Chapter 1

Security

1.1 CIA Model

- Confidentiality
  - Cryptography: encryption and decryption
  - Access Control: some policy that limits access to certain users through personal ID, etc.
  - Entity Authentication: ensure the ID of someone
    - Something they have (token, key)
    - Something they know (password)
    - Something they are (fingerprint)
  - Physical security
- Integrity
  - Backups, checksums, ECC
  - Cryptography: MAC over the message or a digital signature using a private key (symmetric or asymmetric)
- Availability

1.2 Partial Plaintext Attack

Suppose transactions in a RFID system are encrypted as $C = Enc(k, M)$ for some 4-digit PIN k. Then, if a plaintext M_0 and its encryption C_0 is known, then an attacker need only try all 10^4 PINs to compute $C' = Enc(k_i, M_0)$ until $C' = C_0$, at which point they have the PIN. This is much less than the $10^4 \cdot 2^{|M|}$ that would normally be required.

1.3 Buffer Overflow

```
1 void main() {
2         char line[10];
3         gets(line);
4 }
```

Writing more than 10 bytes to `line` will begin to overwrite (in order):

- The old Frame Pointer of the caller
- The return address
- Local variables of the caller

In performing a buffer overflow attack, the goal is to overwrite that return address to point into some code that the attacker has inserted.

1.4 Spectre Attack

```
1 if (x < array1_size)
2         y = array2[array1[x] * 4096]
```

- If `x >= array1_size` and `array1[x]` contains (for example) a secret key, then if the conditions are right the CPU will perform the memory access in line 2 while checking the condition
  - Conditions
    - `array1_size` and `array2_size[k*4096]` are uncached
    - The key `k=array1[x]` is cached
    - The branch predictor assumes that the condition will likely be true (e.g. if many previous iterations were true, this will likely happen for most branch predictors)
  - The `array2` access will miss while the branch is still being checked, and the value `array2[k*4096]` will be placed in cache even though the condition was false
    - The time it takes to access this memory can be measured dependent on `k*4096`, from which the key k can be derived

1.5 Trusted Platform Module

- Standard for a secure co-processor
- Principle: root of trust and transitive trust
  - A layer X has a set D(X) called its **dependencies**
  - $L \in D(X)$ if at least one of the following is true:
    - L has RW access to the data of X
    - L has W access to the code of X
  - Notation: $L \leftarrow X$ means $L \in D(X)$
    - $\leftarrow$ is transitive
- Secure boot
  - Each segment X_i verifies X_{i+1} before passing execution to it
- Authentication Authority
  - Each layer has an AA that holds the secret key used to authenticate items in the layer above it
  - So X in layer J can call the AA in layer $k < J$ to generate auth data for itself that it can send to a remote party for verification
    - E.g. a digital signature using a secret key where the public key is known by the remote

Chapter 2

Practical Cryptographic Schemes

2.1 Pseudo-Random Sequence Generators (PRSG)

2.1.1 FSRs

A feedback shift register is an n-bit register with bits $a_{n-1}...a_0$ where at each cycle the last bit is determined according to a feedback function $f(a_{n-1}...a_0) = f(\vec{a})$. $(a_{n-1}...a_0) \leftarrow (a_n...a_1)$ where $a_n = f(\vec{a})$. The output bit is a_0. If you picture it with the LSB on the right, the output is on the right and the MSB gets loaded with the feedback function on the left in each cycle.

2.1.2 LFSR

An LFSR (Linear FSR) is an FSR where $f(\vec{a}) = \sum_{i=0}^{n-1} c_i a_i$ and $c_i \in \{0, 1\}$.

An m-sequence (or maximal length or pseudo noise sequence) is the output sequence with the maximal period for an LFSR. For an N-bit LFSR this is $2^N - 1$ bits long. An LFSR generates an m-sequence as its output if and only if its **characteristic polynomial** is **primitive**. LFSRs have the following properties:

1. All output sequences are either periodic or *eventually* periodic
2. The minimal polynomial of an LFSR sequence is a divisor of its characteristic polynomial
3. For a given LFSR, all its m-sequences are *shift-equivalent* and *shift-distinct* to all m-sequences **not** generated by the LFSR (i.e. by other LFSRs)

To explain property 2, think of it in the following way. An LFSR A has some characteristic polynomial $f(x)$. It generates a set of sequences depending on the initial conditions. Consider one such sequence $\{a_i\}$. The minimal polynomial of this sequence is the *lowest degree* polynomial that, if it were the characteristic polynomial $h(x)$ of an LFSR B, B would also generate $\{a_i\}$. Property 2 states that $f(x)|h(x)$ (i.e. $f(x) = p(x)h(x)$ for some $p(x)$). As a corollary, if $f(x)$ is *irreducible*, it is the minimal polynomial of all the sequences it generates. Furthermore, the minimal period of any sequence $f(x)$ generates is equal to its period.

2.1.3 Galois Fields

A Galois Field is a **field** with a finite number of elements. All such fields with p^n elements for prime p (denoted GF(p^n)) are isomorphic to $\mathbb{Z}_{p^n}$. The binary field GF(2) is unique with the elements $\{0, 1\}$: the operations * and + are a logical AND and XOR respectively.

2.1.4 Polynomials over GF(2)

The feedback function of an LFSR can be represented as a polynomial with coefficients in GF(2):

$$\sum_{i=0}^{n-1} c_i x_i \leftrightarrow t(x) = \sum_{i=0}^{n-1} c_i x^i \tag{2.1}$$

Such a polynomial is **irreducible** if it cannot be written as the product of two (non-constant) polynomials. For example $x^2 - 2$ is irreducible over the integers but not over the reals.

Note that we can define the **period** of a polynomial as the *minimum* r such that $f(x)|x^r - 1$. A polynomial is **primitive** if it is irreducible and its period is $2^N - 1$. Note that primitive polynomials must have a constant term, otherwise we could factor out an x and it would thus be reducible.

2.1.5 Correlation

The cross correlation of two N-length binary sequences is given by:

$$C_{a,b}(\tau) = \sum_{i=0}^{N-1} (-1)^{a_i + b_{i+\tau}} \tag{2.2}$$

It measures how similar a shifted version of b is to a, and is thus a function over τ. When $a = b$ this is also called the **autocorrelation**. If the lengths are not the same, we can use the following notation (where M is the length of b:

$$C_{a^T,b}(\tau) = \sum_{i=0}^{T-1} (-1)^{a_i + b_{(i+\tau) \mod M}} \tag{2.3}$$

Note that the operation $+$ in Equation 2.3 is XOR

2.1.6 Linear Span Attack

The degree of the minimal polynomial of a sequence **a** of length N is called its **linear span**. In other words, the linear span of the sequence is the minimal l such that an l-bit LFSR with initial state $(a_0, a_1, ...a_{l-1})$ generates the sequence $\{a_0...a_N\}$.

Given **a** can we make an LFSR that generates it? Obviously $f(x) = x^N - 1$ would work since we just load the whole sequence into the LFSR. Using the **Berlekamp-Massey algorithm**, we can construct the *minimal* LFSR that generates **a**. *By definition*, this LFSR will have degree equal to the linear span of **a**, denoted LS(**a**).

As it turns out, the Berlekamp-Massey algorithm only needs 2LS(**a**) bits to perform the construction. This means that if **a** has linear span n, and $2n < N$, we can use the BM algorithm to construct the LFSR and then run it to generate the remaining $N - 2n$ bits. This is called a linear span attack.

2.1.7 Nonlinear Generators

The goal here is to maintain the randomness and efficiency that comes from using m-sequences of LFSRs as random bitstreams, while *increasing* the linear span to avoid a linear span attack. In **filtering sequence generators** we feed the full LFSR state into a nonlinear function that serves as the output bit. In **combinatorial sequence generators** we feed the output bits of M LFSRs into a nonlinear function that serves as the output. In **clock controlled** generators we use some clocked FSM to control which output bits of LFSR1 get fed into LFSR2 and then to the output.

2.1.8 Correlation Attack

When we use a **combinatorial sequence generator**, we can perform this attack to recover the initial states of the m LFSRs. These states are assumed to be the keys (e.g. a stream cipher scheme could be to place the key in the LFSRs and then use the output stream as a one-time pad on the data).

Let the size of LFSR i be n_i. Let the output stream of LFSR i be denoted X_i. Note that $0 \leq i < m$. We assume that we have the output stream of Z, it has length T and is denoted z. We can perform exhaustive search by trying all

possible initial states and seeing if it generates Z. The complexity of this is

$$T_0 = \prod_{i=0}^{m-1} 2^{n_i} \tag{2.4}$$

Under some assumptions, we can improve this:

- X_i are iid binary random variables
- Z is a random variable whose value is $h(X_0, X_1, ...X_{m-1})$
- $I_C \subseteq \{X_0, ...X_{m-1}\} \neq \emptyset$ is a set of LFSRs which, based on the structure of the overall PRSG and function h, we know to be **correlated** with Z

This last point is important; based on Kerchoff's principle, the structure of the PRSG is usually known (including the sizes and polynomials of all the LFSRs and the function h). It is only the key that is secret.

The main idea here is that if we know that an LFSR is correlated with the output sequence, we can find its initial state *independent* of the states of the other LFSRs. We will **still be doing an exhaustive search** on the LFSR (by trying every possible shift of its m-sequence), but only on one LFSR at a time.

The other key idea is that if we know $P[Z = 0|X_k = t] > \frac{1}{2}$ or $P[Z = 1|X_k = t] > \frac{1}{2}$ for $t \in \{0, 1\}$, then for a sufficiently large sample size, we can assume that the initial state of LFSR_k that generates the **highest** correlation between Z and X_k is most likely to be the correct one. In particular, the probability that this assumption is wrong goes to zero as the size of the output sequence we are checking goes to infinity.

With this, the steps to perform a correlation attack are as follows:

1. Write the truth table for the combining function h with inputs X_i (the 0th bit of LFSR_i)
2. Use the formula $P[Z = i|X_j = k] = \frac{P[Z=i \wedge X_j=k]}{P[X_j=k]} = \frac{\#[Z=i \wedge X_j=k]}{\#[X_j=k]}$ (where # means the number of times the event shows up in the truth table) to compute all the conditional probabilities
3. Decide the set I_C using step 2 (X_i such that Z's value has a skewed distribution relative to the value of X_i)
4. For each element $X_i \in I_C$
   (a) Figure out the m-sequence for this LFSR and call it x_i (recall that this is unique for a given LFSR up to shift-equivalence. see subsection 2.1.2)
   (b) Compute $C_{z^T,x_i}(\tau)$ using Equation 2.3 for $\tau \in [0, 2^{n_i})$ (i.e. to check every possible shift)
   (c) Compute $\tau_0 = \text{argmax}_\tau |C_{z^T,x_i}(\tau)|$. Set the initial state of LFSR_i to x_i shifted by τ_0

The complexity of a correlation attack is given by

$$T_1 = \sum_{i=0}^{m-1} 2^{n_i} \tag{2.5}$$

2.2 Stream Ciphers

A stream cipher uses a key to generate a pseudorandom sequence based on the methods described in section 2.1. This sequence is then XORed with the data (which may be of arbitrary size, hence stream instead of block) to encrypt it, analogously to the one-time pad system.

2.2.1 Operation

Stream ciphers consist of two phases: **key initialization** and **PRSG running**. During key initialization, the initial vector IV and key K are mixed. The result is then passed as the initial value to the PRSG which begins outputting a key stream that is XORed with the plaintext. The decryption scheme does the **exact same thing** and therefore needs the same IV.

2.2.2 RC4

RC4 is meant for efficient software implementation. The key-initialization algorithm (**KIA**) uses the key K to generate a permutation of all n-bit integers in memory. The PRSG increments a value i and uses $S[S[i] + S[i + S[i]]]$ modulo $2^n - 1$ as the output block (then swaps the values it used). Yes it seems like nonsense because it was, but since the system was kept secret for a while no one knew. It was used in WEP which caused a huge security issue when the code was leaked and attacks were discovered.

2.3 Block Ciphers

A block cipher involves an encryption and decryption algorithm. Each algorithm is essentially a function which is a one-to-one mapping of n-bit vectors. n is called the block size. The important property (correctness) is that $D \circ E = \mathbb{1}$ (that is, encrypting then decrypting a plaintext gives back the original message).

Two principles in designing a block cipher:

- **confusion**: the statistical relationship between plaintext and ciphertext should depend on many or all bits of the key
- **diffusion**: each plaintext bit should affect multiple ciphertext bits (small change in input $\rightarrow$ large change in output)

Examples of block ciphers include DES and AES. Common structures used in them include

- S-boxes: substitution - essentially tabularly represented functions that map input blocks to output blocks
- Feistel structures - an NLFSR

Another useful way to think of a n-bit block cipher is as the set of all permutations over all 2^n n-bit vectors, where the key selects which permutation to use.

2.3.1 Block Cipher Modes

We need some way to use the block cipher on data that may not be equal to the block size. Modes are the way to achieve this.

Electronic Codebook (ECB)

Divide the input into block-sized units and run the block cipher on each one individually

Cipher Block Chaining (CBC)

ECB causes two equivalent blocks to have the same output, so CBC instead uses the output of one block as an initialization vector to the next.

Cipher Feedback (CFB)

Use the key to generate a keystream from ciphertext bits. $K_i = Enc_K(C_{i-1})$ gets XORed with the plaintext to produce $C_i = M_i \oplus K_i$. This turns the block cipher into a stream cipher.

Counter Mode (CTR)

In CFB, instead of using the ciphertext as the input to the block cipher, use a counter which gets incremented for each bit.

2.4 Birthday Attacks

Given a set S of size n, the probability that two of m randomly chosen elements (with replacement) are the same is given by Equation 2.6

$$\approx 1 - e^{\frac{-m^2}{2n}} \leq \frac{m^2}{2n} \tag{2.6}$$

The practical result is that if an element can take N different values, you can expect it to take on the same value after observing about $\sqrt{N}$ instances. That is, for an n-bit value, we can expect a collision after $\sqrt{2^n} = 2^{\frac{n}{2}}$ instances.

2.4.1 Time-Memory Tradeoff Attack

Also known as a *meet-in-the-middle* attack. Suppose we know that some system is encrypting a fixed plaintext p using a randomly generated key k of n bits (for example if every transaction in some banking system starts with encrypting 'hello'). Then we can choose $2^{\frac{n}{2}}$ random keys and store the encryption of p under each of them. That is, we can compute the set $T = \{(k_i, Enc_{k_i}(p)) \mid i \in [0, 2^{\frac{n}{2}})\}$. We can expect to observe a key k for which we have already precomputed $Enc_{k_i}(p)$ to be used after observing about $2^{\frac{n}{2}}$ transactions.

That is, we have traded off the memory of storing T for the time of observing another $2^{\frac{n}{2}}$ transactions, and the time complexity becomes $2^{\frac{n}{2}}$.

2.5 Security Models

2.5.1 Chosen Plaintext Attack (CPA)

An attacker is given access to an encryption oracle that will provide $Enc(p)$ for any plaintext it is given. The goal of the attacker is to figure out which of two plaintexts m_0 and m_1 corresponds to a given ciphertext c. Note that since the attacker could ask for $Enc(p)$, no deterministic encryption can be CPA secure.

With CBC for example, any further queries of $Enc(m_0)$ and $Enc(m_1)$ will be different since the oracle holds some state.

2.6 Hash Functions

Three important properties of a hash function:

- **Collision Resistance**: Finding x, y s.t. $h(x) = h(y)$ is difficult
- **Second pre-image resistance**: Given x, finding y s.t. $h(x) = h(y)$ is difficult
- **Pre-image resistance**: Given z, finding x s.t. $h(x) = z$ is difficult

Note that satisfying the first property implies the other two are satisfied. The reverse is not true.

A hash function maps n bits to m bits where $n > m$, and thus multiple values may map to the same thing. The design of hash functions is thus similar to block ciphers but with the requirement of a one-to-one mapping being relaxed.

2.7 Message Authentication Code (MAC)

A MAC is used in symmetric-key cryptography to verify that a message was sent by someone who knows the key k. There are many ways to implement it using keyed hash functions, block ciphers, or authenticated encryption. The key property is **unforgeability** - even if an attacker has access to an oracle that produces $t = MAC_k(m)$, it cannot produce (m, t) such that $\text{Verif}(m, t) = 1$ and m was not part of a query to the oracle.

2.8 Public Key Cryptography

2.8.1 One-Way Functions

A fundamental concept in public key cryptography is that of a one-way function. A function is 'one-way' if it is 'easy' to compute its output, but difficult to compute the input given its output. A *trapdoor* one-way function is a one-way function that, if one knows some special piece of information, makes it easy to compute the input given the output.

2.8.2 Diffie-Hellman Key Exchange

The setting for Diffie-Hellman key exchange is a finite field $\mathbb{F}_p$ with p elements. g is a 'primitive element' of the field; that is, each nonzero element of $\mathbb{F}_p$ can be written as g^i for some integer i. Note that in the finite field, the addition and multiplication operations are done modulo p but this will often be omitted for brevity. This is equivalent to:

$$g^{p-1} = 1 \tag{2.7}$$
$$g^i \neq 1 \quad \forall \ 1 \leq i < p-1 \tag{2.8}$$

Each user i has a keypair (i, g^i). By the discrete log problem, i is difficult to compute given g^i (but not vice versa). Two users a and b can compute a shared key k by sending eachother their public key. Then, user a can compute $g^{ab} = (g^b)^a$ using their private key and user b can do the same. Then the shared key is g^{ab} which can be used to perform symmetric encryption/authentication.

Lagrange's Theorem

Equation 2.8 requires us to check $p-1$ values. Instead, we can check if an element is primitive using Lagrange's theorem. Simply put, an element a is primitive if $a^{p-1} = 1$ and $a^q \neq 1$ for all prime factors q of $p-1$.

For example, if $p = 47$ then since $(p-1) = 46 = 23 \cdot 2$ we only need to check the following to verify if a is primitive:

$$a^{p-1} = a^{46} = 1$$
$$a^{23} \neq 1$$
$$a^2 \neq 1$$

2.8.3 RSA

RSA relies on the difficulty of factoring large numbers. $\phi(n)$ is the Euler totient function and is equal to the number of integers up to n that are relatively prime to it (two numbers a, b are relatively prime if $\gcd(a, b) = 1$). Obviously, $\phi(p) = p-1$ for prime numbers p (i.e. all except itself). It also turns out that for the product of two prime numbers p, q: $\phi(p \cdot q) = (p-1)(q-1)$.

Key Generation

The process to generate the public and private keypair is as follows:

1. Generate large primes p and q
2. Compute $n = pq$ and $\phi(n) = (p-1)(q-1)$
3. Select the **public exponent** $e \in [1, \phi(n) - 1]$ such that e is relatively prime to $\phi(n)$

4. Compute $d = e^{-1} \mod \phi(n)$

The **public key** is the tuple (n, e). The **private key** is the tuple (d, p, q).

Anyone who knows n, e cannot easily find $\phi(n)$ and thus can't compute d.

Encryption

A plaintext $m < n$ can be encrypted as $c = m^e \mod n$ using info from the public key.

Decryption

A ciphertext c can be decrypted as $m = c^d \mod n$.

Signature (RSA-DSA)

A message can be signed by computing $h(m)^d \mod n$ for some hash function h that outputs a suitably sized value. This can be verified by computing $s^e \mod n$ and comparing it with $h(m)$.

Computing d

We use the extended euclidean algorithm to find $d = e^{-1} \mod \phi(n)$ given e and $\phi(n)$. It gives us d, k such that $de + k\phi(n) = 1$ and this gives us d.

2.9 Digital Signature Standard (DSS)

This scheme only provides signatures. It has the following elements:

1. p a (large) prime number
2. q a prime factor of $(p-1)$
3. $g \in \mathbb{F}_p$ with $\text{ord}(g) = q$
4. h a cryptographic hash function

2.9.1 Keypair

Signer selects some $x \in [1, q-1]$ as the private key. $y = g^x$ is the public key.

2.9.2 Signature

The signer performs the following for each message to sign:

1. Randomly select $k \in [1, q-1]$
2. Compute $r = g^k$
3. Solve for s in the signing equation (Equation 2.9)

$$h(m) \equiv xr + ks \mod q \tag{2.9}$$

The signature is provided as the tuple (r, s). Note how steps 1 and 2 reflect the keygen process. Step 3 requires the usage of the extended euclidean algorithm.

2.9.3 Verification

The verifier does the following:

1. Compute $u = h(m)s^{-1} \mod q$ and $v = -rs^{-1} \mod q$
2. Compute $w = (g^u y^v \mod p) \mod q$

3. Accept if $w = r$

This works because of the following. Starting with Equation 2.9 (and recalling that $r = g^k$):

$$\begin{aligned} k &= h(m)s^{-1} - xrs^{-1} \\ g^k &= g^{h(m)s^{-1} - xrs^{-1}} \\ r &= g^u (g^x)^{-rs^{-1}} \\ r &= g^u y^v \end{aligned}$$

2.10 Elliptic Curve Cryptography

An elliptic curve is a set of points $(x, y) \in F$ that satisfy some cubic equation. We can define an addition operation on this set of points to form a **group**: to add two points P and Q draw a line between them and set $R = (x, -y)$ where (x, y) is the third point that intersects the curve (note that the y coordinate is flipped). If $P = Q$ then the line connecting them is just the tangent at P.

2.10.1 ECC Finite Fields

We restrict our attention to points where $x, y \in \mathbb{F}_p$. Given parameters (a, b, p) we can solve for all such points on the curve as follows:

1. Compute all the 'quadratic residues' in $\mathbb{F}_p$. That is, the set $QR = \{i | \exists y \in \mathbb{F}_p \text{ s.t. } i = y^2 \mod p\}$
2. For all $x \in \mathbb{F}_p$, compute $x^3 + ax + b$ and see if $x \in QR$. If so, then (x, y) is a point where $y^2 = x^3 + ax + b$.

The group formed by these points is **cyclic**, that is every element is a generator. Do note that there is technically also a 'point at infinity' denoted ∞ or $\mathcal{O}$ which is not a generator which acts as the additive identity.

2.10.2 Keypair

A curve and field is selected, along with an arbitrary generator point P. The **private key** is a random integer d where $0 < d < q$. The **public key** is $Q = dP$.

2.10.3 EC-DH

We can compute a shared diffie-hellman secret using $d_A Q_B = d_B Q_A = (d_A d_B)P$.

2.11 Hash-Chain Based Authentication

Hashing is much faster than verifying a signature in a public-key system. The problem we want to solve is how to authenticate n messages $x_0 ... x_{n-1}$ without having to sign each one.

2.11.1 Hash Chains

A hash chain is a building block that can be used to solve the authentication problem. In a hash chain, we select a value y and compute $k_i = h^{n-i}(y)$ for $i \in [0, n)$ (we start by computing k_{n-1}). Then we sign $k_0 = h^n(y)$ using a private key of some sort, i.e. $\text{Sig}_A(k_0)$. The hash chain is given by the tuple:

$$(\text{Sig}_A(k_0), k_0, ..., k_{n-1}) \tag{2.10}$$

Thus all we need to do to verify is first verify the signature $\text{Sig}_A(k_0)$. Then we know k_0 is correct. Then we can verify k_1 by computing $h(k_1)$ which should be equal to k_0 (assuming h is secure), and we can continue this

process. Thus we only require 1 signature verification and $n-1$ hashes to verify the whole chain, which is much better than n signature verifications.

2.11.2 Basic Hash Chain Message Authentication

The first approach to solving the authentication problem is as follows. Suppose the server is A and the receiver is B. B would like to receive messages $x_1...x_n$ and ensure they are authenticated. A starts by generating an $n+1$-length hash chain (**not** n). It then sends B the message $(k_0, \text{Sig}_A(k_0))$, which B can verify using A's public key. A then sends the message $(x_1, \text{MAC}(k_1, x_1))$, then sends k_1. B first hashes this latter value to check that $h(k_1) = k_0$ so it knows k_1 is the correct value. Then B can authenticate x_1 using $\text{MAC}(k_1, x_1)$. A then sends $(x_2, \text{MAX}(k_2, x_2))$ which can be verified with k_1, and so on and so forth. At any point, B only needs to hold on to k_0 and k_{i-1} to verify x_i.

2.11.3 Hash Chain One-Time Password Authentication

Another application of hash chains is for providing passwords to access some system. Instead of using the same long-term password, a user pre-generates a hash chain $k_0...k_n$ and stores k_0 on the remote system in some secure manner. On the i-th access, they can use k_i as the password (k_0 is never a password). The system will check $k_{i-1} = h(k_i)$ and accept the password if so. This is only good for $n-1$ accesses.

2.11.4 Merkle Trees

Merkle trees solve a slightly different problem. In this problem, there are n messages but they are not 'sequenced' - the system must be able to authenticate any randomly chosen message x_i. The messages may be generated by one party or by n different parties (each contributing one).

A Merkle tree is a complete binary tree where there are n leaves ordered left to right with 'labels' $h(x_i)$. Non-leaf nodes are labeled by $h(L||R)$ where L is the label of the left child and R the label of the right child ($||$ denotes concatenation). The root is called r.

An example (which seems to be the implicit one described in the lectures and textbook) is as follows. We have a **data owner** B who produces $(x_1, ..., x_n)$. They send this along with the full Merkle tree to a **server** A. They also publish and sign the root value. B wants **clients** C to be able to get any x_i they want from A and be assured that it is authentic (e.g. the x_i are database entries). When C requests to authenticate x_i, A sends the values on the authentication path (or *co-path*) for x_i which consists of all the siblings on the path from the root to $h(x_i)$. The client C can use these values to reconstruct the root value. If the root value it reconstructs is the same as the (signed and verified) value published by B, C can be assured that x_i is correct.

2.12 Blockchain

In a digital currency system, we can form **transactions** as tuples (pk_A, pk_B, amt) where users are identified by their public keys. Transactions are signed by the sender. If it is valid (signature and amount) according to the current state of the blockchain, the transaction will be appended to the blockchain.

To prevent double-spending, nodes are awarded for computing a *nonce* (number used once) such that the value in Equation 2.11 has some special bit pattern (e.g. k zeros at the start). Thus there is an incentive to verify the earliest valid transaction. The node that computes the nonce publishes it and is rewarded if it is correct and the blockchain is then updated.

$$h(h(B_{prev})||pk_A||pk_B||nonce) \tag{2.11}$$

Each block contains multiple transactions, and their contents can be validated using a Merkle tree. The storage of nodes in the tree is distributed.

Chapter 3

Network and Wireless Security

3.1 Challenge-Response Authentication

The goal of authentication is for a party A to ensure that the party it is communicating with (also called the *peer*) is B and not someone else. The fundamental building block of authentication protocols discussed in this course is the challenge-response method. It can be used with both symmetric and asymmetric (public + private) key systems.

In the symmetric case, parties A and B share a key K_{AB}. For A to verify that its peer is B, it issues a random message Ch called the **challenge**. The peer must compute $\text{MAC}(K_{AB}, Ch)$ - for a secure MAC only parties that know K_{AB} (which, by assumption, is only A and B) can do this. A can then check the result against its own computation of $\text{MAC}(K_{AB}, Ch)$ and, if it is correct, trust the peer to be B. The challenge must be randomly generated to prevent a replay attack.

In the asymmetric case, A knows B's public key. A sends a challenge Ch to the peer, and the peer must compute $Sig_{S_B}(Ch)$ which can only be computed using B's private key S_B. A can then use the known public key to verify the signature and trust the peer.

To unify the two cases with notation, $[\vec{U}]_B$ will be used to denote an authenticated message where:

$$\vec{U} = (U_1, ..., U_n)$$

In the symmetric case, $[\vec{U}]_B = \text{MAC}(K_B, U_1, ..., U_n)$ where K_B is the symmetric key shared with the receiving party. In the asymmetric case, $[\vec{U}]_B = Sig_{S_B}(U_1, ..., U_n)$.

3.2 Public Key Infrastructure (PKI)

The problem with asymmetric key systems (e.g. RSA) is that anyone can generate a valid public-private keypair. We need some way for a party A to ensure that a given public key is actually the public key of an entity B and not one generated by the adversary E.

This can be accomplished by having a **Certificate Authority** (CA) whose public key is known (usually installed on the device). The CA can create **certificates** which essentially are signatures over the tuple (PK_B, ID_B) where PK_B is the public key of B and ID_B is some string corresponding to the identity of B. The signature can be verified using the known public key of the CA.

It would be infeasible to have one universal CA that signs everyone's public keys. Instead, a **root CA** creates certificates for subordinate CAs, which in turn may sign public keys or other CAs. This forms a **certificate chain** for any given public key. To verify a certificate chain, we first verify the 'lowest level' certificate (i.e. a direct signature of an entity's public key by a CA). We then verify the certificate of the lowest level CA, then the next CA, and so on until we reach the root CA whose public key is known.

3.3 Mutual Authentication and Key Establishment

This section covers some general concepts that are used throughout the rest of this chapter. The goal of this chapter is to design Authentication and Key Establishment (AKE) protocols to generate keys for all the parties and ensure that they are authenticated and then use techniques from chapter 2 to secure communications.

A key establishment protocol allows two or more parties to decide on cryptographic keys to use in securing a communication channel between them. There are two types of keys that will be used in key establishment protocols:

- Long-term keys: keys that exist before the protocol
- Session keys: keys that are established as a result of the protocol

For simplicity, we can assume that the goal of a protocol is to establish just one session key - one can use a **Key Derivation Function** (KDF) to generate more keys when needed. There are two main types of AKE protocols:

- Key **transport**: one party generates the session key which is then transferred to the other parties
- Key **agreement**: the session key is a function of inputs from all parties

We may also have a method for **key confirmation** to ensure that two parties are indeed using the correct key.

3.4 Internet Key Exchange (IKE)

IKE is used to establish **Security Associations** (SA) between two IP hosts before IPSec communications begin. A security association includes a set of cryptographic algorithms each of which can be one of the following:

1. A Diffie-Hellman group ($\mathbb{F}$, elliptic curve, etc.)
2. A pseudorandom function (PRF)
3. An integrity protection mechanism (e.g. a MAC)
4. An encryption algorithm

It also contains the data and/or parameters used in each of these algorithms (keys, elliptic curve params, etc.).

A simplistic version of IKE was given in the lectures under the name 'Protocol C'. It operates as follows. Assume that there are two parties A and B that wish to agree on keys and that **ciphersuite negotiation** has already occurred and that the chosen Diffie-Hellman group was $\mathbb{F}_p$.

1. A sends an *initiation* message containing
   - R_A a random value which allows for confirmation of the key that will be generated
   - g^a its public key to be used in DH key establishment
2. B may optionally verify A's public key using a certificate. B does the following:
   - Generates a random value R_B to be used for key confirmation
   - Computes $\alpha = (g^a)^b$ as the shared DH secret, and a key $K_\alpha = \text{KDF}(\alpha)$ from it. We may not be able to use α as the key directly since it may not be large enough so KDF is often a PSRG that generates enough bits for a key using α
3. B then sends a response containing
   - ID_B containing its identity so that A may authenticate it
   - R_B
   - $Sig_b(R_B, R_A, g^a, g^b)$ which authenticates B as the owner of the keypair (b, g^b). It prevents a replay attack by using R_B and R_A.
   - $\text{MAC}(K_\alpha, R_A, ID_B)$ which confirms the key K_α and prevents a replay attack using R_A.
4. A then authenticates B's identity using a CA. Then it computes K_α using B's public key, verifies the signature in B's response and confirms the key using the MAC from B's response. Finally, it sends a response containing
   - ID_A containing its identity (only sent now to protect privacy if A is a user)
   - $Sig_a(R_B, R_A, g^a, g^b)$ which authenticates A as the owner of the keypair (a, g^a) and prevents a replay attack by using R_B and R_A

- MAC(K_α, R_B, ID_B) which confirms the key K_α and prevents a replay attack using R_B.

3.5 IPSec

IPSec enables the protection of IP packets. There are two modes. In **transport** mode, the SA established by IKE corresponds to the sender and receiver in the IP packet - routing is done using the IP header which is then stripped and the IPSec header/payload is processed by the receiver. In **tunnel** mode, the SA established by IKE corresponds to two nodes that sit between the actual sender and receiver (e.g. security gateways in an enterprise network) - routing is done normally except the IPSec header/payload is inserted at the inner two nodes.

If IPSec is used, the IP headers also include some data known as a **Security Parameter Index** (SPI) to indicate which SA should be used to process the packet.

IPSec consists of two mechanisms: an **Authentication Header** (AH) and **Encapsulating Security Protocol** (ESP). Either one may be used, or even both.

3.5.1 Authentication Header (AH)

The authentication header only provides message integrity and anti-replay. It does so by computing an **Integrity Check Value** (ICV) using a MAC over fields in the IP packet. The shared key to use in the MAC was established by IKE. Since some fields in an IP packet (e.g. TTL) may change as they are routed, the ICV is only computed over *immutable* fields in the packet. This value is then inserted between the IP header and payload. A monotonically increasing sequence number is used to prevent replay attacks.

3.5.2 Encapsulating Security Protocol (ESP)

ESP can provide both confidentiality and integrity. The correct way to do this is to *encrypt then authenticate*. The payload of the IP packet is encrypted, and an ICV is computed over the payload plus the SPI and sequence number. There may also be an initialization vector (IV) value if it is required by the encryption algorithm. The ICV is appended to the end of the encrypted data.

3.6 IKE v2

3.7 TLS

Chapter 4

Web Security