Summary
Information Security

Bjorn Pijnacker

2021–2022 | 1a

Contents

About this summary

This summary is based—in part—on the book Information Security: Principles and Practice, 2nd ed. by Mark Stamp. Chapter 1–10 follow the sectioning of this book. The other chapters (appendix A and B) are based on slides in the course (given 2021–2022 1a) and thus do not follow the book.

Chapter 1

Introduction

1.1 The Cast of Characters

Tradition in the field of information security is to use Alice and Bob as our "good guys", and Charlie and Dave if we require more. Our generic "bad guy" is Trudy. These characters do not need to be human per se: they can also represent computer systems or other actors.

1.2 Alice's Online Bank

Suppose Alice starts a bank named Alice's Online Bank (AOB). What are the security concerns from Alice's point of view, and what are those from Bob's (a customer) point of view?

1.2.1 Confidentiality, Integrity, and Availability

Confidentiality deals with preventing unauthorized reading of information. In the AOB, Alice might not care about this, but Bob certainly does, as Bob does not want his information to be public knowledge.

Integrity deals with preventing, or at least detecting, unauthorized modifying of data. If Alice's bank does not protect the integrity of data, then Trudy may be able to change Bob's account balance.

Integrity and confidentiality are strictly not the same. In some cases, Trudy might not be able to read data but she may be able to write/change data undetected.

Availability is also a fundamental issue. Denial of service, or DoS, attacks—for example—try to reduce availability of information. This is an issue both for AOB and Bob: if AOB's website is unavailable then Alice can't make money and Bob can't get to his money.

1.2.2 Beyond CIA

The above three concerns (CIA) are only the first step in the story of information security. We will also consider things like authentication and authorization.

1.4 The People Problem

The biggest problem with regards to information security are people. For example, a browser may give a warning when an SSL protocol doesn't yield a response that indicates security, Bob may simply choose to ignore this warning and as such do unsafe things.

Another example is passwords, if passwords could be of arbitrary length and completely random, they would be much more secure than what users come up with. Especially since users commonly use the same password for all services they use.

Part I

Crypto

Chapter 2

Crypto Basics

2.1 Introduction

In this chapter we discuss some of the basic elements of cryptography. The discussion lays the foundation for the remaining crypto chapters.

2.2 How to Speak Crypto

A *cipher* or *cryptosystem* is used to encrypt data. The original unencrypted text is known as *plaintext* and the result of the encryption is the *ciphertext*. In a symmetric cipher, the same key is used to encrypt and decrypt the data. In an assymetric cipher, encryption and decryption use different keys.

For an ideal cipher it should be infeasible (not impossible!) to recover the plaintext from the ciphertext. A fundamental tenet in this is that the inner workings of the cryptosystem are completely known to the attacker. Since we can assume that finding out these details is a lot easier than breaking a proper secure cryptosystem we vow to not use security by obscurity. This is known as *Kerckhoff's Principle*.

2.3 Classic Crypto

2.3.1 Simple Substitution Cipher

In the simple substitution cipher we have a plaintext p of arbitrary length. We also have a key n. We shift every character in p by n places w.r.t. the alphabet. To decrypt this (using the same key) we do the substitution in reverse, that is, use the simple substitution cipher with the key n multiplied by -1.

An issue with this cipher is that of the exhaustive key search. Since using the alphabet we only have 26 keys, we may expect Trudy to find the correct key in 13 tries. It's not hard to see

whether a key is correct simply by checking if the resulting plaintext after decryption makes sense.

We can enlarge our keyspace by, instead of shifting, using any permutation of the alphabet as a substitution key. A substitution like this will result in 26! $\approx 2^{88}$ keys, which is a lot more than 26.

2.3.2 Cryptanalysis of a Simple Substitution

If Trudy intercepts a ciphertext that was encrypted using the simple substitution cipher it is still easy to recover the plaintext without trying 2^{87} (on average) keys. If Trudy knows the language of the plaintext she can look at the frequencies of the letters in the ciphertext and in general in the language. She can use the information to see which letter any letter was pre-substitution. This requires far less work than an exhaustive key search, and shows that simple substitution is not very secure at all.

2.3.3 Definition of Secure

Above we name that a simple substitution is not secure, but how do we define secure? We could require that the best known attack on a system is impractical, in the sense that it is computationally infeasible. How we define *secure* is as follows:

> We say that a cryptosystem is *secure* if the best-known attack requires as much work as an exhaustive key search. In other words, no shortcut attack is known.

2.3.4 Double Transposition Cipher

To encrypt with the double transposition cipher, we first write the plaintext into an array of given size and then permute the rows and columns according to specific permutations. For example, for the plaintext `attackatdawn`:

$$\begin{bmatrix} \texttt{a} & \texttt{t} & \texttt{t} & \texttt{a} \\ \texttt{c} & \texttt{k} & \texttt{a} & \texttt{t} \\ \texttt{d} & \texttt{a} & \texttt{w} & \texttt{n} \end{bmatrix}$$

If we transpose the rows according to $(1,2,3) \to (3,2,1)$ and the columns according to $(1,2,3,4) \to (4,2,1,3)$, we get

$$\begin{bmatrix} \texttt{a} & \texttt{t} & \texttt{t} & \texttt{a} \\ \texttt{c} & \texttt{k} & \texttt{a} & \texttt{t} \\ \texttt{d} & \texttt{a} & \texttt{w} & \texttt{n} \end{bmatrix} \to \begin{bmatrix} \texttt{d} & \texttt{a} & \texttt{w} & \texttt{n} \\ \texttt{c} & \texttt{k} & \texttt{a} & \texttt{t} \\ \texttt{a} & \texttt{t} & \texttt{t} & \texttt{a} \end{bmatrix} \to \begin{bmatrix} \texttt{n} & \texttt{a} & \texttt{d} & \texttt{w} \\ \texttt{t} & \texttt{k} & \texttt{c} & \texttt{a} \\ \texttt{a} & \texttt{t} & \texttt{a} & \texttt{t} \end{bmatrix}$$

giving us the ciphertext `nadwtkcaatat`.

In the double transposition cipher, the key consists of the size of the matrix and the row and column permutations. Given these the above process can be easily reversed.

The double transposition cipher does not do anything to disguise the letters that appear in the message, though a statistical attack as we saw before is not possible here since this doesn't tell us anything about the order.

2.3.5 One-Time Pad

The one-time pad, a.k.a. Vernam cipher is provably secure. This is the only provably secure cipher we will see. The first step in this cipher is to map our alphabet to some binary representation like ASCII. The one-time pad key consists of a randomly selected sting of bits that is the same length as the message. This key is then XORed with the plaintext to give the ciphertext.

The best property that this gives us is that the the ciphertext can be made into any other message by decrypting it with the wrong key. If this key is carefully chosen (or guessed on accident) a different message appears with the attacker none the wiser.

In the one-time pad, the key has to be actually random (not pseudorandom as most computers do it), used only once, and known only to Alice and Bob. This key must be used only once since it's possible to obtain the key using two messages encrypted with the same key.

Chapter 3

Symmetric Key Crypto

3.1 Introduction

The two main branches of symmetric key crypto are stream ciphers and block ciphers, both of which we discuss here. Stream ciphers generalize the idea of a one-time pad, except we forego provable security in favor of a shorter key. Block ciphers can be seen as the modern successor to classic codebook ciphers, where the key determines the codebook.

3.2 Stream Ciphers

A stream cipher takes a key K of length n bits and stretches it into a *keystream*. This is XORed with our plaintext P resulting in the ciphertext C. Since XOR is used for encryption, decryption can also be done using the same keystream. We find that C is the bitwise XOR of P with the keystream and P is the bitwise XOR of C with the keystream.

3.2.1 A5/1

A5/1 is a stream cipher used in GSM confidentiality. It employs three linear feedback shift registers X, Y, and Z. Register X holds 19 bits, Y holds 22, and Z holds 23; giving a total of 64 bits. The key K for A5/1 is also 64 bits in length. This key is used to fill the initial three registers.

We can *step* our registers according to the following operations. For X a step consists of:

$$\begin{aligned} t &= x_{13} \oplus x_{16} \oplus x_{17} \oplus x_{18} \\ x_i &= x_{i-1} \text{ for } i = 18, 17, 16, \ldots, 1 \\ x_0 &= t \end{aligned}$$

For Y, a step consists of:

$$\begin{aligned} t &= y_{20} \oplus y_{21} \\ y_i &= y_{i-1} \text{ for } i = 21, 20, 19, \ldots, 1 \\ y_0 &= t \end{aligned}$$

And finally for Z:

$$\begin{aligned} t &= z_7 \oplus z_{20} \oplus z_{21} \oplus z_{22} \\ z_i &= z_{i-1} \text{ for } i = 22, 21, 20, \ldots, 1 \\ z_0 &= t \end{aligned}$$

Given three bits x, y, z we define $\text{maj}(x, y, z)$ to be the majority vote function, that is, if the majority of x, y, z are 0 then the function returns 0; otherwise it returns 1. In A5/1, for each keystream bit we generate the following takes place: We compute $m = \text{maj}(x_8, y_{10}, z_{10})$. If $x_8 = m$, X steps; if $y_{10} = m$, Y steps; if $z_{10} = m$, Z steps. We then generate a single keystream bit s as $s = x_{18} \oplus y_{21} \oplus z_{22}$.

3.2.2 RC4

While A5/1 is designed for hardware implementation, RC4 is designed for software implementation. RC4, in contrast to A5/1, computes a keystream *byte* per step; not a bit.

RC4 consists of two phases: The first is to initialize the lookup table it uses using the key. The pseudocode for this is given in the book. After the initialization phase, each keystream byte is generated by looking at the table. After this, the table is changed.

An attack is possible against RC4. Luckily this attack is infeasible if we discard the first 256 keystream bytes that are generated. After the table initialization, we simple generate 256 keystream bytes that we then don't use.

3.3 Block Ciphers

Block ciphers are the counterparts to stream ciphers. Block ciphers split the plaintext into fixed-size blocks and generate fixed-size blocks of ciphertext from these. In most block ciphers, ciphertext is obtained from the plaintext by iterating a function F over some number of *rounds*. The goal of block ciphers is to be secure and efficient.

3.3.1 Feistel Cipher

In a Feistel cipher the plaintext P is split up in two halves L_0 (left) and R_0 (right). For each round $i \in [1, n]$ new left and right halves are computed by making the new left half equal to the previous right half, and by making the new right half equal to the old left half, XORed with

$F(R_{i-1}, K_i)$, where K_i is the *subkey* for round i. This subkey is derived using a key schedule algorithm. The resulting ciphertext is the output of the final found: $C = (L_n, R_n)$.

Using this process we can also do decryption simply by reversing the next-round steps. We then finally obtain L_0 and R_0 from L_n and R_n and we have our plaintext once more.

Any function F will work in a Feistel cipher provided it gives the correct number of bits. How secure the Feistel cipher is is mostly down to the choice of this function F. Interestingly however is that F need not be invertible.

3.3.2 DES

DES is a Feistel cipher with the following properties: It has 16 rounds, a 64-bit block length, a 56-bit key, and 48-bit subkeys. Each round is relatively simple. One of DES' most important security features are its S-boxes. In DES, each S-box maps 6 bits to 4 bits. Since DES uses eight of these it essentially maps 48 bits to 32 bits. These S-boxes are implemented as a lookup table.

The function F for DES can be written as

$$F(R_{i-1}, K_i) = \text{P-box}(\text{S-boxes}(\text{Expand}(R_{i-1}) \oplus K_i))$$

In DES, the expand and compress permutations change the input size from 32 to 48 and from 56 to 48 bits, respectively. While S-boxes also compress 48 to 32 bits, P-boxes have both 32-bit input and output. What these exact permutations are should be looked up in the book.

The key schedule algorithm in DES essentially selects 48 of the 56 bits to use as a key each round. First, 28 bits of the DES key are extracted then permuted. We call this LK. We do the same with the remaining 28 bits and name it RK. We then define r_i equal to 1 if $i \in \{1, 2, 9, 16\}$ and 2 otherwise. We have the key schedule algorithm as in Algorithm 1. In this algorithm, LP and RP are permutations as defined in the book.

Algorithm 1: DES Key Schedule Algorithm

for *each round* $i = 1, 2, \ldots, n$ **do**
- LK = cyclically left shift LK by r_i bits
- RK = cyclically left shift RK by r_i bits
- The left half of subkey K_i consists of bits LP of LK
- The right half of subkey K_i consists of bits RP of RK

3.3.3 Triple DES

Triple DES, or 3DES, is a popular variant of DES. In DES, we have $C = E(P, K)$ and $P = D(C, K)$ where E and D are encrypt and decrypt, respectively. Triple DES is defined as $C = E(D(E(P, K_1), K_2), K_1)$ which doubles the keyspace for DES. We use EDE instead of EEE for backwards compatibility. If 3DES is used with $K_1 = K_2 = K$ it collapses to $E(P, K)$ since the inner encrypt and decrypt cancel each other out when using the same key.

3.3.4 AES

AES is—like DES—an iterated block cipher. It is, however, not a Feistel cipher. This means that in order to decrypt, AES operations must be invertible. AES is also highly mathematical in its structure. Some properties of AES are: a 128-bit block size; keys of 128, 192, or 256 bits; 10–14 rounds depending on key size; four functions in three layers per round of the cipher.

These four functions are named `ByteSub`, `ShiftRow`, `MixColumn`, `AddRoundKey`. AES treats the 128-bit block as a 4×4 byte matrix on which the four functions act. All four of these functions are invertible and thus AES can encrypt and decrypt using these.

`ByteSub` is roughly equivalent to the DES S-boxes. It's a non-linear yet invertible composition of two mathematical functions, but can be viewed as a lookup table. This lookup table is shown in the book.

`ShiftRow` is a cyclic shift of the bytes where each row is shifted by its index, that is, row 0 is not shifted, row 1 is shifted one position, row 2 is shifted two positions, etc.

`MixColumn` consists of shift and XOR operations applied to each column of the matrix. It is most efficiently implemented as a lookup table and serves a similar purpose to the DES S-boxes.

`AddRoundKey` applies the subkey to the matrix. This subkey is generated using a key schedule algorithm. Let k_{ij} be the 4×4 subkey matrix. Then the subkey is XORed with the current 4×4 byte matrix a_{ij} resulting in b_{ij}.

3.3.5 Three More Block Ciphers

In this section we consider International Data Encryption Algorithm (IDEA), Blowfish, and RC6; in particular we look at their interesting features.

IDEA uses mixed mode arithmetic. It combines addition modulo two (XOR) with addition module 2^{16} and Lai-Massey multiplication. These operations provide the necessary nonlinearity and as such we don't need an explicit S-box.

Blowfish uses key-dependent S-boxes instead of having fixed S-boxes. These S-boxes are based on the key.

RC6 has the unusual aspect that it uses data-dependent rotations. It's highly unusual to rely on the data as an essential part of the operation of a crypto algorithm since this makes its ciphertext properties dependent on the plaintext.

3.3.7 Block Cipher Modes

If we have multiple plaintext blocks $P_0, P_1, P_2, \ldots$ and a key K, we can encrypt each block according to $C_i = E(P_i, K)$. This is called ECB mode. This approach works, but has serious security issues. In this mode, if a hacker observes that $C_i = C_j$ they can infer that $P_i = P_j$. There are more of these issues.

A better mode to use is CBC, or cipher block chaining, mode. The ciphertext from a block is then used to obscure the plaintext of the next block before it is encrypted, given by

$$C_i = E(P_i \oplus C_{i-1}, K)$$

In this case our first step requires an initialization vector. It should be randomly selected, but need not be secret.

When using CBC mode, we are still susceptible to a cut-and-paste attack (see book). To solve this we can employ CTR mode, where we encrypt according to

$$C_i = P_i \oplus E(IV + i, K)$$

where IV is the initialization vector.

3.4 Integrity

Whereas confidentiality deals with preventing unauthorized reading, integrity deals with unauthorized writing. We will show here that block ciphers also provide data integrity.

A message authentication code, or MAC, uses a block cipher to ensure data integrity. We simply encrypt the data in CBC mode and then discard all ciphertext blocks except the final once. Since in CBC mode we use the data from the previous block in encrypting the current block, the final block will house information of all previous blocks. If any of these blocks changes, so does the final ciphertext block. This serves as our MAC. Since only Alice and Bob have the key that is used to compute the MAC, Trudy will not be able to change the plaintext and the MAC to match.

Chapter 4

Public Key Crypto

4.1 Introduction

Public key crypto, a.k.a. assymetric crypto, uses one key for encryption and another for decryption. A result of this is that the encryption key can be made public.

A public key cryptosystem is based on a trap door one-way function. The function is easy to compute one way but computing it the other was is computationally infeasible. The "trap door" feature ensures that an attacker cannot use the public information to infer private information.

4.2 Knapsack

The Merkle-Hellman knapsack cryptosystem is based on the knapsack problem, which we define as follows:

> **The Knapsack Problem** Given a set of n weights labeled as $W_0, W_1, \ldots, W_{n-1}$ and a desired sum S, find $a_0, a_1, \ldots, a_{n-1}$ where each $a_i \in \{0, 1\}$ such that $S = \sum_{i=0}^{n-1} a_i W_i$ provided this is possible.
>
> While the general knapsack problem is NP-complete, the superincreasing knapsack has the weights arranged from least to greater, with each weight greater than the sum of all the previous weights. This problem can be solved in linear time.
>
> We can convert a superincreasing knapsack into a general knapsack by choosing a multiplier m and a modulus n such that m and n are relatively prime and n is greater than the sum of all elements in the superincreasing knapsack. The general knapsack is then computed from the superincreasing knapsack by modular multiplication.

In the process of constructing a knapsack cryptosystem we have four steps: (1) generate a superincreasing knapsack; (2) converting the superincreasing knapsack into a general knapsack; (3) the public key is the general knapsack; (4) the private key is the superincreasing knapsack together with the conversion factors.

To encrypt using the general knapsack, Alice uses the 1 bits in her message to select the elements of the general knapsack that are summed to give the ciphertext. Bob uses the private key to find the superincreasing knapsack sum that corresponds using

$$S = C \cdot m^{-1} \bmod n$$

and solves the superincreasing knapsack for this sum.

Sadly, the general knapsack that we construct above isn't actually a general knapsack. It is a highly structured case of the knapsack, leaving it open to an attack that allows recovery of the plaintext with high probability.

4.3 RSA

To generate an RSA public/private key pair we choose to large primes p and q and define $N = pq$. We then choose e relatively prime to $(p-1)(q-1)$ and find the multiplicative inverse of $e \bmod (p-1)(q-1) = d$. We can now forget the factors p and q. We have our modulus N, our encryption exponent e, and our decryption exponent d. The RSA key pair is then (N, e) for our public key and d for our private key.

Encryption and decryption are done via modular exponentiation according to:

$$C = M^e \bmod N \qquad M = C^d \bmod N$$

RSA can be broken by factoring, which is a hard problem for large primes. Given N it is possible to recover p and q and as such recover d from the public information. Since factoring for large primes is a very hard problem, this is not a concern if p and q are sufficiently large.

4.3.3 Speeding Up RSA

A trick that can be used to speed up RSA is to use the same e for all users. This does not weaken RSA in any way, as far as is known. The decryption components can still be different, since different p and q are chosen for each key pair. A suitable and widely used choice for this is $e = 3$. This still has expensive decryption operations, but that is acceptable. When using $e = 3$ special care needs to be taken to avoid attacks: To avoid the cube root attack, M must be padded to have enough bits such that, as a number, $M > N^{1/3}$. Similarly when many users use different keys yet the same message, M must be padded with some user-specific information, to avoid using the Chinese Remainder Theorem to reconstruct M.

Another popular choice is $e = 2^{16} + 1$. Having a larger e protects against the Chinese Remainder Theorem attack.

4.4 Diffie-Hellman

The Diffie-Hellman key exchange algorithm is an algorithm that allows for securely establishing a shared symmetric key. It is not used for encrypting or decrypting. The security of DH relies on the computational difficulty of the discrete log problem.

The exchange works as follows: Let p be prime and g be a generator, which means that for any $x \in \{1, 2, \ldots, p-1\}$ there is an exponent n such that $x = g^n \bmod p$. Both g and p are public. Alice randomly selects a secret exponent a and Bob selects a secret exponent b. Alice computes $g^a \bmod p$ and sends this to Bob. Bob computes $g^b \bmod p$ and sends this to Alice. Alice then computes $(g^b)^a \bmod p = g^{ab} \bmod p$ and Bob computes $(g^a)^b \bmod p = g^{ab} \bmod p$. This is then used as the shared secret key, wince both Alice and Bob have it. Without knowing either a or b and the respective log from the counterpart, it is infeasible to determine $g^{ab} \bmod p$ as it requires solving the discrete log problem.

DH is susceptible to a Man-in-the-Middle attack. Possible ways to prevent this is to encrypt the DH exchange with a shared symmetric key, to encrypt the DH exchange with public keys, or to sign the DH values with private keys.

4.5 Elliptic Curve Cryptography

Elliptic curves provide a different domain for performing the complex mathematical operations required for public key cryptography. For example, we will consider an elliptic curve version of Diffie-Hellman.

The advantage of ECC is that we need fewer bits for the same level of security. On the downside, ECC math is more complex and as such each mathematical operation is somewhat more expensive.

4.5.1 Elliptic Curve Math

An elliptic curve E is the graph of a function of the form

$$E : y^2 = x^3 + ax + b$$

together with a special point at infinity, denoted ∞. If we have some points on this curve we can do math on these. For example the addition of P_1 and P_2 on E. First, a line is drawn through the points. This line usually intersects the curve at some other point. This intersection point is reflected about the x axis to obtain P_3 on E which is defined to be $P_1 + P_2$. In cryptography we usually wish to deal with a discrete set of points, so we define our curve to be of the form

$$E' : y^2 = x^3 + ax + b \pmod{p}$$

Using this definition of E' we can create an algebraic algorithm for adding two points on the curve. This algorithm is shown in the book and won't be shown here.

4.5.2 ECC Diffie-Hellman

In ECC DH, the public information consists of a curve and a point on this curve. We select any point (x, y) and determine b in the curve such that this point lies on the resulting curve. This yields our public information.

Alice and Bob both randomly select their own secret multipliers. Alice then computes the given point multiplied by her secret multiplier.[1] Alice sends her computed result to Bob who also computes the given point multiplied by his secret multiples, which he sends to Alice. Alice and Bob then both multiply the received value with their own secret identifier, yielding a shared secret.

4.6 Public Key Notation

We adopt the following notation for public key encryption, decryption, and signing:

- Encrypt message M with Alice's public key: $C = \{M\}_{\text{Alice}}$.
- Decrypt ciphertext C with Alice's private key: $M = [C]_{\text{Alice}}$.
- Signing message M with Alice's private key: $S = [M]_{\text{Alice}}$.

Curly brackets represent public key operations and square brackets represent private key operations. Either is subscripted telling us whose key is being used. Public and private operations cancel each other out:

$$[\{M\}_{\text{Alice}}]_{\text{Alice}} = \{[M]_{\text{Alice}}\}_{\text{Alice}} = M$$

4.7 Uses for Public Key Crypto

4.7.1 Confidentiality in the Real World

Public key crypto is less efficient than symmetric key crypto, but it's easier to use since there is no shared key required. We can get the best of both worlds where we use public key crypto to establish a symmetric key that is then used to encrypt the data. This is called a *hybrid cryptosystem*. Hybrid crypto (with secure authentication!) is widely used in practice today. Authentication is important here, since Bob must know he is talking to Alice, and not Trudy pretending to be Alice.

4.7.2 Signatures and Non-repudiation

A signature can be used for integrity. It is the public key crypto counterpart of a MAC. A digital signature also provides non-repudiation, which symmetric keys—by nature—cannot.

[1] Multiplication in ECC is defined as repeated addition

When a message is signed by Alice and then sent to Bob, Bob can prove that is was Alice that sent the message, even if Alice later says she did not, as we can assume that only Alice has access to her own private key.

4.7.3 Confidentiality and Non-repudiation

If Alice wants to send a confidential message to Bob she can encrypt it with Bob's public key, and for integrity and non-repudiation, she signs it with her own private key. The solution seems straight forward, that is, Alice can send $\{[M]_{\text{Alice}}\}_{\text{Bob}}$ to Bob. This is, however, wrong. If Bob decrypts the message wit his private key, what remains is $[M]_{\text{Alice}}$. He can then encrypt this with Charlie's public key to obtain $\{[M]_{\text{Alice}}\}_{\text{Charlie}}$ fooling Charlie into thinking that M was meant for him and was sent by Alice.

If we reverse the process, and obtain $[\{M\}_{\text{Bob}}]_{\text{Alice}}$ we are now susceptible to a Man-in-the-Middle attack, where Charlie can strip Alice's signature and resign it himself, making Bob think that the message came from Charlie.

4.8 Public Key Infrastructure

A PKI is everything that is required to securely use public keys in the real world. A digital certificate M contains a user's name along with their public key. This certificate is signed by a certificate authority, or CA, resulting in S. Bob can verify this certificate by computing $S = [M]_{\text{CA}}$ and checking that it matches M. In this case, the CA is a trusted third party.

A PKI must deal with three issues: key generation and management, CAs, and certificate revocation. There are many PKI trust models in use today of which we discuss a few:

The most obvious is the monopoly model, where one universally trusted organization is the CA for everyone. A drawback of this model is that it creates one big target for attack. The oligarchy model has multiple trusted CAs. A user is then free to decide which CA they trust and which they do not. A more typical user might just trust the default on their application. The basically opposite model is the anarchy mode, where anyone can be a CA and it's up to users to decide whom to trust. This approach is used in PGP by the name "web of trust".

Chapter 5

Hash Functions++

5.1 Introduction

Cryptographic hash functions are used for, among others, digital signatures and hashed MACs and other uses, like online bidding and spam reduction.

5.2 What is a Cryptographic Hash Function?

Hashing—in computing science—is a widely used term with different meanings in different fields. In cryptography we define a cryptographic hash function $h(x)$ to have all of the following properties:

- Compression — The output $y = h(x)$ is small regardless of the size of x;
- Efficiency — $h(x)$ must be easy to compute for any input x;
- One-way — Given any value $y = h(x)$ it is infeasible to find x such that $h(x) = y$;
- Weak collision resistance — Given x and $h(x)$ it's infeasible to find any $y \neq x$ such that $h(x) = h(y)$;
- Strong collision resistance — It's infeasible to find any x, y where $x \neq y$ such that $h(x) = h(y)$.

Previously we considered signing a message to find $S = [M]_{\text{Alice}}$. If the message is large, then S takes long to compute. If we instead first hash M to obtain $h(M)$ we can sign this instead. We send both M and $S = [h(M)]_{\text{Alice}}$ to Bob and achieve the same result. Bob can verify the signed hash came from Alice, and he can check that M wasn't changed in transit.

Assuming no collisions, signing $h(M)$ is as good as signing M. It is important to realize that now the security of the signature depends on both the security of the public key system *and* the hash function.

5.3 The Birthday Problem

There are N people in a room. How large must N be before we expect two or more people will have the same birthday? It's easier to solve for the probability of the complement and subtract from one. Solving this, we find $N = 23$, even though there is 365 possible birthdays. This is way less than we would intuitively expect.

In cryptography this is an important problem. Suppose we have a hash function $h(X)$ that gives an N-bit long output. This has 2^N possible values. The birthday problem immediately implies that if we hash about $2^{N/2}$ different inputs we can expect to find a collision. In contrast, a secure symmetric key cipher requires a factor of 2^{N-1} of work.

5.4 A Birthday Attack

Suppose that a hash function h generates an n-bit output. Trudy can now perform a birthday attack as follows: (1) Trudy selects an "evil" message E that she wants Alice to sign but which Alice is unwilling to sign. Trudy also creates an innocent message I that she is confident Alice is willing to sign. (2) Trudy generates $2^{n/2}$ variants of the innocent message by making minor editorial changes. We denote these as I_i for $i \in [0, 2^{n/2})$. Trudy also creates $2^{n/2}$ variants of the evil message, denoted E_i. (3) Trudy hashes all evil messages and innocent messages. By the birthday problem she can expect to find a collision $H(E_j) = H(I_k)$. Trudy then sends I_k to Alice having her sign it, and receives I_k and $[h(I_k)]_{\text{Alice}}$. Since the hashes are the same, it follows that $[h(E_j)]_{\text{Alice}} = [h(I_k)]_{\text{Alice}}$ and so, in effect, Trudy now has Alice's signature on E_j.

5.5 Non-Cryptographic Hashes

Consider the data $X = (X_0, X_1, X_2, \ldots, X_{n-1})$ where each X_i is a byte. A simple hash function $h(X)$ is given by

$$h(X) = \left(\sum_{i=0}^{n-1} X_i \right) \bmod 256$$

While this hash certainly provides compression, it would be easy to break: we expect to find a collision by hashing just 2^4 random inputs. It's even easy to construct a collision directly.

Another hash function we can look at is

$$h(X) = \left(\sum_{i=0}^{n-1} (n-i) X_i \right) \bmod 256$$

but sadly this is also insecure, since we can still easily find collisions, either by the birthday problem or by directly constructing them. While this hash is not cryptographically secure, it is still used in the application `rsync`, where it is not required to be cryptographically secure.

Another example of a non-cryptographic hash is the cyclic redundancy check (CRC). It is sometimes mistakenly used as a cryptographic hash. It is essentially long division with the

remainder acting as the hash value; only in place of actual division we use the XOR operation. In CRC the divisor is specified as part of the algorithm and the data acts as the dividend. It's easy to find collisions simply by doing the CRC calculation for some data, then working backwards.

See slides for example of CRC collision generation.

5.6 Tiger Hash

The input into Tiger is divided into 512-bit blocks with the input padded to a multiple of 512 bits, if necessary. Tiger hashes these block-by-block, not unlike a block cipher. The output of Tiger is 192 bits. The input X is written as $X = (X_0, X_1, \ldots, X_{n-1})$. The Tiger algorithm employs one *outer round* for each X_i. It also uses values a, b, c which are defined for the first round:

$$\begin{aligned} a &= \texttt{0x0123456789ABCDEF} \\ b &= \texttt{0xFEDCBA9876543210} \\ c &= \texttt{0xF096A5B4C3B2E187} \end{aligned}$$

The final (a, b, c) output from one round is the initial a, b, c for the next round.

Within the outer round, tiger calls the function F three times. This function F_m consists of eight *inner rounds*. The 512-bit input to F is denoted as $W = (w_0, w_1, \ldots, w_7)$. Each inner round is represented as $f_{m,i}$. This function is also passed the values a, b, c. We write $c = (c_0, c_1, \ldots, c_7)$. We then define $f_{m,i}$ as

$$\begin{aligned} c &= c \oplus w_i \\ a &= a - (S_0[c_0] \oplus S_1[c_2] \oplus S_2[c_4] \oplus S_3[c_6]) \\ b &= b + (S_3[c_1] \oplus S_2[c_3] \oplus S_1[c_5] \oplus S_0[c_7]) \\ b &= b \cdot m \end{aligned}$$

where each S_i is an S-box mapping 8 bits to 64 bits.

The value of W here is provided by the key-schedule algorithm. The exact details of this can be found in the book. Most important to know is that the initial value of W is X_i for each outer round, which is then ran through the key schedule algorithm between each F_m, of which there is three. This (together with the S-boxes) ensures every bit of X has influence on the final a, b, c values.

At the end of each outer round we do $a = a \oplus a'; b = b - b'; c = c + c'$ where a', b', c' are the values of a, b, c at the start of this outer round.

For a block-diagram representation of Tiger hash, please see the book

5.7 HMAC

HMAC is the hashing-counterpart to MAC. Instead of the trivial solution to ensure integrity, where Alice encrypts the hash before sending it to Bob, we can use a hashed MAC, or HMAC. We mix the key directly into M when computing the hash. We can either append or prepend the key to the message. Both methods have subtle attacks.

Instead, we mix the key into the message in a more sophisticated manner, to protect against these attacks. Let B be the block length of hash in bytes. We define "ipad" equal to `0x36` repeated B times and "opad" to `0x5C` repeated B times. We then define the HMAC of M as

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

This approach mixes the key into the resulting hash and protects against writing in the manner that any unauthorized writing to the message can be detected.

Part II

Access Control

Chapter 7

Authentication

7.1 Introduction

Authentication is the process of determining whether a user should be allowed to access a system. We focus on authenticating humans to machines. Authentication is not authorization, since authorization deals with the question whether someone is allowed to do something, and authentication deals with the question whether someone says who they are.

7.2 Authentication Methods

A human can be authenticated to a machine using any of three "somethings": (1) something you know; (2) something you have; (3) something you are. A password is an example of "something you know". An example of "something you have" is an ATM card or a TOTP code. An example of "something you are" is any form of biometrics.

7.3 Passwords

Users usually select bad passwords, making password cracking surprisingly easy. As such, achieving security via passwords is very difficult.

7.3.1 Keys Versus Passwords

Cryptographic keys would easily solve the problem of passwords since passwords are not random. For an example and a mathematical explanation of this fact, please see the book.

7.3.2 Choosing Passwords

A password that is hard to guess and hard to remember is secure, but will end up on a Post-It note on a computer thus being insecure. A password that's personal to someone but changed

a bit is still easy to guess. An example is the password `POkemON` for a fan of Pokémon. A passphrase is a good middle-ground. Easy to remember but hard to guess.

7.3.3 Attacking Systems via Passwords

A common path for an attacker that has no access to a system is to go from an outsider to a normal user to an administrator. This means that one weak password on a system might be enough for the first step of the attack.

An interesting issue that concerns this is what to do when password cracking is detected. Locking a user for just a few seconds might not evade the attack, especially since Trudy can try different users too. The correct answer to this dilemma is not really clear.

7.3.4 Password Verification

Instead of storing raw passwords in a file or encrypting the passwords (which is reversible), it is more secure to store hashed passwords in a file, where we hash with a cryptographic hash function. When trying to log in the candidate password is also hashed and compared to the stored hash. If they're equal, we authenticate the user.

A downside of the hashing approach is that if Trudy manages to find a password and its corresponding hash, she immediately knows the original passwords of all the same hashes in the password file. A solution to this is hashing: Let p be a newly generated password. We generate a random salt value s and compute $y = h(p, s)$ and store the pair (y, s) in the password file. The salt isn't secret, but it does make the hash value different to other hashes of the same password and a different (or no) salt. This means Trudy has a factor N more work to do to crack all the passwords, where N is the length of the password file.

7.3.5 Math of Password Cracking

Throughout this section we assume that all passwords are eight characters in length and that there are 128 choices for each character, which implies we have $128^8 = 2^{56}$ possible passwords. We assume that a password file contains 2^{10} passwords and that Trudy has a dictionary of 2^{20} common passwords. We consider four cases:

Case I — Trudy wants to determine Alice's password. Trudy does not use her dictionary

The only approach for Trudy here is a brute force approach. This is exactly similar to an exhaustive key search, and thus the expected work is

$$2^{56}/2 = 2^{55}$$

Case II — Trudy wants to determine Alice's password. Trudy uses her dictionary

The dictionary has probability $1/4$ of containing Alice's password. Suppose the passwords are salted and the dictionary contains Alice's password, then Trudy expects to find Alice's

password after hashing half of the words in the dictionary, that is, after 2^{19} tries. Since this has a probability of $1/4$ we add to this the result of the first case with $3/4$ probability. The expected work for Trudy is now

$$1/4\left(2^{19}\right) + 3/4\left(2^{55}\right) \approx 2^{54.6}$$

If the passwords are unsalted Trudy could precompute the hashes of all 2^{20} passwords the hashes of all the password in her dictionary. This reduces the work per attack since hashing will only need to happen once for the dictionary, not once per attack.

Case III — Trudy will be satisfied to crack any password in the password file, without using her dictionary

If we let $y_0, y_1, \ldots, y_{1023}$ be our password hashes. We assume all passwords in the file our distinct. Let $p_0, p_1, \ldots p_{2^56-1}$ be a list of all the *possible* passwords. In the brute force case, Trudy needs to make 2^{55} comparisons before she expects to find a match.

If the passwords are unsalted then Trudy can compute $h(p_0)$ and compare with y_i for $i \in [0, 1023]$. Then continue the same for all other p_j. We measure work in hashes and not comparisons, so we have expected work equal to

$$2^{55}/2^{10} = 2^{45}$$

If the passwords are salted the above y_i for all i comparison doesn't work with just calculating $h(p_0)$ once. The hash of p_0 will need to be calculated for each y_i and as such we multiply the expected work by 2^{10} resulting in expected work equal to 2^{55}, which is the same as case I.

Case IV — Trudy wants to find any password in the hasped password file, using her dictionary

First, note that the probability of any of the 1024 passwords being in Trudy's dictionary is

$$1 - (3/4)^{1024} \approx 1$$

so we can ignore any case where no password is in the dictionary. In reality, if no password is in the dictionary an attacker would just give up anyway.

If the passwords are not salted then Trudy can simply try each password in the password file by hashing them and comparing them to each stored hash. Since we expect to only need half of the comparisons before we expect to find a password, so our expected work is

$$2^{19}/2^{10} = 2^9$$

If the passwords are salted we can approximate the amount of work to 2^{22}. This is the most realistic scenario. In general we can approximate the expected work by dividing the size of the dictionary by the probability that a password in the password file is in the dictionary.

7.3.6 Other Password Issues

Nowadays, users have a ton of passwords. Users can't remember a large number of passwords, and so passwords are reused significantly. If Trudy manages to find one of your passwords she will definitely try the same password (or slight variations) in different places.

Another concern is social engineering. If someone calls you claiming to be a system administrator saying they need your password, 34% of people are willing to give away their password. This increases to 70% if you offer them a candy bar.

Other issues to consider are keystroke logging software and similar spyware, but also password cracking tools that can automate the work we described for Trudy, above.

7.4 Biometrics

Biometrics is the "something you are" method of authentication. In the information security field, biometrics are seen as a more secure alternative to passwords. To be a suitable replacement however, biometrics need to be cheap and reliable. An ideal biometric satisfies all of the following:

- Universal — A biometric should apply to virtually everyone;
- Distinguishing — A biometric should distinguish with good certainty. A certainty for 100% is not realistically possible, but we want to get as close as possible;
- Permanent — A characteristic used for biometrics shouldn't change. It's realistically sufficient if it changes very little over time;
- Collectable — The characteristic should be easy to collect without any potential harm to the subject;
- Reliable, robust, and user-friendly — These are some real-world considerations for a practical system.

In a biometric system we have two phases: the enrollment phase, and the recognition phase. In the enrollment phase a subject has their biometric information gathered and entered into a database. Very careful measurement is required, and since it's one-time work, it doesn't matter if the process is slow. The recognition phase is used when the system is determining whether to authenticate a user. This phase must be quick, simple, and accurate.

For the identification problem, users aren't always cooperative. For example, a facial recognition system in a casino is something that casino cheaters will want to avoid. This kind of uncooperativeness makes the issue of both enrollment and recognition far more difficult.

7.4.1 Types of Errors

There are two types of errors in biometric recognition. The first occurs when Bob poses as Alice, and the system mistakenly authenticates Bob as Alice. The rate this happens at is known

as the *fraud rate*. The second happens when Alice tries to authenticate as herself yet the system fails to authenticate. This is known as the *insult rate*. Increasing or decreasing either of these rates will influence the other in the opposite way. The *equal error rate* is the rate for which the fraud and insult rate are the same. This is a useful measure for comparing different biometric system.

7.4.4 Biometric Conclusions

Biometrics are clearly better than passwords, and in practice they are difficult (not impossible) to forge. It is possible to build in protections against—for example—copies of a fingerprint, but this makes the system more expensive and so less desirable.

With biometrics many attacks are software-based, where an attacker tries to subvert the software that does to comparison to the database. While a broken password can be replaced, there is no replacing a biometric.

7.5 Something You Have

Smartcards or other hardware tokens can be used for authentication too. These follow the "something you have" principle. Since a key is used which can be completely random, password guessing attacks can be eliminated.

An example here is a password generation device. If Alice wants to authenticate herself to Bob, Bob sends a random challenge R which Alice enters into her password generator together with her pin code. The password generator responds with $h(K, R)$ which she sends back to Bob. Bob can then verify this.

7.6 Two-Factor Authentication

Whenever an authentication requires multiple of the three named authentication methods, we call this two-factor (or multi-factor) authentication. This is naturally more secure than single-factor authentication.

7.7 Single Sign-On and Web Cookies

Users don't like entering their login information often. A convenient solution to this is to have Alice authenticate once and have the result automatically "follow" her around the Internet. This is known as single sign-on. There is multiple approaches to this, and as with many things there is the Microsoft way and the "everybody else" way.

Websites often place cookies when users are browsing the Web. Cookies are not secure, but websites often use them for purposes of single sign-on, where first a password is required, but afterwards the cookie is sufficient to authenticate Alice.

Chapter 8

Authorization

8.1 Introduction

Authorization is concerned with restrictions of actions of authenticated users. The issue here is how we enforce restrictions on users who are already authenticated.

8.3 Access Control Matrix

A classic view of authorization is Lampson's access control matrix. This matrix contains all the information needed to make authorization decisions. On the rows we have all the users, on the columns we have the different files that we are concerned with. In each cell we then have any combination of `r`, `w`, and `x` meaning read, write, and execute, respectively. Important is that anything can be treated both as a subject and an object at the same time. For example, Bob may make changes to some program. This program can then make changes to some other files.

8.3.1 ACLs and Capabilities

There exist practical issues in managing access control matrices. For a large system with may users, these get extremely large very fast. This will manifest itself in both significant storage size but also in performance penalties for lookup operations.

We can split the matrix up into manageable piece. We can either split these by file or by user. We call the first an ACL, that is, access control list. We could also store them by user. This is known as a C-list, that is, a capabilities list.

In a C-list, the association between users and files is built in to the system. When using ACLs a separate method is needed for associating users to files. In reality, C-lists have several security advantages over ACLs too.

8.3.2 Confused Deputy

Suppose Alice, who has `x` for the compiler program and no rights to change the file named `BILL`, and we have a computer with `rx` privileges and `rw` privileges to `BILL`.

If Alice now calls the compiler to compile something to the file `BILL`, the compiler will happily overwrite it, even though Alice shouldn't be able to. With ACLs it's more difficult to avoid this. When using capabilities this is far easier, since capabilities are easily delegated.

Since capabilities are more complex to implement and have a higher overhead, they are often not used in practice. ACLs are used in practice much more regardless of their drawbacks.

8.4 Multilevel Security Models

Security models are descriptive, not proscriptive: they tell us what needs to be protected but not how to provide this protection. In multilevel security, the subjects are users and objects are the data to be protected. Classifications apply to objects, while clearances apply to subjects.

As an example we will use the classification of the U.S. Department of Defense:

$$\text{TOP SECRET} > \text{SECRET} > \text{CONFIDENTIAL} > \text{UNCLASSIFIED}$$

A subject with SECRET clearance is allowed to access object classified SECRET and lower, but not TOP SECRET.

A difficult issue to solve is that of granularity vs. aggregation. It is entirely possible to construct a document where each paragraph is individually UNCLASSIFIED but the overall document is TOP SECRET. On the flip side, someone might be able to gain TOP SECRET data from UNCLASSIFIED documents.

We consider two specific MLS models. In these examples we define O and S to be an object and subject, respectively. We define the level of either as $L(O)$ or $L(S)$.

8.4.1 Bell-LaPadula

In the BLP model there is minimal requirements that any MLS system must satisfy. It consists of the following two statements:

- **Simple Security Condition —** Subject S can read object O iff $L(O) \leq L(S)$.
- ***-Property —** Subject S can write object O iff $L(S) \leq L(O)$.

The first of these two simply requires that a user cannot read information above their security level. The second property prevents—for example—TOP SECRET information being written to a SECRET document, leaking this TOP SECRET information.

BLP is very simple and this has lead to some criticisms. In attempt to poke holes in BLP, McLean defined a "system Z" in which temporarily reclassifying objects is allowed. This yields an indirect violation of the star property, even though no actual violation occurs. In response to this, BLP was fortified with a *tranquility property* which has two versions:

The strong tranquility property states that security labels can never change. This removes "system Z" from the BLP reals, but is also impractical. The other is the weak tranquility property, which says a security label can change as long as such a change doesn't violate some "established security policy". What is meant with this is generally considered unclear and thus nearly meaningless for analytic purposes.

8.4.2 Biba's Model

Biba's model deals with integrity instead of confidentiality, which is what BLP deals with. It is—in essence—an integrity version of BLP. We define $I(O)$ and $I(S)$ to mean the integrity of the object and/or subject contained. We also say that if we trust the integrity of object O_1 and not of O_2, the composite object O of O_1 and O_2 is also not to be trusted.

Biba's model is defined by two rules:

- **Write Access Rule —** Subject S can write object O iff $I(O) \leq I(S)$
- **Biba's Model —** A subject S can read the object O iff $I(S) \leq I(O)$

The first rule is taken to mean that we don't trust anything that S writes any more than we trust S. Biba's model states that we can't trust S any more than the lowest integrity object S has read.

Biba's model is very restrictive, since no object S can ever view an object at a lower integrity level. In some cases it is desirable to replace it with the following rule:

- **Low Water Mark Policy —** If subject S reads object O then $I(S) = \min\{I(S), I(O)\}$

Under the low water mark policy, S can read anything given that the integrity of S is downgraded after accessing an object at a lower level.

8.5 Compartments

Often, simple hierarchical security labels are not sufficient to deal with a realistic situation. In practice, we often also use compartments to restrict flow among security levels. This is denoted as

$$\text{SECURITY LEVEL}\{\text{COMPARTMENT}\}$$

Compartments serve the *need to know* principle, where subjects are only allowed to access information that they must know. If they do not have a legitimate need to know everything, then compartments can be used to limit the information that a subject can access.

Compartments within or across a security level are not comparable in the way that security levels in a MLS are. Only security levels where one of the compartment sets is a subset of the other can be compared. For a concrete example of this, see the book.

8.6 Convert Channel

A covert channel is a communication path not intended by system designers. These exist in many situations but are very prevalent in networks. MLS systems designed to restrict legitimate channels of communication do little to prevent covert channels.

Three things are required for a covert channel to exist: (1) the sender and receiver must have access to a shared resource; (2) the sender must be able to vary some property of the shared resource that the receiver can observe; (3) the sender and receiver must be able to synchronize their communication.

Generally, companies don't bother trying to eliminate all covert channels. The DoD—for example—call for reducing their covert channel capacity to no more than one bit per second.

8.7 Inference Control

Specific information can be obtained from general questions. For example if we know that the mean salary of computing science professors at SJSU is $100.000 per year, and we have a list of computing science professors that coincidentally contains just a single person, then we know this person's salary. There is several techniques used in inference control that aims to reduce the possibility for the above. One technique is *query set size control* where no result is returned if the set size is too small. There is many more of these methods, but none are fully satisfactory. We will look more into this in the chapter regarding Privacy.

Chapter 9

Simple Authentication Protocols

9.1 Introduction

Protocols are the rules that are followed in some interaction. A security protocol must meet some specified security requirements, but we also want them to be efficient both in computational cost and network bandwidth usage. A security protocol also shouldn't be too fragile nor should it break when the environment in which it is employed changes.

Some of the largest security challenges in protocols today exist de to the fact that these protocols were designed for an academic environment, which the current-day internet is absolutely not.

9.2 Simple Security Protocols

The first simple security protocol we consider is that for entry into a secure facility. Employees needed to enter the facility have some badge and an accompanying PIN. The protocol can be described as follows: (1) insert the badge into a reader; (2) enter PIN; (3) is the PIN correct? If yes: enter building; if no: get shot by a security guard. This is virtually identical for a system that is used for ATM withdrawals, for example.

The military has protocols that are more specialized. An example of this is the IFF (identify friend or foe) protocol. A simple example of this is the following: A radar detects an aircraft approaching its base. A random number, or challenge, N is sent to the aircraft. All friendly aircraft have a key K that they use to encrypt $E(N, K)$ which is computed and sent back. Enemy aircraft do not have K and so they cannot send back the required response.

An easy way to attack this system it using a man-in-the-middle attack. When a friendly aircraft enters the airspace of a radar, an enemy aircraft enters at the same time. The enemy aircraft cannot obtain $E(N, K)$ and sends the challenge on to the friendly aircraft that can in such a manner that this aircraft doesn't know it came from the enemy. The enemy aircraft can then simply forward the response to the radar and it now identifies as a friendly aircraft.

9.3 Authentication Protocols

Alice must prove to Bob that she's Alice, where they are communicating over a network. In many cases it's sufficient for Alice to prove to Bob that she's Alice, though sometimes mutual authentication is required. Having a one-way protocol and using it both ways isn't always secure, so we must consider this as a special case.

In addition to authentication we always need a session key. This is a symmetric key that will be used to protect the confidentiality and/or integrity of the current session provided the authentication succeeds.

A simple authentication method is sending a password. This happens in three steps:

(1) Alice $\rightarrow$ Bob: "I'm Alice";

(2) Bob $\rightarrow$ Alice: Prove it;

(3) Alice $\rightarrow$ Bob: *password*.

This protocol is simple yet insecure. Trudy can observe these messages and send them to Bob exactly as Alice did. We could try hashing or encrypting Alice's password, but a replay attack is still possible. We need a unique challenge, a number only used once or *nonce*, in the authentication.

Our above protocol including a nonce would be the following:

(1) Alice $\rightarrow$ Bob: "I'm Alice";

(2) Bob $\rightarrow$ Alice: Nonce;

(3) Alice $\rightarrow$ Bob: $h(\text{Alice's password}, Nonce)$.

The downside of this method is now that Bob must know Alice's password to verify that the response is correct. Since both Alice and Bob are usually machines, we will liberate ourself from passwords and use keys instead.

9.3.1 Authentication Using Symmetric Keys

When discussing security with regard to protocols we concern ourselves with attacks on the protocols, and we assume that the cryptography is secure.

Suppose that Alice and Bob share the symmetric key K_{AB}. Alice can authenticate herself to Bob by encrypting a nonce R with the symmetric key. So, Bob sends R to Alice, and Alice returns $E(R, K_{AB})$. This is not susceptible to a replay attack, and it is secure. This protocol however does lack mutual authentication.

We may propose mutual authentication by using two nonces, where Alice starts by sending Bob a nonce R_A, and Bob replies with a different Nonce R_B, and $E(R_A, K_{AB})$. Alice then responds with $E(R_B, K_{AB})$. Sadly this can fail with a man-in-the-middle attack, where Trudy does the first and second step: sending the nonce and receiving another nonce plus Bob's reply that is encrypted with K_{AB}. She can then send this second nonce as the first nonce in a separate

request to Bob and receive the encrypted $E(R_B, K_{AB})$, which she can use to reply to Bob in the first request. She can then let the second request be timed out.

This is not a hard issue to solve; we simply let actors add their names into the encrypted message. The mutual authentication becomes the following:

(1) Alice $\rightarrow$ Bob: "I'm Alice", R_A;

(2) Bob $\rightarrow$ Alice: R_B, $E(\text{"Bob"}, R_A, K_{AB})$;

(3) Alice $\rightarrow$ Bob: $E(\text{"Alice"}, R_B, K_{AB})$.

9.3.2 Authentication Using Public Keys

One-way authentication using public keys can be done in two ways. Alice starts the interaction. Bob can then either send $\{R\}_{\text{Alice}}$ or R to Alice. Alice uses her private key on what she receives from Bob, and so either returns R or $[R]_{\text{Alice}}$ respectively.

This method has the problem that Trudy can pose as Bob, and so she can get Alice to apply her private key to whatever Trudy sends Alice. This means that someone should *never* use the same key for encrypting and authentication or signing.

9.3.3 Session Keys

Even when a symmetric key is used for authentication, we want to use a session key to encrypt data within each connection. This key is to be established as part of the authentication protocol such that when authentication is complete, we have also established a shared symmetric session key.

A way to do this that also provides mutual authentication is the following:

(1) Alice $\rightarrow$ Bob: "I'm Alice", R;

(2) Bob $\rightarrow$ Alice: $\{[R, K]_{\text{Bob}}\}_{\text{Alice}}$;

(3) Alice $\rightarrow$ Bob: $\{[R+1, K]_{\text{Alice}}\}_{\text{Bob}}$.

Of course we can swap the encryption on Bob's side for Signing on Alice's side and vice versa. We end up with a shared key K at the end of this procedure.

9.3.4 Perfect Forward Secrecy

Suppose Bob and Alice share a long-term symmetric key K_{AB}. Then if they want perfect forward secrecy (PFS) they can't use this as their encryption key. Instead, they must use a session key K_S and forget K_S after it's no longer needed. If we use the method previously discussed and send $E(K_S, K_{AB})$ then PFS is not provided, since Trudy breaking K_{AB} would yield K_S being compromised, which is exactly what PFS is trying to avoid.

The most elegant way to achieve PFS is by using Diffie-Hellman key exchange to establish the shared key. Alice and Bob can encrypt this Diffie-Hellman exchange using their shared symmetric key K_{AB} to prevent a man-in-the-middle attack here.

Chapter 10

Real-World Security Protocols

10.1 Introduction

In this chapter, several widely used real-world security protocols are discussed.

10.2 SSH

SSH creates a secure tunnel which can be used to secure otherwise insecure remote commands to a system. SSH authentication can be based on public keys, digital certificates, or passwords. We discuss a simplified version based on certificates.

SSH is illustrated in Figure 10.1. In this diagram, we use the following notation:

- certificate_A is Alice's certificate
- certificate_B is Bob's certificate
- CP is the proposed crypto and CS is the selected crypto
- S_B is $[H]_{\text{Bob}}$
- S_A is $[H, \text{Alice}, \text{certificate}_A]_{\text{Alice}}$
- H is $h(\text{Alice}, \text{Bob}, \text{CP}, \text{CS}, R_A, R_B, g^a \bmod p, g^b \bmod p, g^{ab} \bmod p)$
- K is $g^{ab} \bmod p$

and as usual, h is a cryptographic hash function.

In general, Alice and Bob both send each other nonces, and Alice sends a list of proposed crypto parameters, of which Bob chooses one. Then, Diffie-Hellman is used to establish a session key. Bob also sends his certificate along with all the previous parameters discussed hashed, then signed by him. Alice responds with her identity, her certificate, and the H, her identity, and certificate signed.

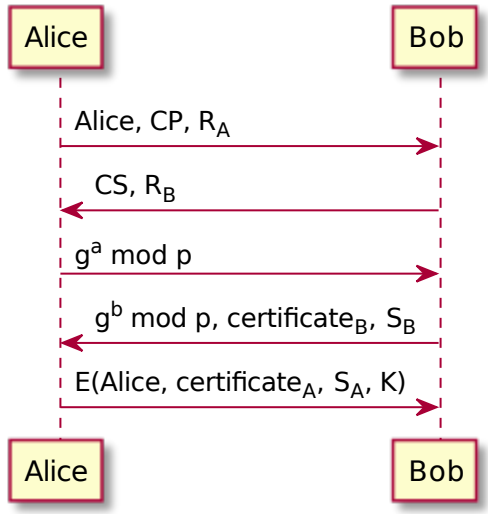

Figure 10.1: SSH Illustrated

After all these steps we have a secure connection with mutual authentication and a shared symmetric key. We also have an agreement on which crypto parameters to use.

10.3 SSL

SSL is the protocol that secures the Web. It is most often used with HTTP in the application layer, and TCP in the transport layer. SSL lives in the socket abstraction that the transport layer provides to the application layer.

The general idea behind SSL is that Alice asks Bob to talk securely, then Bob provides his certificate. Alice then responds with a shared key that is encrypted with Bob's private key. This key can then be used to talk securely. Just doing this directly isn't very secure since Bob isn't authenticated and Alice can't possibly be sure she is actually talking to Bob. Alice isn't authenticated to Bob either, though this is often enough not a problem on the Internet.

A basic version of actually secure SSL is shown in Figure 10.2. In this figure,

- S is the pre-master secret
- $K = h(S, R_A, R_B)$
- msgs is shorthand for "all previous message"
- CLNT and SRVR are literal strings

As can be seen, Alice first informs Bob that she wants to establish an SSL connection. She includes a list of ciphers that she supports and a nonce R_A. Bob responds with his certificate and a selected cipher. He returns a nonce R_B as well. Alice sends the so-called pre-master secret

S which she randomly generates along with a hash that is encrypted with K. Bob responds with a similar hash. Alice can now compute whether Bob has received the messages correctly and she can authenticate Bob, since only Bob should have been able to decrypt $\{S\}_{\text{Bob}}$.

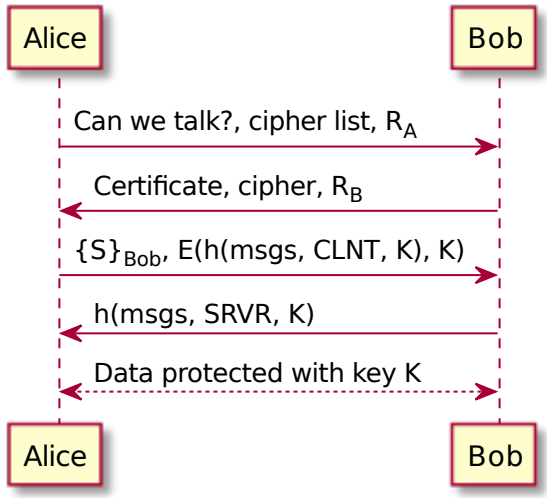

Figure 10.2: SSL Illustrated

Now, Alice has authenticated Bob and Alice and Bob now have a shared session key K which they can use for secure conversation. In reality, six keys are derived from $h(S, R_A, R_B)$, namely two encryption keys (one for either direction of conversation), two integrity keys used in the same way, and two initialization vectors, one for the client and one for the server.

10.3.1 SSL and the Man-in-the-Middle

SSL is resistant to a man-in-the-middle attack. Bob's certificate must be signed by a certificate authority, so Trudy cannot send her own certificate to Alice when performing a MiM attack. If Trudy simply sends along Bob's certificate, the MiM attack would not succeed, since Alice would simply authenticate Bob as was the plan all along.

Typically SSL is a Web browsing protocol. If the certificate checking fails when Alice is browsing, then Alice is informed if this, and in true user fashion she will ignore it and continue browsing anyway, leaving here data out in the open for Trudy....

10.3.2 SSL Connections

An SSL session is established as shown in Figure 10.2. This is relatively expensive since public key operations are involved. As such, it's not preferred for there to be a new SSL session for each new HTTP connection. This is why SSL also offers connections provided that a session already exists. After establishing one SSL session, Alice and Bob share a session key K which they can reuse to establish a new connection.

10.3.3 SSL Versus IPSec

IPSec is short for Internet Protocol Security. Its purpose is similar to that of SSL though the implementation is very different.

The most obvious difference is that they operate on different layers in the protocol stack. SSL lives at the socket layer. IPsec lives in the network layer and is not directly accessible from the user space, rather it's part of the operating system. SSL and IPSec both provide encryption, integrity protection, and authentication. SSL is quite simple whereas IPSec is complex and as such includes some significant flaws.

IPSec is used for more than the Web, where SSL is prevalent. It's often used to secure VPNs. IPSec is also required in IPv6, so if (read: when) IPv6 becomes mainstream, IPSec will be ubiquitous.

10.4 IPSec

IPSec consists of two main parts: The first is the Internet Key Exchange (or IKE) which provides for mutual authentication and a session key. This part has two phases which are analogous to SSL sessions and connections. The second part is the Encapsulating Security Payload and Authentication Header, or ESP/AH. ESP provides protection encryption and integrity protection to IP packets, whereas AH only provides integrity.

In IKE Phase 1 we establish a so-called IKE security association, or IKE-SA. In Phase 2, an IPSec security association, IPSec-SA, is established. Phase 1 corresponds to an SSL session while Phase 2 is comparable to an SSL connection.

In Phase 1 we can choose one of four different key options: public key encryption (original or improved version), digital signature, or symmetric key. We can also choose to use main mode, or aggressive mode, yielding eight different versions of IKE Phase 1, of which we discuss six.

10.4.1 IKE Phase 1: Digital Signature

We first consider digital signature main mode. We use the following abbreviations:

- CP is crypto proposed
- CS is crypto selected
- IC is initiator cookie
- RC is responder cookie
- $K = h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B)$
- $\text{SKEYID} = h(R_A, R_B, g^{ab} \bmod p)$
- $\text{proof}_A = [h(\text{SKEYID}, g^a \bmod p, g^b \bmod p, \text{IC}, \text{RC}, \text{CP}, \text{“Alice”})]_{\text{Alice}}$

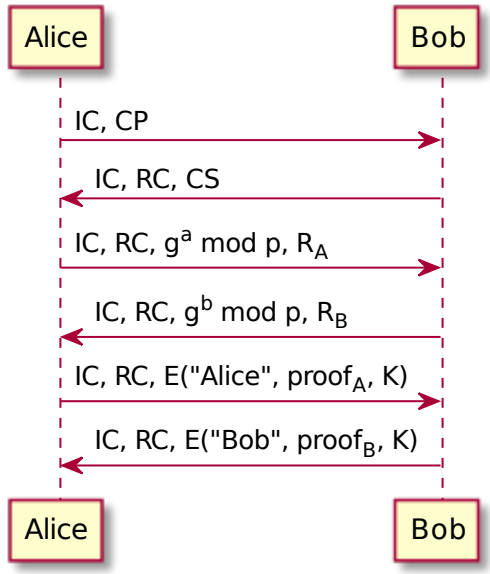

Figure 10.3: IKE Phase 1: Digital Signature Illustrated

The protocol is illustrated in Figure 10.3

In the above, a passive attacker—that is, one that cannot insert, delete, alter, or replay messages—is unable to make out Alice's or Bob's identity. In the aggressive mode, there is no attempt to hide these identities. The aggressive mode is shown in Figure 10.4

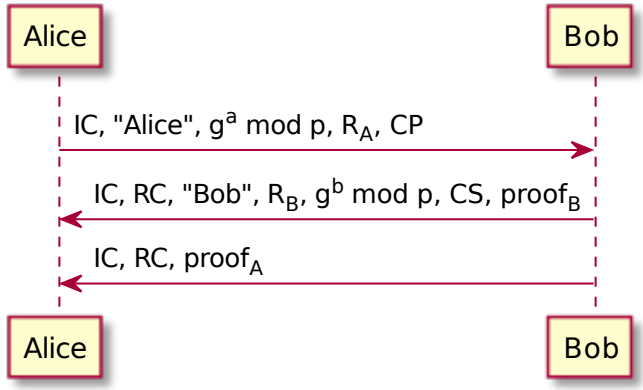

Figure 10.4: IKE Phase 1: Digital Signature (Aggressive Mode) Illustrated

10.4.2 IKE Phase 1: Symmetric Key

In the symmetric key option—both main and aggressive mode—the format is essentially the same as with Figure 10.3 and 10.4 but we interpret the notations differently. In main mode,

- K_{AB} = symmetric key known by Alice and Bob in advance
- $K = h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B, K_{AB})$
- $\text{SKEYID} = h(K, g^{ab} \bmod p)$
- $\text{proof}_A = h(\text{SKEYID}, g^a \bmod p, g^b \bmod p, \text{IC}, \text{RC}, \text{CP}, \text{Alice})$

One problem with this is that Alice's identity is encrypted with K_{AB}, but Bob cannot know to decrypt with K_{AB} before he knows that he is talking to Alice. The solution that the IPSec developers came up with is to use the IP-address of incoming packets to determine who he is talking to. Of course, now Alice must have a static IP address. If it changes, this mode fails. This issue does not present itself in aggressive mode.

Aggressive mode uses the same format as Figure 10.4, but instead of what is shown there, the key and the signature are computed as in symmetric key mode.

10.4.3 IKE Phase 1: Public Key Encryption

In public key encryption we assume that Alice and Bob have access to each other's certificates, without having to reveal their identities. Public key encryption main mode protocol uses the structure as shown in Figure 10.5 where the notation is the same as previous modes, except

- $K = h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B)$
- $\text{SKEYID} = h(R_A, R_B, g^{ab} \bmod p)$
- $\text{proof}_A = h(\text{SKEYID}, g^a \bmod p, g^b \bmod p, \text{IC}, \text{RC}, \text{CP}, \text{“Alice”})$

Aggressive mode is illustrated in Figure 10.6.

Interestingly, in the public key encryption version, Trudy can act like both Alice and Bob at the same time and successfully authenticate. This means she can make a conversation "happen" that didn't actually happen. While this seems like a problem, the IPSec developers praise this as a feature, since now any conversation has plausible deniability, where Alice and Bob can deny any conversation ever took place.

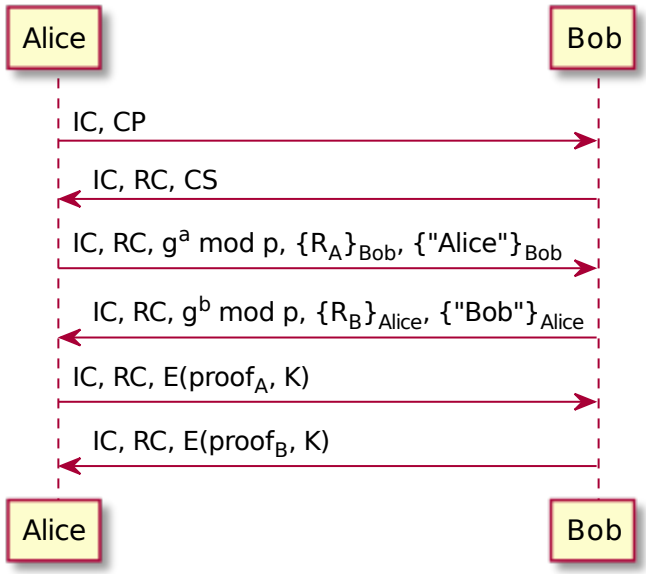

Figure 10.5: IKE Phase 1: Public Illustrated

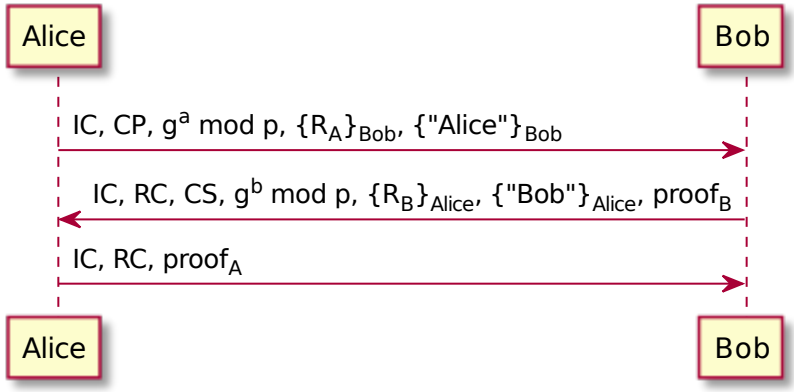

Figure 10.6: IKE Phase 1: Public (Aggressive Mode) Illustrated

10.4.4 IPSec Cookies

The purpose of IPSec cookies IC and RC is to make DoS attacks more difficult. These IPSec cookies however achieve no significant protection against DoS attacks.

10.4.5 IKE Phase 1 Summary

After Phase 1 we have mutual authentication and a shared session key. After Phase 1, Phase 2 must occur that is required once per application that wishes to use IKE-SA.

10.4.6 IKE Phase 2

In Phase 2, we establish IPSec-SA. Here the crypto proposal includes whether ESP or AH is used.

10.4.8 Transport and Tunnel Modes

To use IPSec in the network layer we can use either transport mode or tunnel mode. In transport mode, we sandwich a new ESP/AH header in between the IP header and the data. In transport mode, Alice's and Bob's identity are not kept secret. In tunnel mode, the entire IP packet is encapsulated in another IP packet. This results in the original IP packet not being visible to the attacker, assuming the packet is encrypted. If Bob and Alice are communicating directly, the new and old IP headers are exactly the same and so this mode would be useless. Tunnel mode was intended for firewall-to-firewall communication however, so this is secure.

10.4.9 ESP and AH

When we've decided whether to use transport or tunnel mode we decide whether we want confidentiality, integrity, or both. We must also consider the protection to apply to the header. We can choose between AH and ESP.

AH provides integrity on the data and some fields in the IP header, but no confidentiality. In ESP we have both integrity and confidentiality. Both apply to everything beyond the IP header, that is, the data. We can use ESP for integrity only, by using the so-called NULL cipher, that is,

$$\text{Null}(P, K) = P$$

10.5 Kerberos

Kerberos is a popular authentication protocol that uses symmetric keys and timestamps. Kerberos is designed for a far smaller scale than SSL and IPSec, such as a LAN or a cooperation.

Usually when we have symmetric keys per user pair with N users we would require N^2 keys. Kerberos lessens this to just N symmetric keys for N, by relying on a Trusted Third

Party (TTP). Each users shares one key with the KDC. The KDC then acts as a go-between that enables any pair of users to communicate securely.

Kerberos is used for authentication and to establish a session key that can subsequently be used for confidentiality and integrity between users. Any symmetric cipher can be used, but most often the algorithm of choice is DES.

A KDC issues various types of *tickets*. A ticket contains the keys and other information required to access network resources. Once special ticket is the *ticket-granting ticket*, or TGT, which is issued when a user logs in to the system, acting as that user's credentials. This TGT is then used to obtain ordinary tickets to access network resources. This is crucial to the statelessness of Kerberos.

Each TGT contains a session key, user ID, and expiration time. A TGT is encrypted with K_{KDC} and only the KDC knows this key. As such, only the KDC can read it.

10.5.1 Kerberized Login

When Alice wishes to log in to a system she presents her password to the computer. The computer determines Alice's key K_A from this password and uses this to obtain Alice's TGT from the KDC. This is sent back encrypted using K_A, which Alice's computer can decrypt so she can obtain the TGT.

10.5.2 Kerberos Ticket

Once Alice's computer has the TGT it can use this to request access to resources. Suppose Alice wants to talk to Bob, then Alice's computer presents the TGT to the KDC along with an authenticator. This authenticator is an encrypted timestamp to avoid a replay attack.

Once Alice has the ticket to Bob she can securely talk to Bob. Bob decrypts this ticket with his key to obtain the shared Alice-Bob key that they use for the subsequent conversation. Bob also verifies the timestamp Alice sent him, to avoid a replay attack.

Appendix A

Privacy

A.1 Data Privacy

In the context of privacy we are concerned with the collection, handling and dissemination of sensitive data. We consider data as private when it is personally identifiable information (PII), of which some is and some isn't sensitive, or when it is personal health information (PHI), all of which is considered sensitive. Aside from PII and PHI there is also a gray area. This is data then isn't per se personally identifiable, but may still be private data, such as social media posts, locations you visited, etc. This is data that's often collected by software that you may wish to keep private.

An important aspect when collecting data is how to anonymize this. For example, in 2007, Netflix released a dataset of user ratings. This dataset did not contain any PII, yet researchers were still able to recover 99% of the personal information that was removed, by using auxiliary IMDb data.

A.1.1 Pseudonymization vs. Anonymization

In anonymization we completely remove any personally identifiable information. This process is irreversible. In pseudonymization we change personally identifiable information to a pseudonym. For example, in anonymization, all names in a dataset may be removed, or changed to xxxxxx. In pseudonymization we may change all names to unique random strings.

A.2 Anonymization & Pseudonymization Techniques

A.2.1 *k*-Anonymity

The core idea in k-anonymity is given a dataset with private info, we require that each individual record cannot be distinguished from $k-1$ records of the same data set. This results in a $1/k$ probability of re-identification for each record/individual. Our objective here is to allow

some useful information to stay such that useful statistics can be extracted without possible re-identification.

We introduce the concept of quasi-identifiers (QIs): These are attributes of a record that do not unique identify an individual, but when combined with other QIs *may* lead to identification. We give adversaries access to only QIs after anonymizing them. We do not give them access to the unique attributes. Several methods exist to anonymize QIs: we can either suppress them completely by hiding individual attributes (e.g. replace with *) or we can generalize them, turning individual attributes into broader catagories (e.g. ZipCode = 2134*). Generally, we apply suppression to fully uniquely identifiable attributes, and generalization to quasi-identifiers.

After anonymization we wish to verify that each quasi-identifier tuple appears at least in k records. If so, we have achieved k-anonymity.

In reality, k-anonymization does not guarantee any privacy, since the values of sensitive attributes remain unchanged and may still be associated with a quasi-identifier in such a manner that personally identifiable information can be extracted. The adversary may also know more than just the QIs.

A.2.2 *l*-Diversity

The objective of l-diversity is to generate diversity of sensitive attributes given quasi-identifier groups. For distinct l-diversity we need to have at least l distinct sensitive values for each equivalence class. For probabilistic l-diversity we need the frequency of the most frequent value to be bounded by $1/l$. For entropy l-diversity, we need the entropy of the distribution of sensitive values to be at least $\log l$.

Given a quasi-identifier group q^* with l distinct values for the sensitive attribute, s is a sensitive attribute value from domain S and $P(E, s)$ is the fraction of records in q^* with the sensitive attribute value s, then the entropy for q^* is defined as

$$\text{Entropy}(q^*) = -\sum_{s \in S} p(q^*, s) \log(p(q^*, s))$$

A.2.3 *t*-Closeness

An equivalence class is t-close if the distance between the distribution of a sensitive attribute in this class and the distribution of the attribute in the whole table is no more than some threshold t. A table is said to have t-closeness is this holds for all the equivalence classes.

A.3 Differential Privacy

Differential privacy deals with the paradoxical situation where want a data set to be practical for learning information about a population while learning nothing about any individual that partook in the collection of data for this dataset. The exact definition of the problem to solve is the following:

> "The risk associated with privacy violation of an individual should not substantially increase as a result of participating in a statistical database"

We need a mechanism K here that for all pairs of very similar datasets D and D' behaves approximately the same, that is, if $K(D) = X$ and $K(D') = Y$ then X and Y should be indistinguishable. The most common method to achieve this is to add controlled noise (e.g. Laplace, Gaussian) to our data.

A randomized function K gives ε-differential privacy if for all datasets D and D' differing on one entry and all $S \subseteq \text{Range}(K)$

$$\Pr[K(D) \in S] \leq \exp(\varepsilon) \cdot \Pr[K(D') \in S]$$

and a randomized function K gives (ε, δ)-differential privacy if

$$\Pr[K(D) \in S] \leq \exp(\varepsilon) \cdot \Pr[K(D') \in S] + \delta$$

We can achieve differential privacy using the general Laplace(μ, b) with f:

$$f(x \mid \mu, b) = \frac{1}{2b} e^{\left(-\frac{|x-\mu|}{b}\right)}$$

where b is set to ε and μ is set to f's sensitivity, that is

$$\max_{D,D'} ||Q_D - Q_{D'}||_1$$

that is, the maximum difference in the values that the query Q may take on a pair of databases that differ only in one row.

A.4 Location Privacy

Your physical location history tells a lot about your habits, interest, health, relations, and political views. Many devices and services collect this data. There is a trade-off here, between accuracy and privacy. For the service experience, as much accuracy as possible is preferred, while as a user you may wish to have more privacy.

A way to introduce privacy is by adding a radius R to your coordinates x_i, y_i, where x_i, y_i is collected by—for example—a GPS system very accurately. Another way to manage this is by "blurring" the user's location, by just passing along the general region that a user is in. A time delay can also be used to limit real-time user tracking and we can implement k-anonymity, where any service is only passed the region of this user and its $k - 1$ "neighbors". In a dense place this will be more accurate than in scarce places.

Appendix B

PGP

PGP/GPG is a widely used software for encryption and signing. It also has a PKI to manage public keys. Access to private keys is managed per-user using password protection. PGP/GPG allows for confidentiality and integrity, that is, encryption and digital signing. It also provides non-repudiation.

In PGP/GPG there is no central certificate authority but instead there is a "Web of Trust", where every user decides for themselves who else to trust. Public keys are signed by other people that you may already trust. This way, you know you can also trust the newly signed certificate, building your web. Web of Trust is considered a failure by some, since it is difficult to implement technically and socially. Public keys may be retrieved from key servers and used, but there is no guarantee that these are not malicious since there is no central CA.