Chapter 1

Topic 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

```
void main() {
        char line[10];
        gets(line);
}
```

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

Topic 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 → 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

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 Diffie-Hellman Key Exchange

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.