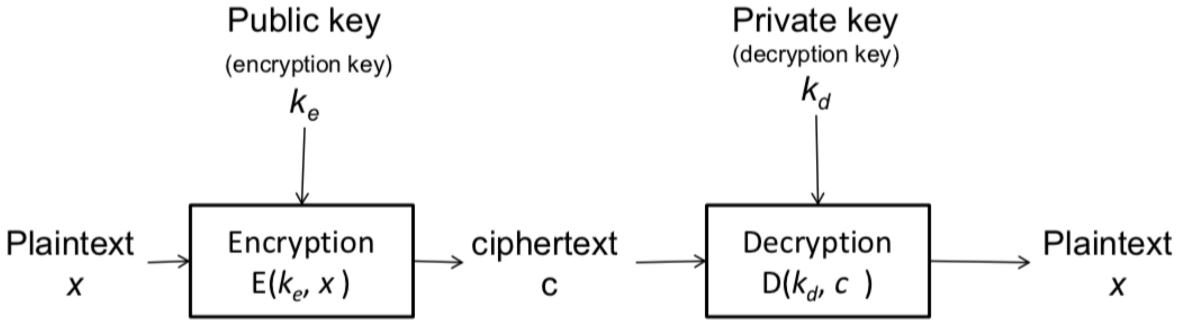
Authentication (Data Origin)

Public Key Cryptography

Symmetric Key vs Public Key

A symmetric key encryption scheme uses the same key for both encryption and decryption, while a public key or asymmetric key scheme uses two different keys for encryption and decryption.



In truth, there is no need to clearly indicate which key is the encryption key and which is the decryption key, as both keys are able to perform either decryption or encryption, just not both. As such, the following two statements are both valid:

$$P = D\big(k_{PRIV}, E(k_{PUB}, P)\big)$$
$$P = D\big(k_{PUB}, E(k_{PRIV}, P)\big)$$

In fact, decryption can even be done before encryption. All these will work, as it is effectively impossible to deduce one key from the other. It is also important for the plaintext to be difficult to determine without the private key, even with the public key and ciphertext.

As such, typically, the public key is the one shared with everyone, while the private key is kept secret by the user. It is thus possible for a person sending a message to me to encrypt it with my public key and send it to me for me to decrypt with my private key. The reverse also works – I can “decrypt” it with my private key for people to “encrypt” with my public key. However, generally, if I were to send something to somebody, I would encrypt it with the other party’s public key.

Why use Public-Key Scheme (PKS)?

Without PKS, every individual would need to keep a symmetric secret key pair with everyone else. If the number of entities is large, the number of keys to keep track would be very huge as well. Any new parties joining in would result in a lot of secret key establishing as well, which is high difficult.

The total number of keys needed without using PKS and using symmetric-key setting is n(n-1)/2 = O(n^2).

With public-key setting, every entity just needs its own public and private key. The total number of keys is 2n = O(n).

Rivest-Shamir-Adleman Algorithm

The RSA Algorithm was proposed in 1977, after Diffie and Hellman published the concept of the public-private key cryptosystem in 1976.

Textbook RSA

Keys for the RSA Algorithm are generated by:

1. Choose two distinct prime numbers p and q
   a. They should be chosen at random and should be similar in terms of magnitude, but differ in length by a few digits
   b. Can be found efficiently via the primality test
   c. They are to be kept secret
2. Compute $n = pq$
   a. n is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length
   b. n is released as part of the public key
3. Compute $\lambda(n)$. After some calculations, we get $\lambda(n) = \text{lcm}(p - 1, q - 1)$.
   a. $\lambda(n)$ is kept secret
4. Choose an integer e such that $1 < e < \lambda(n)$ and $\gcd(e, \lambda(n)) = 1$; that is, e and $\lambda(n)$ are coprime.
   a. e is released as part of the public key
5. Determine d as $d \equiv e^{-1} \pmod{\lambda(n)}$; that is, d is the modular multiplicative inverse of e modulo $\lambda(n)$.
   a. This means: solve for d the equation $d \cdot e \equiv 1 \pmod{\lambda(n)}$.
   b. d is kept secret as the private key exponent.

For encryption and decryption:

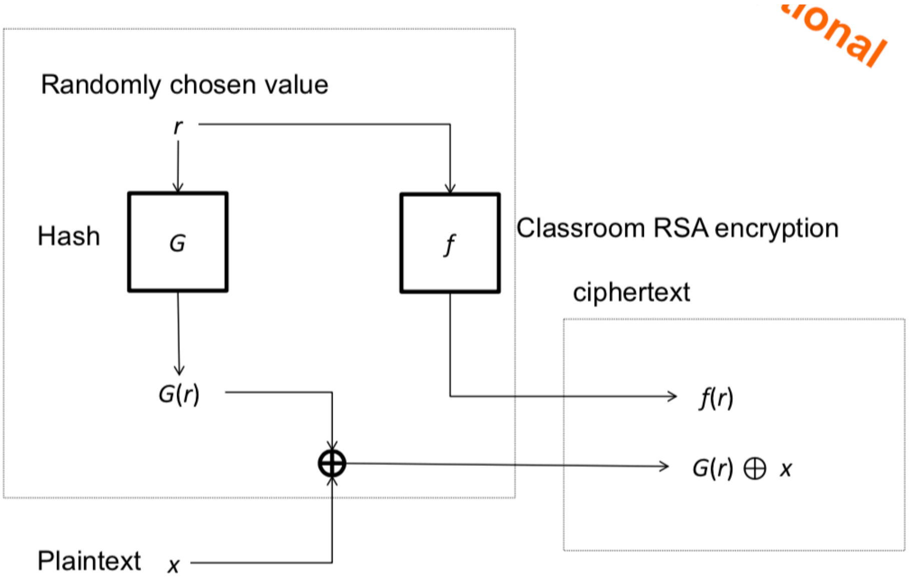
(e,n) is the public key, (d,n) the private one.

- To encrypt a message m, compute $c \equiv m^e \bmod n$.
- To decrypt a ciphertext c, compute $m \equiv c^d \bmod n$.

However, RSA has the same problem that symmetric-key encryption faces, which is that the encryption of the same plaintext at different times will give the same ciphertexts. Thus, some form of IV or padding is required to introduce an element of randomness.

The standard Public-Key Cryptography Standards (PKCS) #1 adds “optimal padding” to achieve the above.

ional



Issues that RSA faces

Poor Efficiency and Performance

- RSA is significantly slower than AES (10,000x slower)
- A 128-bit AES has the same key strength as a 3072-bit RSA

To overcome this issue:

When a large file F is to be encrypted under the public-key setting, for efficiency, the following steps can be carried out:

1. Randomly choose an AES key k
2. Encrypt F using AES with k as the key to produce the ciphertext C
3. Encrypt k using RSA to produce the ciphertext q
4. The final ciphertext consists of two components: (q, C)

The reverse is to be done for decryption:

1. Decrypt q using RSA to produce the key k
2. Decrypt C using AES with k as the key to produce plaintext F

Security of RSA

RSA is not necessarily “more secure” than AES. It can be shown that getting the private key from the public key is as difficult as factorization. But it is not known whether the problem of getting the plaintext from the ciphertext and public key is as difficult as factorization.

As mentioned before, the “textbook” RSA has to be modified so that different encryptions of the same plaintext lead to different ciphertexts, and such modifications are not straightforward (e.g. PKCS#1).

Strengths of PKC

The main strength is the “public-key” setting, which allows an entity in the public to perform an encryption without a pre-established pair-wise secret key. This “secret-key-less” feature is also very useful for providing authentication. In practice, PKC is rarely used to encrypt a large data file.

Cryptographic Hash

A hash is a function that takes in an arbitrarily long message as input and outputs a fixed-size **digest**.

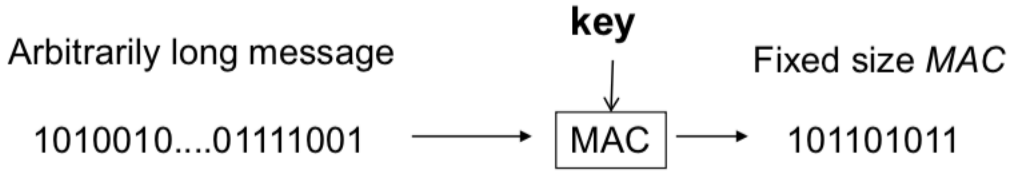
This is basically the hash function that we have learnt before.

Security Requirements

- Preimage Resistant or One-way
  - Given a digest d, it is difficult to find a m such that h(m) = d
  - In other words, it is difficult to reverse-engineer
- Second-preimage resistant
  - Given m_1, it is difficult to find a second preimage $m_1 \neq m_2$ such that $h(m_1) = h(m_2)$
- Collision-resistant
  - It is difficult to find two different messages $m_1 \neq m_2$ that "hashes" into the same digest, i.e. $h(m_1) = h(m_2)$

Message Authentication Code (MAC) aka Keyed-Hash

A keyed-hash is a function that takes an arbitrarily long message and a secret key as input, and outputs a fixed-size MAC.



Security Requirements

- Without knowing the key, it is difficult to forge the MAC

Popular Hashes

SHA-0, SHA-1, SHA-2, SHA-3

History:

1. **SHA-0** was published by NIST in **1993**.
   a. It produces a 160-bits digest.
   b. It was withdrawn shortly after publication and superseded by the revised version SHA-1 in 1995.
   c. In 1998, an attack that finds collision of SHA-0 in 261 operations was discovered. (Simply using the straight forward birthday attack, a collision can be found in 2160/2 = 280 operations).
   d. In 2004, a collision was found, using 80,000 CPU hours.
   e. In 2005, Wang Xiaoyun et al. (Shandong University) gave an attack that can find collisions in 239 operations.
2. **SHA-1** is a popular standard. It produces 160-bits message digest. It is employed in SSL, SSH, etc.
   a. In 2005, Xiaoyun Wang et al. gave a method of finding collision in SHA-1 using 269 operations, which was later improved to 263.
   b. In Feb 2017, researchers from CWI and Google announced the first successful SHA1 collision (https://shattered.io) – see the next 2 slides
   c. In 2001, NIST published SHA-224, SHA-256, SHA-384, SHA-512, collectively known as SHA-2.
   d. The number in the name indicates the digest length.

e. No known attack on full SHA-2 but there are known attacks on “partial” SHA-2, for e.g. attack on a 41-round SHA-256 (whereas the full SHA-256 takes 64 rounds).

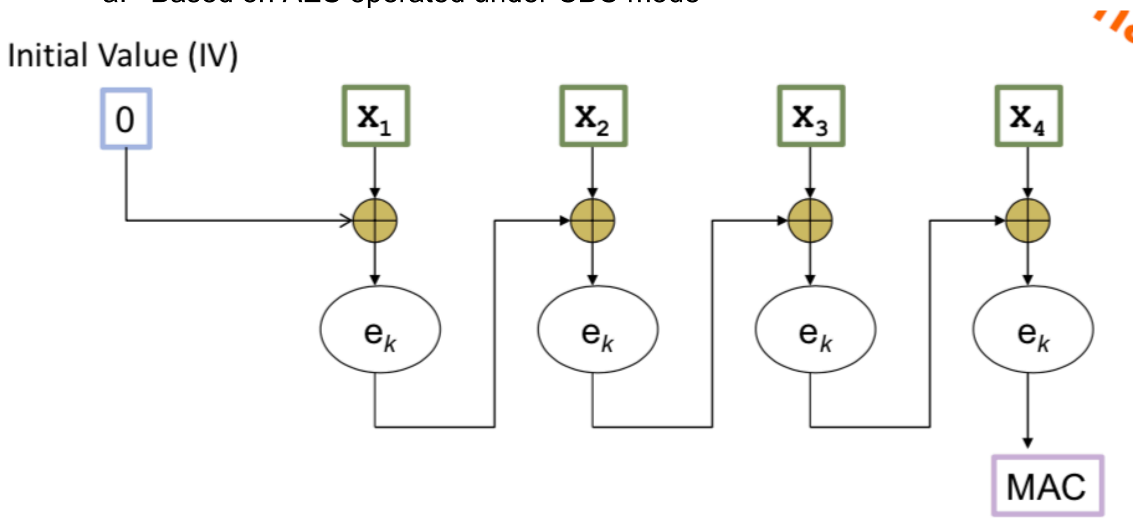
f. In Nov 2007, NIST called for proposal of SHA-3. In Oct 2012, NIST announced the winner, Keccak (pronounced “catch-ack”).

Popular but Obsolete Hashes

- **MD5**
    a. Designed by Rivest, who also invented MD, MD2, MD3, MD4, MD6.
    b. MD6 was submitted to NIST SHA-3 competition, but did not advance to the second round of the competition.
    c. MD5 was widely used. It produces 128-bit digest.
    d. In 1996, Dobbertin announced a collision of the compress function of MD5.
    e. In 2004, collision was announced by Xiaoyu Wang et al. The attack was reported to take one hour.
    f. In 2006, Klima give an algorithm that can find collision within one minute on a single notebook!
    g. Security implication: *Do not* use MD5!

Popular Keyed-Hash (MAC)

1. CBC-MAC
    a. Based on AES operated under CBC mode



2. HMAC
    a. Based on any iterative cryptographic hash function (e.g. SHA)
    b. Hashed-based MAC
    c. Standardized under RFC 2104
    d. Still deterministic, however, despite its more implicated operations

$$\text{HMAC}_k(x) = \textbf{SHA-1}\left((K \oplus opad) \,||\, \textbf{SHA-1}\left((K \oplus ipad) \,||\, x \right) \right)$$

where:

opad = 3636...36 (outer pad)

ipad = 5c5c...5c (inner pad)

(Note: the values above are in hexadecimal)

|| *means concatenation*

Data Integrity

How do we know if an email we received or a software we downloaded is authentic? Or that a file F that we need is authentic?

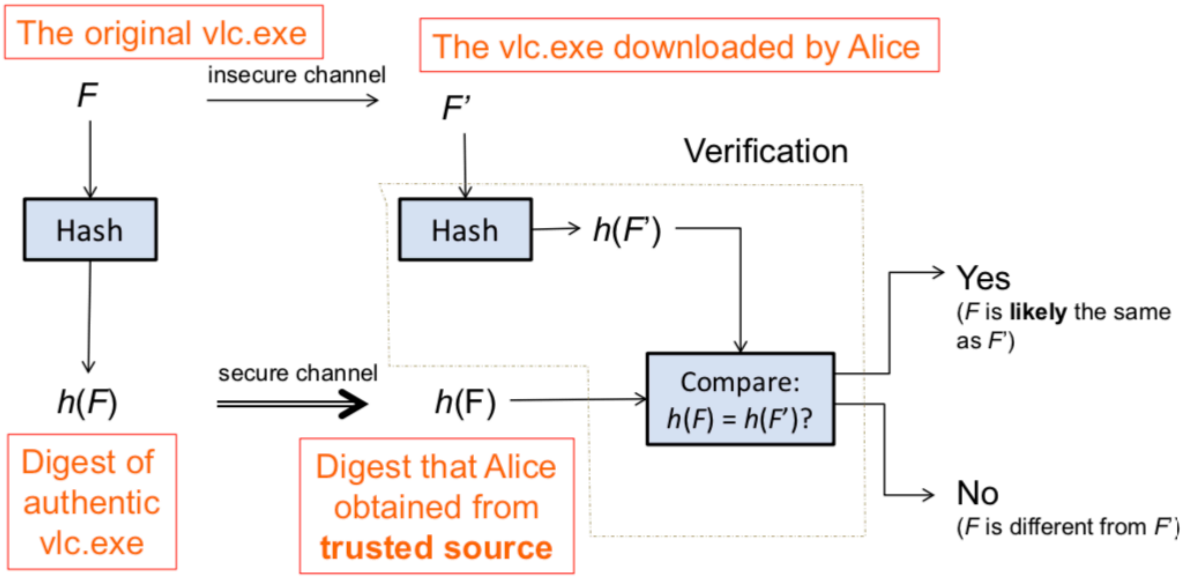
Unkeyed Hash for Integrity Protection

Let us assume that there is a secure channel to send a short piece of information. We can then carry out the following steps:

- Let F be the original file
- We obtain the digest h(F) from the secure channel
- We then obtain the file, F', whose origin claims that it is F
- We can compute and compare the two digests h(F) and h(F')
    a. If h(F) = h'(F), then F = F' (with a very high confidence)
    b. Else if h(F) ≠ h'(F), then the file integrity is compromised

We may even argue that with the digest, the verifier can be assured that the data is authentic, and thus the authenticity of the data origin is achieved. Nonetheless, in many literature and documents, when there is no secret key involved, the hash function only provides integrity, not authenticity. There are even sources that argue that even integrity is not ensured by unkeyed hash.

This is because a man-in-the-middle can potentially edit the original message before rehashing it using the same hash function, then sending the new digest over. This is why there is a need for the digest to be sent separately via some secure channel e.g. digest posted on a webpage which is sent over HTTPS, which many would argue defeats the purpose of ensuring integrity already.

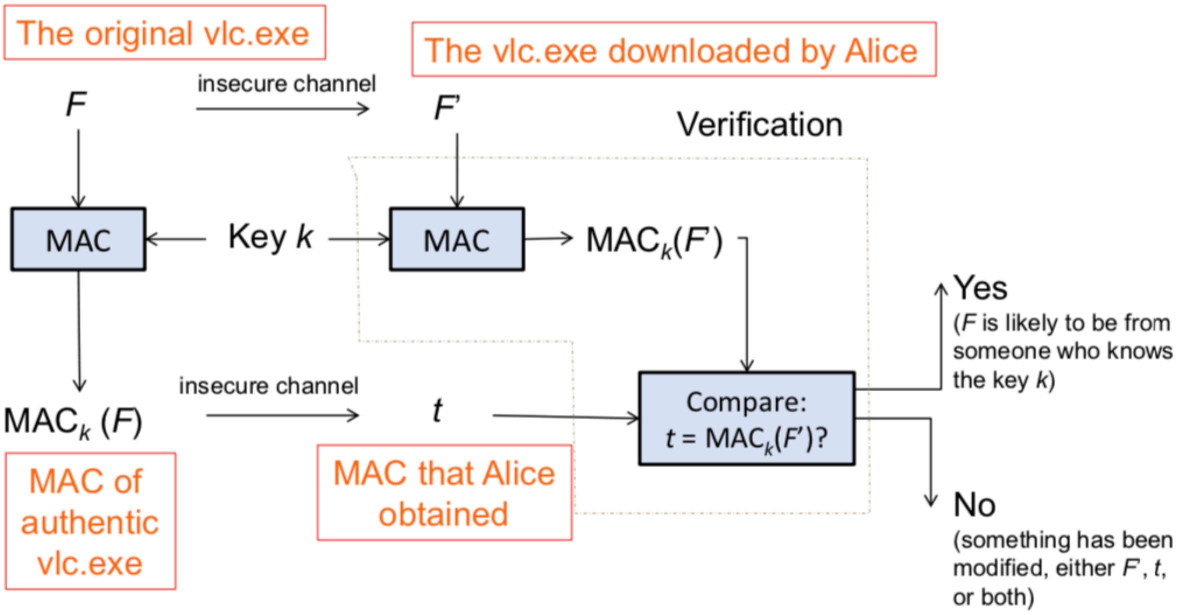


Data-Origin Authenticity

The previous example of using an unkeyed hash can have us obtaining a digest that is faked. So what can we do when we do not have a secure channel to deliver the digest?

Message Authentication Code

MAC can help to ensure integrity and authenticity as only the sender will know of the secret key used for the MAC. By comparing $MAC_k(F)$ (which can be sent through an insecure channel) and $MAC_k(F')$, we can easily determine if the file is from the right person. If the MAC matches, F is likely to be from someone who knows the key k. Else, something has been modified, either F', the $MAC_k(F)$ sent through the insecure channel, or both.



Note that there are actually no concerns about confidentiality for this situation. The file F can be sent in clear. Usually, the MAC is appended to F and the two are stored in a single file or transmitted together through a communication channel. Hence, the MAC is also called the **authentication tag**. Later, an entity who wants to verify the authenticity of F can carry out the verification process using the secret key.

Digital Signature

Digital Signature is essentially the asymmetric-key version of MAC. The steps are as follows:

- The original file, F, is first hashed using a hash function
- The hash value is then passed through the signature function together with the private or signature key k_{PRIV} to produce signature s, the signature of the authentic file
- The signature is then appended to the original file F and sent to the recipient via an insecure channel
- The recipient then passes the received signature s' and the public or verification key k_{PUB} into the verification function
- The verification function will produce a hash value
- The file F' is then pass through a hash function to return a hash value
- The hash values are compared to see if the file has been modified, or if the digital signature is valid
- Since digital signature is created by 'private' key of signer and no one else can have this key; the signer cannot repudiate signing the data in future

The computed signature is typically appended to F and stored as a single file. Subsequently, the authenticity of F can be verified by anyone who knows the signer's public key. This is because the valid signature can be computed only be someone knowing the private key.

In addition to authenticity, signature scheme also achieves non-repudiation: assurance that someone cannot deny his/her previous commitments or actions.

Why hash the file before signing? (Not tested)
This is due to the greater efficiency. As the hash value is a unique representation of the data, it is sufficient to sign the hash in place of data. Let us assume RSA is used as the signing algorithm. As discussed in public key encryption chapter, the encryption/signing process using RSA involves modular exponentiation.

Signing large data through modular exponentiation is computationally expensive and time consuming. The hash of the data is a relatively small digest of the data, hence signing a hash is more efficient than signing the entire data.

Attacks and Pitfalls

Birthday Attack on Hash

The birthday paradox has us seeing 23 people being the minimum number of people required before the probability of any pair of students sharing the same birthday exceeds 50%. The reason for this unintuitive number is due to us not looking at comparing one individual's birthday with the rest, but about comparing between every possible pair of students.

This can be applied to the hash function. Suppose we have M messages, and each message is tagged with a value randomly chosen from {1, 2, 3, 4, ...,T}.

$$M > 1.17\sqrt{T}$$

If M >. 1.17 $T^{0.5}$, then there is >50% chance of a pair of messages being tagged to the same vzlue. This result is the mathematics behind a **birthday attack** on a hash function, as the arobability of finding a collision is rather high. Generally, the probability that a collision occurs can be approximated with 1 – exp(-M^2 / 2T).

$$1 - e^{-M^2/2T}$$

For the birthday attack variant, we are looking at calculating the probability of a random set of elements sharing at least one element with another set of distinct elements, with both sets being subsets of an even larger set. Let the second set be S, with k distinct elements, where each element is a n-bit binary string. We now select m random n-bit binary strings. The probability that at least one of the randomly chosen strings is in S is larger than:

$$1 - 2.7^{-km2^{-n}}$$

This has a serious consequence on the digest length required by a hash function to be collision resistant. When the key length of a symmetric key is 112, the corresponding recommended length for digest is at least 224.

Using Encryption for Authenticity

It is common for people to claim that their communication channel is secure as they use a certain encryption scheme that provides a high level of security. This is in fact a false sense of security, as encryption schemes merely provide confidentiality. However, it does not provide the integrity and authenticity required for communication channels.

For example, a server that communicates instructions via SMS does not provide integrity and authenticity if it simply encrypts its messages. A secure design could use MAC instead of encryption.

Even then, there are opportunities for "replay attacks". For example, if I intercept one of the messages, even with a MAC appended, I can simply repeatedly send the message I intercepted, which will result in some command or instruction being repeatedly executed. To prevent replay attacks, a cryptographic nonce is required, which is a random or pseudo-random number issued that prevents old communication from being reused.

Hashed and Salted Passwords

Covered before under passwords.

Tutorial Learnings

Pseudo Randomness of Hash Functions

Cryptographic hash functions, such as SHA-1, are often employed to generate “pseudo-random” numbers. The SHA hash function family is thus deterministic if the seed is known. Thus any form of key or IV generation with hash functions and a known seed is not secure, as any attacker can simply repeat the same process to obtain the exact same key and IV.

Using a Random Function that Depends on Time

Assume

```
srand(time(NULL));
int s = rand();
```

If an adversary knows the time, which is possible in practice, then he/she can derive the key. Otherwise, if the adversary knows only the approximate time, still he/she can exhaustively search all possible times.

Even if the adversary knows nothing about the time, it is still possible to brute force the variable s. This is since int data type in C is only either 2-byte (16-bit) or 4-byte (32-bit) long depending on the platforms used.

More can be found in the section on Encryption.

Hashing IVs to Get Keys

Highly insecure, since IVs are known by the attacker. Let’s say we generate a random number in a secure manner, then we hash it to get a 160-bit string. We use the first 128-bits as the IV, then we hash the 160-bit string again to get another 160-bit string, which we use the first 128-bits of as the key.

Now it seems like it may be secure, but with knowledge of the IV, we just need to generate all possibilities of the remaining 32-bits of the first 160-bit string i.e. 2^{32} possibilities. Then we hash it and check if the 128-bit key can decrypt the ciphertext.

Requirements of Public Keys

Two requirements must be met:

1. It is difficult to get the private key from the public key.
2. It is difficult to get the plaintext from the public key and the ciphertext.

Many people argue that RSA is secure just because of requirement 1, but that is an incomplete justification.

Using HMAC with the same key for plaintexts

This allows attackers to actually observe the MACs of two plaintext blocks to infer about the plaintext blocks. If the MACs are the same, then the plaintext must be identical.

Collision-resistant(h) → one-way(h)?

To prove this, we can prove the contrapositive, i.e. ¬one-way(h) → ¬collision-resistant(h). To show this, suppose that there is a fast algorithm A that, when given x, can easily find a m such that h(m) = x, i.e. h is not one-way since it is not computationally difficult to get a m from x.

Then we can show that we can easily derive another fast algorithm B that can find a collision with high probability, i.e. B can find m_1, m_2 such that $h(m_1) = h(m_2)$ and $m_1 \neq m_2$. We would then have proven that if h is not one-way, it is also not collision resistant.

B is as such:

- Randomly pick a m
- Compute x = h(m)
- Compute the inverse m' = A(x)
- If m' ≠ m, then return (m, m'), else repeat.

There is a good probability that algorithm B will soon stop, since the set of hash values is finite, and the set of messages is much larger than the set of hash values. Therefore, many different messages would be hashed into a single hash value, i.e. collisions should occur frequently.

Example Question for Birthday Attack Variant

Consider this scenario. There are 2^7 = 128 students in the class. Each student is assigned a secret 16-bit ID, which is known only by the student and the lecturer. The probability of correctly guessing the ID of a particular student is thus 2^{-16}, which is very small. One day, the lecturer posted a multiple choice question during the lecture, and asked each student to write down the answer on a piece of paper together with his/her 16-bit ID, and insert it into a box in the lecture hall.

Suppose you know the correct answer, and want to generously share it with your classmates. You quickly write down the correct answer on 32 pieces of paper, each with a randomly chosen ID, and covertly insert them into the box.

What is the probability that at least one student benefits from your attempted good deed?

Let n = 16, k = 2 , and m = 2. By applying the given formula,

$$1 - 2.7^{-km2^{-n}}$$

we have the probability ≈ 0.06 (which is low).

How many pieces of paper do you need to submit so that the probability is more than 0.5?

We need to find m such that the above expression equals to 0.5. We thus get m ≈ 362. Hence, you need to submit more than 362 pieces of paper.

Note: There is also a quick approximation to find m simply by setting km = 2^n, so that the probability becomes $1 - 2.7^{-1} = 0.63$. We thus can find the approximate m = $2^{16-7} = 2^9$ = 512.

This approximation technique is useful in case you don't have a calculator with you, or you just want to have an approximate m.