HASH FUNCTIONS

Hash Function: It is any functions that takes an input of any size and produce an output of a fixed size. A ***cryptographic hash function h(x)*** must provide all of the following.

• **Compression** — For any size input x, the output length of $y = h(x)$ is small. In practice, the output is a fixed size, regardless of the length of the input.

• **Efficiency** — It must be easy to compute $h(x)$ for any input x. The computational effort required to compute $h\{x)$ will grow with the length of x, but it cannot grow too fast.

• **One-way** — Given any value y, it's computationally infeasible to find a value a; such that $h\{x) = y$. Another way to say this is that there is no feasible way to invert the hash.

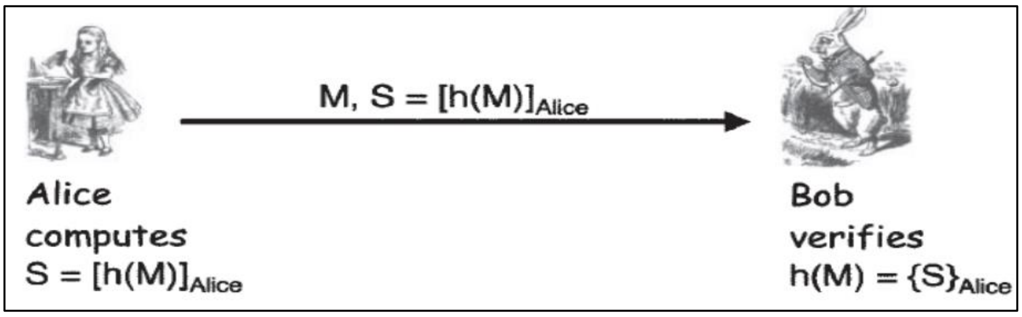
• **Weak collision resistance** — Given x and $h(x)$, it's infeasible to find any y, with $y \neq x$, such that $h(y) = h(x)$. Another way to state this requirement is that it is not feasible to modify a message without changing its hash value.

• **Strong collision resistance** — It's infeasible to find any x and y, such that $x \neq y$ and $h(x) = h(y)$. That is, we cannot find any two inputs that hash to the same output.

Hash functions used in the computation of **digital signatures**.

Motivation:

- Alice signs a message M by using her private key to "encrypt," that is, she computes $S = [M]_{Alice}$ · If Alice sends M and S to Bob, then Bob can verify the signature by verifying that $M = \{S\}_{Alice}$, if M is large, $[M]_{Alice}$ is costly to compute.
- If a cryptographic function h, Alice will sign M by first hashing M then signing the hash, that is, Alice computes $S = [h(M)]_{Alice}$. Hashes are efficient (comparable to block cipher algorithms), and only a small number of bits need to be signed.
- Then Alice can send Bob M and S, as illustrated in Figure. Bob verifies the signature by hashing M and comparing the result to the value obtained when Alice's public key is applied to S. That is, Bob verifies that $h(M) = \{S\}_{Alice}$·



The Birthday Problem:

- Suppose there are N people in a room.
- How large must N be before the probability someone has same birthday as me is>=1/2?
- Assuming all birth dates are equally likely.
- The probability that person does not have the same probability as you is given as:

1-(1/365)=364/365

- Probability that none of N people have the same birthday as you is

$(364/365)^N$

- The probability that at least one person has the same birthday as you is

$1-(364/365)^N$

- Solving: $1/2 = 1 - (364/365)^N$ for N. Therefore N = 253
- Number of people(N) that must be in a room before probability is ≥ 1/2 that any two (or more) have same birthday is

$1 - (365/365) \cdot (364/365) \cdots (365-N+1)/365 = 1/2$. Therefore N=23

- With N people in a room, the number of comparisons is N^2.
- Since there are 365 different birth dates, a match n be find at the point where $N^2=365$ or $N=\sqrt{365}=19$.
- If h(x) is N bits, then 2^N different hash values are possible.
- So, if you hash about $\text{sqrt}(2^N) = 2^{N/2}$ values then you expect to find a collision
- Secure N-bit hash requires $2^{N/2}$ work to "break" & Secure N-bit symmetric cipher has work factor of 2^{N-1}.

A Birthday Attack

- If M is the message that Alice wants to sign, then she computes $S = [h(M)]_{Alice}$ & sends S and M to Bob.
- Attacker selects an "evil" message E that she wants Alice to sign, but which Alice is unwilling to sign.
- Attacker also creates an innocent message I that she is confident Alice is willing to sign.
- Attacker generates $2^{n/2}$ variants of the innocent message. These innocent messages, which we denote I_i, *all have the same meaning as I, but since the* messages differ, their hash values differ.
- Attacker creates $2^{n/2}$ variants of the evil message, which denoted as E_i. *but their hashes differ.*
- By the birthday problem, attacker can expect to find a collision, $h(E_j) = h(I_k)$.
- Given such a collision, attacker sends I_k to Alice, & asks Alice to sign it.

- Alice signs it and returns I_k *and* $h[I_k]_{Alice}$ *to attacker.*
- *Since* $h(E_j) = h(I_k)$, it follows that $h[E_j]_{Alice} = h[I_k]_{Alice}$. Consequently attacker has obtained Alice's signature on the evil message E_j.
- To prevent this attack, choose a hash function for which *n*, the size of the hash function output, is so large that attacker cannot compute $2^{N/2}$ hashes.

Non-Cryptographic Hashes

- Consider data $X = (X_1,X_2,X_3,\ldots,X_n)$, each X_i is a byte.
- Defining hash function $h(X) = (X_1+X_2+X_3+\ldots+X_n) \bmod 256$. This provides compression, since any size of input is compressed to an 8-bit output. Hash would be easy to break, since the birthday problem tells us that if we hash just $2^4 = 16$ randomly selected inputs, we can expect to find a collision.

  For example: swapping two bytes will always yield a **collision**, such as X = (10101010, 00001111), Hash is h(X) = 10111001. If Y = (00001111, 10101010) then h(X) = h(Y).

  i.e h(10101010,00001111) = h(00001111,10101010) = 10111001.
- Consider data $X = (X_0,X_1,X_2,\ldots,X_{n-1})$
- Suppose hash is defined as $h(X) = (nX_1+(n-1)X_2+(n-2)X_3+\ldots+2\cdot X_{n-1}+X_n) \bmod 256$. It gives different results when the byte order is swapped.

  For example: h(10101010,00001111) $\neq$ h(00001111,10101010)
- But there exists birthday problem issue and it also happens to be relatively easy to construct collisions.

For example: h(00000001,00001111) = h(00000000,00010001) = 00010001.

- This is not a secure cryptographic hash, it's useful in a particular non-cryptographic application known as **Rsync.**
- Cyclic Redundancy Check is the remainder in a long division calculation, good for detecting burst **errors** and such random errors unlikely to yield a collision.
- CRC has been mistakenly used where crypto integrity check is required (e.g., WEP).

Tiger Hash

- “Fast and strong”
- Designed by Ross Anderson and Eli Biham – leading cryptographers
- Design criteria:

- o Secure
- o Optimized for 64-bit processors.
- o Easy replacement for MD5 or SHA-1.

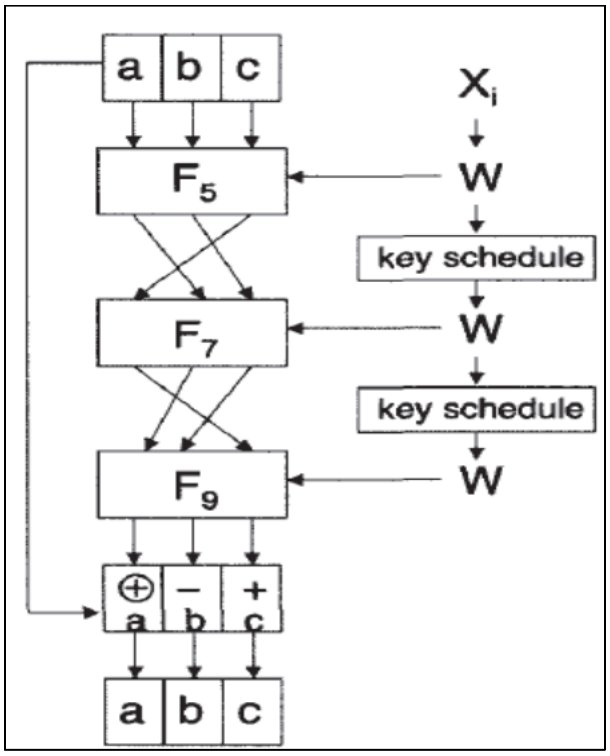
- ● Input to Hash function is divided into 512 bit blocks (padded).
- ● Output is **192 bits** (three 64-bit words).
- ● Intermediate rounds are all 192 bits
- ● 4 S-boxes(Substitution-box) is used, each maps 8 bits to 64 bits.
- ● A "key schedule" is used, since there is no key, is applied to the input block.

Tiger Outer Round:

- ➢ The input *X* is padded to a multiple of 512 bits and written as

$$X = (X_0, X_1, \ldots, X_{n-1})$$

- ➢ Employs one outer round for each X_i
- ➢ Initial (a,b,c) constants.
- ➢ The final (a, b, c) output from one round is the initial triple for the subsequent round and the final (a, *b,* c) from the final round is the 192-bit hash value.



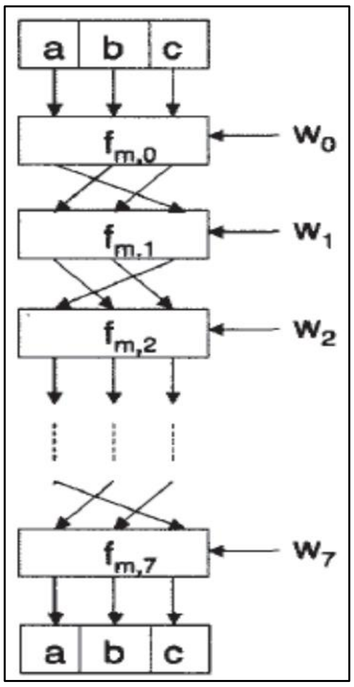
Tiger Outer Round

- ➢ In Outer round, input to outer round F5 is (a,b,c).
- ➢ The output of F5 as (a,b,c), the input toF7 is (c,a,b), the input to F9 is (b,c,a).
- ➢ Each function F_m consists of eight inner rounds.

Tiger Inner Rounds:

- Each F_m consists of precisely 8 inner rounds.
- 512 bit input W to F_m
  - $W=(w_0,w_1,\dots,w_7)$
  - W is one of the input blocks X_i
- All lines are 64 bits
- The input values for $f_{m,i}$, for i=0,1,2,…,7 are

$$\boxed{(a,b,c),(b,c,a),(c,a,b),(a,b,c),(b,c,a),(c,a,b),(a,b,c),(b,c,a)}$$



Tiger Inner Round for F_m

Tiger Hash: One Round

- Each $f_{m,i}$ is a function of a,b,c,w_i and m
  - Input values of a,b,c from previous round.
  - And w_i is 64-bit block of 512 bit W.
  - Subscript m is mul tiplier
  - And $c = (c_0,c_1,\dots,c_7)$
- Output of $f_{m,i}$ is
  - $c = c \oplus w_i$
  - $a = a - (S_0[c_0] \oplus S_1[c_2] \oplus S_2[c_4] \oplus S_3[c_6])$
  - $b = b + (S_3[c_1] \oplus S_2[c_3] \oplus S_1[c_5] \oplus S_0[c_7])$
  - $b = b * m$
- Each S_i is S-box(i.e., lookup table): 8 bits mapped to 64 bits.

Tiger Hash Key Schedule:

- Input is X
  - o $X=(x_0,x_1,\ldots,x_7)$
- Small change in X will produce large change in key schedule output.

$$\begin{aligned}
w_0 &= w_0 - (w_7 \oplus \texttt{0xA5A5A5A5A5A5A5A5}) \\
w_1 &= w_1 \oplus w_0 \\
w_2 &= x_2 + w_1 \\
w_3 &= w_3 - (w_2 \oplus (\bar{w}_1 \ll 19)) \\
w_4 &= w_4 \oplus w_3 \\
w_5 &= w_5 + w_4 \\
w_6 &= w_6 - (w_5 \oplus (\bar{w}_4 \gg 23)) \\
w_7 &= w_7 \oplus w_6 \\
w_0 &= w_0 + w_7 \\
w_1 &= w_1 - (w_0 \oplus (\bar{w}_7 \ll 19)) \\
w_2 &= w_2 \oplus w_1 \\
w_3 &= w_3 + w_2 \\
w_4 &= w_4 - (w_3 \oplus (\bar{w}_2 \gg 23)) \\
w_5 &= w_5 \oplus w_4 \\
w_6 &= w_6 + w_5 \\
w_7 &= w_7 - (w_6 \oplus \texttt{0x0123456789ABCDEF})
\end{aligned}$$

Summary:

- Hash and intermediate values are 192 bits.
- 24 (inner) rounds:
  - o **S-boxes:** Claimed that each input bit affects a, b and c after 3 rounds.
  - o **Key schedule:** Small change in message affects many bits of intermediate hash values.
  - o **Multiply:** Designed to ensure that input to S-box in one round mixed into many S-boxes in next.
- S-boxes, key schedule and multiply together designed to ensure strong **avalanche** effect.

Note: A desirable property of any cryptographic hash function is the so-called ***avalanche effect***. The goal is that any small change in the input should cascade and cause a large change in the output.

- At a higher level, Tiger employs
  - o Confusion
  - o Diffusion

HMAC

For message integrity we can compute a message authentication code, or **MAC**, where the MAC is computed using a block cipher in CBC mode. The MAC is the final encrypted block, which is also known as the ***CBC residue***. Since a hash function effectively gives us a fingerprint of a file, we should also be able to use a hash to verify message integrity.

Motivation:

Consider Alice protect the integrity of M by simply computing *h(M)* and sending both M and *h(M)* to Bob.If *M* changes, Bob will detect the change, provided that *h(M)* has not changed (and vice versa). However, if attacker replaces *M* with *M'* and also replaces *h(M)* with *h(M')*, then Bob will have no way to detect the tampering. But using a hash function to provide integrity protection, involves a key to prevent attacker from changing the hash value.

Approach: Alice encrypt the hash value with a symmetric cipher, *E(h(M),K),* and send this to Bob. A slightly different approach is used to compute a ***hashed MAC,* or HMAC.** Instead of encrypting the hash, directly mix the key into *M* when computing the hash.

Two approaches are to prepend the key to the message, or append the key to the message:

- ***h(K,M)***
- ***h(M,K)***

h(K,M) :

If *h(K,M) is used to* compute an HMAC, then consider cryptographic hashes hash the message in blocks—for MD5, SHA-1, and Tiger, the block size is 512 bits. As a result, if M = *{B1, B2),* where each *Bi* is 512 bits, then

h(M) = F(F(A, B1),B2) = F(h(B1), B2) ....equation (1)

for some function *F*, where *A* is a fixed initial constant.

For example, in the Tiger hash, the function *F* consists of the outer with each *Bi* corresponding to a 512-bit block of input and *A* corresponding to the 192-bit initial value *{a,b,c).*

If an attacker chooses *M'* so that *M'* = *(M,X), attacker* might be able to use equation (1) to find *h(K,M')* from *h(K,M)* without knowing *K* since, for *K, M,* and *X* of the appropriate size,

h(K, M') = h(K,* M*, X) = F(h(K, M),X)

where the function *F* is known.

h(M,K):

If it should happen that there is a known collision for the hash function *h*, that is, if there exists some *M'* with *h(M')* = *h(M)*, then by equation (1), then

h(M, K) = F(h(M), K) = F(h(M'), K) = h(M', K)

provided that *M* and *M'* are each a multiple of the block size.

Conclusion: If such a collision exists, the hash function is considered insecure.

HMAC:

Approved method to mix the key into the hash for computing an HMAC is as follows.

Let *B* be the block length of hash, in bytes. For all popular hashes (MD5, SHA-1, Tiger, etc.), *B* = 64. Next, define

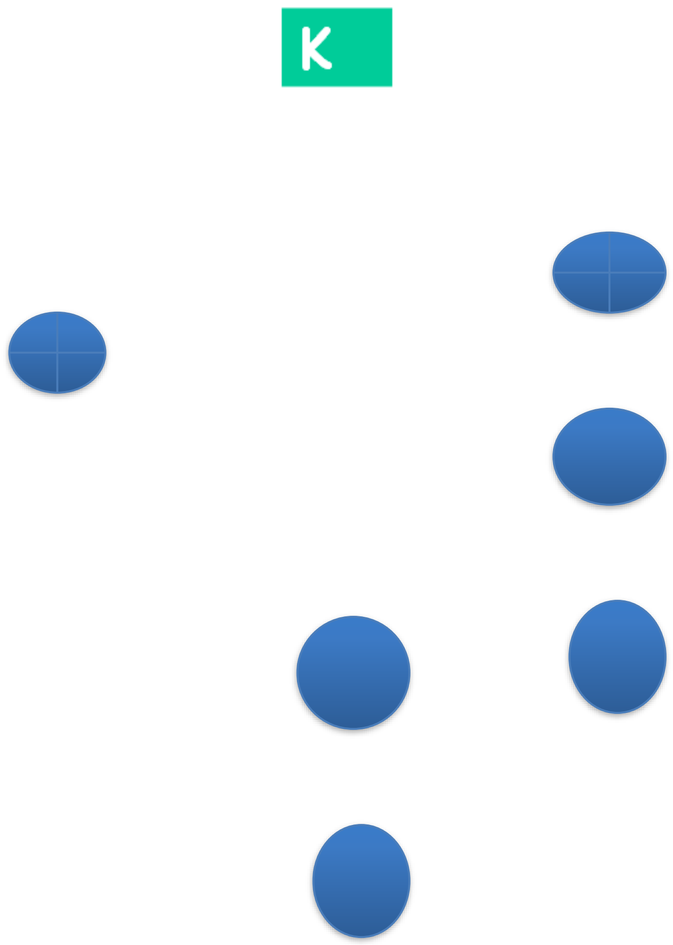
ipad = 0x36 repeated B times

&

opad = 0x5C repeated *B* times

Then the HMAC of *M* is defined to be

$$\boxed{\text{HMAC}(M, K) = H(K \oplus \text{opad}, H(K \oplus \text{ipad}, M))}$$



This approach thoroughly mixes the key into the resulting hash. While two hashes are required to compute an HMAC, note that the second hash will be computed on a small number of bits—the output of the first hash with the modified key appended.

5.8 Uses for Hash Functions:

Hash functions include authentication, message integrity (using an HMAC), message fingerprinting, error detection, and digital signature efficiency and can also be used to solve security-related problems.

5.8.1 Online Bids:

- Consider suppose an item is for sale online and Alice, Bob, and Charlie all want to place bids and these are supposed to be sealed bids, that is, each bidder gets one chance to submit a secret bid and only after all bids have been received are the bids revealed.
- The highest bidder wins. Alice, Bob, and Charlie don't necessarily trust each other and they definitely don't trust the online service that accepts the bids.
- Each bidder is concerned that the online service might reveal their bid to the other bidders—either intentionally or accidentally.

For example,

- Suppose Alice places a bid of $10.00 and Bob bids $12.00. If Charlie is able to discover the values of these bids prior to placing his bid (and prior to the deadline for bidding), he could bid $12.01 and win. The point is that nobody wants to be the first (or second) to place their bid, since there might be an advantage to bidding later.

So, the online service proposes the following scheme.

- Each bidder will determine their bids, say, bid *A* for Alice, bid *B* for Bob, and *C* for Charlie, keeping their bids secret.
- Then Alice will submit *h{A),* Bob will submit *h(B),* and Charlie will submit *h(C).*
- Once all three hashed bids have been received, the hash values will be posted online for all to see.
- At this point all three participants will submit their actual bids, that is, *A*, *B*, and *C*.

Advantage: If the cryptographic hash function is secure, it's one-way, so no disadvantage to submitting a hashed bid prior to a competitor. And since it is infeasible to determine a collision, no bidder can change their bid after submitting their hash value.
i.e, the hash value binds the bidder to his or her original bid, without revealing any information about the bid itself. If there is no disadvantage in being the first to submit a hashed bid, and there is no way to change a bid once a hash value has been submitted, then this scheme prevents the cheating.

Disadvantage: It is subject to a forward search attack.

5.8.2 Spam Reduction

- Spam is defined as unwanted and unsolicited bulk email.
- In this scheme, Alice will refuse to accept an email until she has proof that the sender expended sufficient effort to create the email. Effort will be measured in terms of computing resources, in particular, CPU cycles.
- This scheme would not eliminate spam, but it would limit the amount of such email that any user can send.
- Let *M* be an email message and let *T* be the current time. The message *M* includes the sender's and intended recipient's email addresses, but does not include any additional addresses.
- The sender of message *M* must determine a value *R* such that

$$\boxed{h(M,R,T) = (\underbrace{00\ldots0}_{N}, X)}$$

- That is, the sender must find a value *R* so that the hash in equation (5.5) has zeros in all of its first *N* output bits.
- Once this is done, the sender sends the triple *(M,R,T).* Before Alice, the recipient, accepts the email, she needs to verify that the time *T* is recent, and that *h(M, R, T)* begins with *N* zeros.
- Again, the sender chooses random values *R* and hashes each until he finds a hash value that begins with TV zeros. Therefore, the sender will need to compute, on average, about 2 hashes.
- The recipient can verify that *h(M, R, T)* begins with *N* zeros by computing a single hash. So the work for the sender (measured in terms of hashes) is about 2^N, while the work for the recipient is always a single hash.
- In this scheme, we would need to choose *N* so that the work level is acceptable for normal email users but unacceptably high for spammers.
- It might also be possible for users to select their own individual value of *N* to match their personal tolerance for spam.

For example, if Alice hates spam, *N* = 40. While this would deter spammers, it might also deter many legitimate email senders. If Bob, doesn't mind receiving some spam and he never wants to deter a legitimate email sender, he might set his value to, *N* = 10. Spammers dislikes this scheme. Legitimate bulk emailers also might not like this scheme, since they would need to spend resources (i.e., money) to compute vast numbers of hashes.

5.9 Miscellaneous Crypto-Related Topics

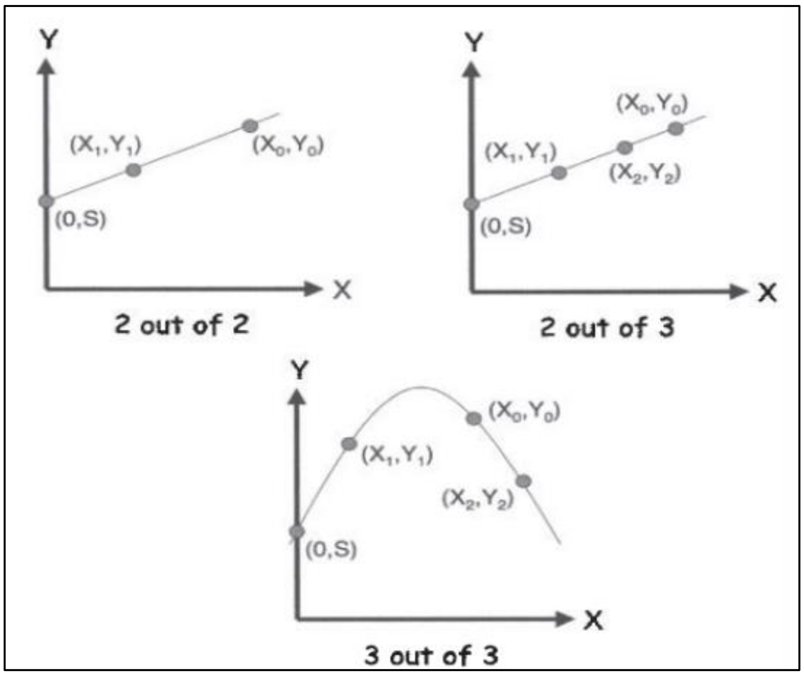
5.9.1 Secret Sharing

Suppose Alice and Bob want to share a secret *S* in the sense that:

• Neither Alice nor Bob alone (nor anyone else) can determine *S* with a probability better than guessing.

• Alice and Bob together can easily determine *S*.

Suppose the secret *S* is a real number. Draw a line *L* in the plane through the point (0, *S)* and give Alice a point $A = (X_0, Y_0)$ on *L* and give Bob another point $B = (X_1, Y_1)$, which also lies on the line *L*. Then neither Alice nor Bob individually has any information about *S*, since an infinite number of lines pass through a single point. But together, the two points *A* and *B* uniquely determine *L*, and therefore the y-intercept, and hence the value *S*. This example is illustrated in the "2 out of 2" scheme which is given below:



Secret Sharing Schemes

It's easy to extend this idea to an "m out of n" secret sharing scheme, for any $m < n$, where *n* is the number of participants, any *m* of which can cooperate to recover the secret. For $m = 2$, a line always works. For example, a "2 out of 3" scheme appears in Figure.

A line, which is a polynomial of degree one, is uniquely determined by two points, whereas a parabola, which is a polynomial of degree two, is uniquely determined by three points. In general, a polynomial of degree $m - 1$ is uniquely determined by m points. This elementary fact is allows us to construct an m out of *n* secret sharing scheme for any $m < n$. For example, a "3 out of 3" scheme is illustrated in Figure.

5.9.1.1 Key Escrow

One particular application where secret sharing would be useful is in the *key escrow* problem . Suppose that we require users to store their keys with an official escrow agency. The government could then get access to keys as an aid to criminal investigations.

- One concern with key escrow is that the escrow agency might not be trustworthy.

- It is possible to ameliorate this concern by having several escrow agencies and allow users to split the key among *n* of these, so that m of the *n* must cooperate to recover the key.
- Alice could select escrow agencies that she considers most trustworthy and have her secret split among these using an m out of *n* secret sharing scheme.

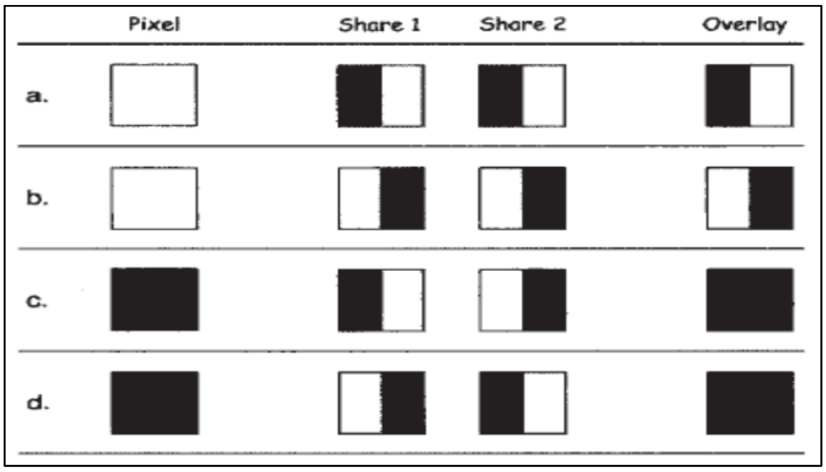
Shamir's secret sharing scheme could be used to implement such a key escrow scheme.

For example, suppose *n* = 3 and m = 2 and Alice's key is *S*. Then the "2 out of 3" scheme illustrated in above Figure could be used.

Alice might choose to have the Department of Justice hold the point = (X_0, Y_0), the Department of Commerce hold = (X_1, Y_1), and Fred's Key Escrow, Inc., hold (X_2,Y_2)· Then at least two of these three escrow agencies would need to cooperate to determine Alice's key *S*.

5.9.1.2 Visual Cryptography

- Visual secret sharing scheme is absolutely secure, as is the polynomial-based secret sharing scheme.
- In visual secret sharing (aka visual cryptography), no computation is required to decrypt the underlying image.
- In the simplest case, we start with a black-and-white image and create two transparencies, one for Alice and one for Bob.
- Each individual transparency appears to be a collection of random black and white subpixels, but if Alice and Bob overlay their transparencies, the original image appears (with some loss of contrast).
- In addition, either transparency alone yields no information about the underlying image.



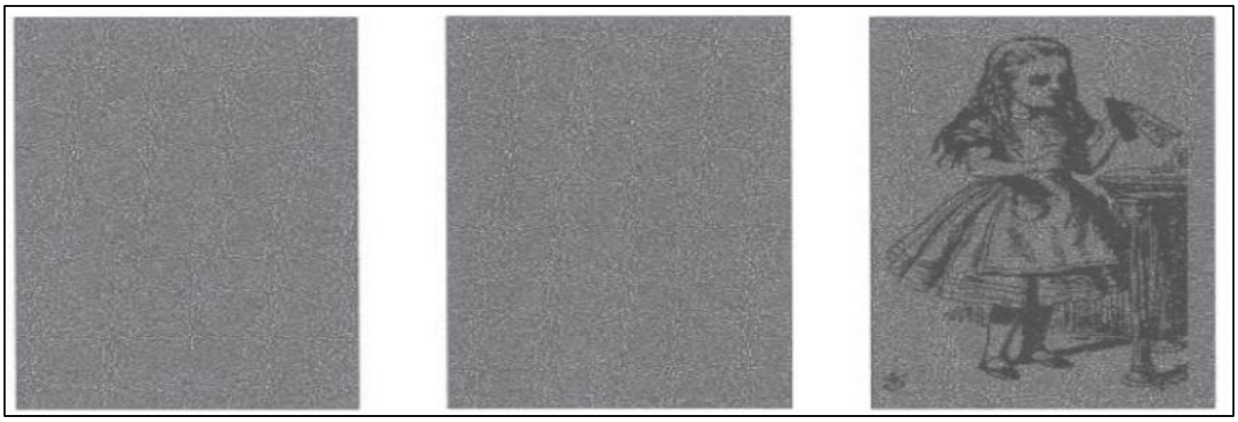
Pixel shares

- Above figure shows various ways that an individual pixel can be split into "shares," where one share goes to Alice's transparency and the corresponding share goes to Bob's.

For example,

- If a specific pixel is white, then we can flip a coin to decide whether to use row "a" or row "b" from above Figure. Then, Alice's transparency gets share 1 from the selected row (either a or b), while Bob's transparency gets share 2.
- The shares are put in Alice's and Bob's transparencies at the same position corresponding to the pixel in the original image. When Alice's and Bob's transparencies are overlaid, the resulting pixel will be half-black/half-white.
- In the case of a black pixel, we flip a coin to select between rows "c" and "d" and we again use the selected row to determine the shares.
- If the original pixel was black, the overlaid shares always yield a black pixel.
- If the original pixel was white, the overlaid shares will yield a half-white/half-black pixel, which will be perceived as gray.
- This results in a loss of contrast (black and gray versus black and white), but the original image is still clearly discernible.

For example, Below Figure illustrates a share for Alice and a share for Bob, along with the resulting overlaying of the two shares.



Alice's Share, Bob's Share, and Overlay Image

5.9.2 Random Numbers

In cryptography, random numbers are needed to generate symmetric keys, RSA key pairs (i.e., randomly selected large primes), and Diffie-Hellman secret exponents as well as in security protocols.

Random numbers are used in many non-security applications such as simulations and various statistical applications. In such cases, the random numbers usually only need to be statistically random.

Cryptographic random numbers must be statistically random and they must be unpredictable.

Consider the following example:

Suppose that a server generates symmetric keys for users. Suppose the following keys are generated for the listed users:

• K_A for Alice

• K_B for Bob

• K_C for Charlie

• K_D for Dave

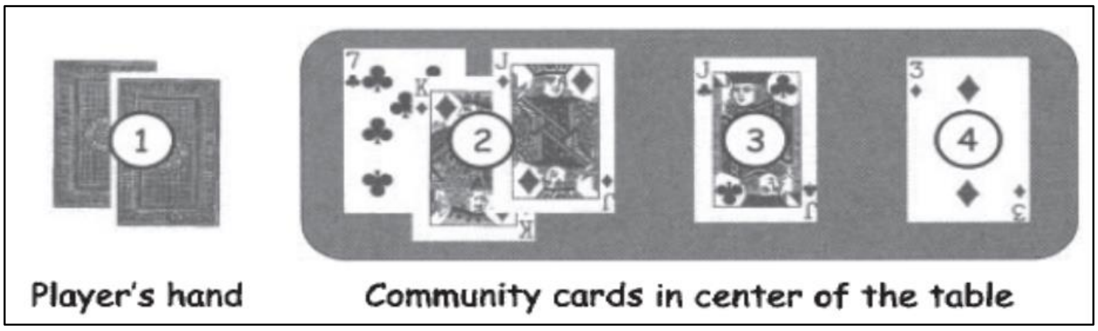
Now, if Alice, Bob, and Charlie don't like Dave, they can pool their information to see if it will help them determine Dave's key. That is, Alice, Bob, and Charlie could use knowledge of their keys, K_A, K_B, and K_C, to see if it helps them determine Dave's key K_D. If K_D can be predicted from knowledge of the keys K_A, K_B, and K_C, then the security of the system is compromised.

Commonly used pseudo-random number generators are predictable. Consequently, pseudo-random number generators are not appropriate for cryptographic applications.

5.9.2.1 Texas Hold 'em Poker

Consider a real-world example that illustrates the wrong way to generate random numbers:

- ASF Software, Inc., developed an online version of the card game known as Texas Hold 'em Poker .
- In this game, each player is first dealt two cards, face down. Then a round of betting takes place, followed by three community cards being dealt face up—all players can see the community cards and use them in their hand.
- After another round of betting, one more community card is revealed, then another round of betting.
- Finally, a final community card is dealt, after which additional betting can occur. Of the players who remain at the end, the winner is the one who can make the best poker hand from his two cards together with the five community cards.
- The game is illustrated in the below Figure.



Texas Hold 'Em Poker

- In this game, random numbers are required to shuffle a virtual deck of cards.

- The AFS poker software had a serious flaw in the way that random numbers were used to shuffle the deck of cards.
- As a result, the program did not produce a truly random shuffle, and it was possible for a player to determining the entire deck in real time.
- A player who could take advantage of this flaw could cheat, since he would know all of the other players' hands, as well as the future community cards before they were revealed.

How was it possible to determine the shuffle?

First, there are $52! > 2^{225}$ distinct shuffles of a 52-card deck.

- The AFS poker program used a "random" 32-bit integer to determine the shuffle. Consequently, the program could generate no more than 2^{32} different shuffles out of the more than 2^{225} possible. This was an inexcusable flaw.
- To generate the "random" shuffle, the program used the pseudo-random number generator, or PRNG, built into the Pascal programming language.
- The PRNG was reseeded with each shuffle, with the seed value being a known function of the number of milliseconds since midnight. Since the number of milliseconds in a day is less than 2^{27} distinct shuffles could actually occur.
- Trudy, the attacker, if she synchronized her clock with the server, Trudy could reduce the number of shuffles that needed to be tested to less than 2^{18}.
- These 2^{18} possible shuffles could all be generated in real time and tested against the community cards to determine the actual shuffle for the hand currently in play.
- After the first set of community cards were revealed, Trudy could determine the shuffle uniquely and she would then know the final hands of all other players—even before any of the other players knew their own final hand!

The AFS Texas Hold 'em Poker program is an extreme example of the ill effects of using predictable random numbers where unpredictable random numbers are required. In this example, the number of possible random shuffles was so small that it was possible to determine the shuffle and thereby break the system.

How can we generate cryptographic random numbers?

Since a secure stream cipher keystream is not predictable, the keystream generated by, say, the RC4 cipher must be a good source of cryptographic random numbers.

5.9.2.2 Generating Random Bits

- True randomness is difficult to achieve. The concept of *entropy,* as developed by Claude Shannon explains Entropy is a measure of the uncertainty or, conversely, the predictability of a sequence of bits.
- Sources of true randomness do exist.

For example,

- Radioactive decay is random.
- Hardware devices are available that can be used to gather random bits based on various physical and thermal properties that are known to be unpredictable.
- Another source of randomness is the lava lamp, which achieves its randomness from its chaotic behaviour.
- Since software is deterministic, true random numbers must be generated external to any code.
- Reasonable sources of randomness include mouse movements, keyboard dynamics, certain network activity, and so on.

5.9.3 Information Hiding

The two methods of information hiding, namely,

- Steganography and
- Digital watermarking.

➢ Steganography, or hidden writing, is the attempt to hide the fact that information is being transmitted.

➢ Watermarks generally involve hidden information, but for a slightly different purpose.

For example,

A copyright holder might hide a digital watermark (containing some identifying information) in digital music is a effort to prevent music piracy.

Digital watermarks: Digital watermarks can be categorized in many different ways.

• *Invisible* — Watermarks that are not supposed to be perceptible in the media.

• *Visible* — Watermarks that are meant to be observed, such as a stamp of TOP SECRET on a document.

Watermarks can also be categorized as follows:

• *Robust* — Watermarks that are designed to remain readable even if they are attacked.

• *Fragile* — Watermarks that are supposed to be destroyed or damaged if any tampering occurs.

For example,

1) Insert a robust invisible mark in digital music in the hope of detecting piracy. Then when pirated music appears on the Internet, we can trace it back to its source. Or we might insert a fragile

invisible mark into an audio file. In this case, if the watermark is unreadable, the recipient knows that tampering has occurred. This latter approach is essential an integrity check.

2) Many modern currencies include (non-digital) watermarks. Several current and recent U.S. bills, including the $20 bill pictured in below figure visible watermarks. In this $20 bill, the image of President Jackson is embedded in the paper itself (in the right-hand section of the bill) and is visible when held up to a light. This visible watermark is designed to make counterfeiting more difficult, since special paper is required to duplicate this easily verified watermark.

Watermarked Currency

Example of a simple approach to steganography:

This particular example is applicable to digital images.

Consider images that employ the well-known 24 bits color scheme— one byte each for red, green, and blue, denoted R, G, and B, respectively.

For example,1) the color represented by (R, G, B) = (0x7E, 0x52,0x90) is much different than (R, G,B) = (OxFE, 0x52,0x90), even though the colors only differ by one bit.

On the other hand, the color (R, G, B) = (OxAB, 0x33, OxFO) is indistinguishable from (R, G,B) = (OxAB, 0x33, OxF1), yet these two colors also differ by only a single bit.

The low-order RGB bits are unimportant, since they represent imperceptible changes in color. Since the low-order bits don't matter, it can use them for any purposes we choose, including information hiding.

2) Consider the two images of Alice in the below Figure. The left-most Alice contains no hidden information, whereas the right-most Alice has the entire *Alice in Wonderland* book (in PDF format) embedded in the low-order RGB bits.

- ➢ To the human eye, the two images appear identical at any resolution. If we compare the bits in these two images, the differences would be obvious.
- ➢ In particular, it's easy for an attacker to write a computer program to extract the low-order RGB bits—or to overwrite the bits with garbage and thereby destroy the hidden information, without doing any damage to the image.
- ➢ It is difficult to apply Kerckhoffs' Principle for this example.

A Tale of Two Alices

3) Consider an HTML file that contains the following text, taken from the well-known poem

"To talk of many things
    Of shoes and ships and sealing wax
    Of cabbages and kings
    And why the sea is boiling hot
    And whether pigs have wings."

In HTML, the RGB font colors are specified by a tag of the form <font color="#rrggbb"> ... </font> where rr is the value of R in hexadecimal, gg is G in hex, and bb is B in hex.

For example, the color black is represented by #000000, whereas white is #FFFFFF.
Since the low-order bits of R, G, and B won't affect the perceived color, we can hide information in these bits, as shown in the HTML snippet in the Table . Reading the low-order bits of the RGB colors yields the "hidden" information 011 010 100 100 000 101.

Table : Simple Steganography Example-

```
<font color="#010100">"The time has come,"
the Walrus said,</font><br>
<font color="#000100">"To talk of many things :</fontxbr>
<font color="#010100">0f shoes and ships and sealing wax</font><br>
<font color="#000101">0f cabbages and kings</fontxbr>
<font color="#000000">And why the sea is boiling hot</font><br>
<font color="#010001">And whether pigs have wings."</font><br>
```

- Hiding information in the low-order RGB bits of HTML color tags is obviously not as impressive as hiding *Alice in Wonderland* in Alice's image.

- This method is not at all robust—an attacker who knows the scheme can read the hidden information as easily as the recipient.
- Or an attacker could instead destroy the information by replacing the file with another one that is identical, except that the low-order RGB bits have been randomized.

The conclusion here is that for information hiding to be robust, the information must reside in bits that do matter. But this creates a serious challenge, since any changes to bits that do matter must be done very carefully for the information hiding to remain "invisible."

As noted above, if Trudy knows the information hiding scheme, she can recover the hidden information as easily as the intended recipient. Watermarking schemes therefore generally encrypt the hidden information before embedding it in a file. But even so, if Trudy understands how the scheme works, she can almost certainly damage or destroy the information.

Watermarking schemes often use spread spectrum techniques to better hide the information-carrying bits.