Modern Cryptography

Design, analysis and implementation of mathematical techniques for securing information, systems and computation against adversarial attacks.
If you don’t understand what you want to achieve, how can you possibly know when (or if) you have achieved it?
Modern Cryptography

Importance of clear assumptions:
• allows researchers to validate assumptions
• comparison between schemes based on different assumptions
• re-adjust for weaknesses in assumptions

any new cryptographic construction should be proven secure with respect to a specific definition, and a set of clearly stated assumptions
Scenarios and Goals

**Confidentiality**
Keep others from reading Alice’s messages/data

**Integrity**
Keep others from undetectably tampering with Alice’s messages/data

**Authenticity**
Keep others from undetectably impersonating Alice (keep her to her word too!)
Cryptography: Terms

Encryption (E): The process of transforming a message so that its meaning is not obvious
Decryption (D): The process of transforming an encrypted message back into its original form.
Plaintext (m): Original, unencrypted form of a message
Ciphertext (p): The encrypted form of a message

Formal Notation: We seek a cryptosystem for which P = D (E (P))
Ceasar Cipher: Substitution Cipher

Plaintext letters replaced with letters fixed shift way in the alphabet.

Example:
- Plaintext: HEY BRUTUS BRING A KNIFE TO THE PARTY.
- Ciphertext: KHB EUXWXV EULQJ D NQLIH WR WKH SDUWB
- Key Shift 3:
  - ABCDEFGHIJKLMNOPQRSTUVWXYZ
  - DEFGHIJKLMNOPQRSTUVWXYZABC

- Encryption and Decryption are symmetric.
- Key space?
  - 26
- Attack shift ciphers?
  - brute force
Monoalphabetic Substitution Cipher

• What is the key space?
  • 26

• Launching an attack?
  • frequency analysis: the study of frequency of letters or groups of letters (grams).
  • Common letters: T, A, E, I, O
  • Common 2-letter combinations (bi-grams): TH, HE, IN, ER
  • Common 3-letter combinations (tri-grams): THE, AND, ING.
Kerckhoffs’ principle

Encryption and Decryption and Key Generation Algorithm are publicly known. The only unknown is the shared secret key.
Ceasar Cipher: Substitution Cipher

Plaintext letters replaced with letters fixed shift way in the alphabet.

Lessons Learnt?

- Simple, exhaustive key search can be effective
- Key space needs to be large enough to prevent attack
- Use different substitutions to prevent frequency analysis
Vigenère Cipher (1596)

- Main weakness of monoalphabetic substitution ciphers:
  - Each letter in the ciphertext corresponds to only one letter in the plaintext

- Polyalphabetic substitution cipher
  - Given a key $K = (k_1, k_2, ..., k_m)$
  - Shift each letter $p$ in the plaintext by $k_i$, where $i$ is modulo $m$

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |

Plaintext: CRYPTOGRAPHY
Key: LUCK LUCKLUCK (Shift 11 20 2 10 11 20 2 11 ...)
Ciphertext: NLAZEIIBLJJI
Kasisky Test Index of coincidence

- Repeating patterns (of length >2) in ciphertext are a tell
  - Likely due to repeated plaintext encrypted under repeated key characters
  - The distance is likely to be a multiple of the key length
Vigenère Cipher (1596)

• Lessons learnt?
  • As key length increases, letter frequency becomes more random
  • If key never repeated, Vigenère wouldn’t be breakable!

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |

Plaintext  CRYPTOGRAPHY
Key        LUCK LUCKLUCK (Shift 11 20 2 10 11 20 2 11 ...)
Ciphertext NLAZEIIIBLJIJI
One Time Pad (1920s)

- Fix vulnerability in Vigenère cipher: use very long keys
- Key is a random string: \textit{at least as long as the plaintext}
- Plaintext: Message that is $n$ bits long
- Key: $\{0, 1\}^n$ sequence of $n$ bits chosen uniformly at random.
The XOR operator takes two bits and outputs one bit

| \(0 \oplus 0 = 0\) | \(0 \oplus 1 = 1\) | \(1 \oplus 0 = 1\) | \(1 \oplus 1 = 0\) |

Useful properties of XOR

| \(x \oplus 0 = x\) | \(x \oplus x = 0\) | \(x \oplus y = y \oplus x\) | \((x \oplus y) \oplus z = x \oplus (y \oplus z)\) | \((x \oplus y) \oplus x = y\) |
Review: XOR Algebra

Algebra works on XOR too

<table>
<thead>
<tr>
<th>$y \oplus 1 = 0$</th>
<th>Goal: Solve for $y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y \oplus 1 \oplus 1 = 0 \oplus 1$</td>
<td>XOR both sides by 1</td>
</tr>
<tr>
<td>$y = 1$</td>
<td>Simplify with identities</td>
</tr>
</tbody>
</table>
One-Time Pads: Key Generation

| Alice | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

The key $K$ is a randomly-chosen bitstring.

Recall: We are in the symmetric-key setting, so we’ll assume Alice and Bob both know this key.
### One-Time Pads: Encryption

The plaintext $M$ is the bitstring that Alice wants to encrypt. Idea: Use XOR to scramble up $M$ with the bits of $K$. 

<table>
<thead>
<tr>
<th>Alice</th>
<th>$K$</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
One-Time Pads: Encryption

Encryption algorithm: XOR each bit of $K$ with the matching bit in $M$.

The ciphertext $C$ is the encrypted bitstring that Alice sends to Bob over the insecure channel.
One-Time Pads: Decryption

Bob receives the ciphertext $C$. Bob knows the key $K$. How does Bob recover $M$?
# One-Time Pads: Decryption

The decryption algorithm for One-Time Pads is to XOR each bit of the key $K$ with the matching bit in the ciphertext $C$. This can be represented as $M = K \oplus C$.

<table>
<thead>
<tr>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$C$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$M$</td>
</tr>
</tbody>
</table>

Decryption algorithm: XOR each bit of $K$ with the matching bit in $C$. 

any new cryptographic construction should be proven secure with respect to a specific definition, and a set of clearly stated assumptions
Threat Models

- **Ciphertext-only attack**: An attacker (Eve) observes ciphertext and nothing else
  - Can Eve observe more than one ciphertext?
  - this distinction can make a big difference!
Threat Models

• **Known-Plaintext Attack**: An attacker (Eve) observes ciphertext and knows underlying plaintext
  • e.g., Alice: plaintext: Hello! ciphertext: 23asdf1941
  • Bob: plaintext: Hello! ciphertext: 23asdf1941
Threat Models

- **Chosen-Plaintext Attack:**
  - Observe one or more ciphertext, where plaintext is unknown
  - Also observe ciphertext for plaintext of attacker’s choosing.

Attacker’s chosen plaintext: **test attack**

observed ciphertext **asdlkjwery**
Threat Models

- **Chosen-Ciphertext Attack:**
  - Attacker is able to get the parties to decrypt certain cipher texts of that attacker's choice.

```
Attacker's chosen ciphertext: asdlkjwery
observed plaintext: top secret attack
```
A Perfectly Secure Encryption Scheme

Regardless of any prior information the attacker has about the plaintext, the ciphertext observed by the attacker should leak no additional information about the plaintext.

Alice can only observe one ciphertext going over the network.
A Secure Encryption Scheme

An encryption scheme given by: (key gen alg., encryption alg, decryption alg.) over message space $M$ is perfectly secure iff

$\forall$ probability distribution over $M$

$\forall$ message $m \in M$

$\forall$ ciphertext $c \in C$ for which $\Pr[C = c] > 0$

we have

$\Pr[M = m \mid C = c] = \Pr[M = m]$
One Time Pad: Perfectly Secure?

• OTP achieves Perfect Secrecy
  • Shannon or Information Theoretic Security
  • Basic idea: ciphertext reveals no “additional information” about plaintext
Proving Perfect Security: One Time Pads

Problem Statement:
• Suppose Alice has sent one of two messages $M_0$ or $M_1$, and Eve has no idea which was sent.
• Eve tries to guess which was sent by looking at the ciphertext.

To Show:
• Eve’s probability of guessing correctly is $\frac{1}{2}$
• This is no different than it would be if she had not intercepted the ciphertext at all.
Proving Perfect Security: One Time Pads

Alice randomly chooses a bit string $\in \{0, 1\}^n$, and Alice sends the encryption of $M_b$.
If Eve observes that the ciphertext has some specific value $C$, what is the conditional probability that $b = \text{the input bit string given her observation}$?

Fix arbitrary distribution over $M = \{0, 1\}^n$, and arbitrary $m, c \in \{0, 1\}^n$

$$Pr[M = m \mid C = c] = ?$$
$$= Pr[C = c \mid M = m] \cdot Pr[M = m] / Pr[C = c]$$

$$Pr[M = m \mid C = c] = ?$$
$$= Pr[C = c \mid M = m] \cdot Pr[M = m] / Pr[C = c]$$
$$= Pr[K = m \oplus c] \cdot Pr[M = m] / 2^n$$
$$= 2^{-n} \cdot Pr[M = m]$$
$$= Pr[M = m]$$

$$Pr[C = c]$$
$$= \Sigma_{m'} Pr[C = c \mid M = m'] \cdot Pr[M = m']$$
$$= \Sigma_{m'} Pr[K = m' \oplus c] \cdot Pr[M = m']$$
$$= \Sigma_{m'} 2^{-n} \cdot Pr[M = m']$$
$$= 2^{-n}$$
One Time Pad: Limitations

- The key is as long as the message
- Only secure if each key is used to encrypt a single message
- Parties must share keys of (total) length = the (total) length of all the messages they might ever send!
Using the same key twice?

Say \( c_1 = k \oplus m_1 \)
\( c_2 = k \oplus m_2 \)

Attacker can compute
\[
\begin{align*}
    c_1 \oplus c_2 & = (k \oplus m_1) \oplus (k \oplus m_2) \\
                     & = m_1 \oplus m_2
\end{align*}
\]

This leaks information about \( m_1, m_2 \)!
Limitations of Perfect Security

Regardless of any prior information the attacker has about the plaintext the ciphertext observed by the attacker should **leak no additional information** about the plaintext.

- The key is as long as the message
- Only secure if each key is used to encrypt a single message

*Limitations are not only of One Time Pads, but inherent to any perfectly secure encryption scheme.*

*Assumes the attacker as unlimited computational power*
Computational Security/Secrecy

Would be okay if a scheme leaked information with a tiny probability to eavesdroppers with bounded computational resources.

- Allowing security to fail with a tiny probability (negligible in key length n)
  - how tiny is tiny? $2^{-60}$: probability of an event occurring every 100 billion years!
- Only consider efficient attackers (bounded in polynomial time by key length)
  - attackers that can brute-force the key space in bounded time.
  - try testing $2^{112}$ keys? Would take a supercomputer since Big Bang!
  - modern key space? $2^{128}$ or more!
Computational Secrecy: One Time Pads

Key Insight: Randomness –

• something an adversary won’t know, can’t predict and can’t figure out.
Randomness

Explicit Uses of Randomness:
• Generate secret cryptographic keys
• Generate random initialization vectors or nonces for encryption

Use cases
• Generate passwords for new users
• Shuffle the order of votes in an electronic voting machine
• Shuffle cards etc. (for online games)
What does “random” mean?

• What does “uniform” mean?

• Which of the following is a uniform string?
  • 0101010101010101
  • 0010111011100110
  • 0000000000000000

• If we generate a uniform string, each of the above occurs with probability $2^{-16}$
What does “random” mean?

- “Randomness” is not a property of a string, but a property of a distribution.

- The uniform distribution on $n$-bit strings is the distribution $U_n$ where
  - $U_n(x) = 2^{-n}$ for all $x \in \{0,1\}^n$
What does “pseudorandom” mean?

- Informal: cannot be distinguished from uniform (i.e., random)
- Which of the following is pseudorandom?
  - 0101010101010101
  - 0010111011100110
  - 0000000000000000
- Pseudorandomness is a property of a distribution
How Random is “Random”?

How a months-old AMD microcode bug destroyed my weekend [UPDATED]

AMD shipped Ryzen 3000 with a serious microcode bug in its random number generator.

JIM SALTER - 10/29/2019, 7:00 AM
C’s built-in rand() function

```c
unsigned long int next = 1;
/* rand: return pseudo-random integer on 0…32767 */
int rand(void){
    next = next * 11 - 3515245 + 12345;
    return (unsigned int) (next/65536) % 32768;
}
/* srand: set seed for rand() */
void srand(unsigned int seed){
    next = seed;
}
```

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” -- John von Neumann
Random-number generation

• Two steps:
  1. Continually collect a “pool” of high-entropy (i.e., “unpredictable”) data from external inputs
     1. Delays between network events
     2. Hard-disk access times
     3. Keystroke/mouse movements
  2. When random bits are requested, process this data to generate a sequence of uniform, independent bits/bytes
     • May “block” if insufficient entropy available

• Other... – Hardware random-number generator (e.g., Intel)
How might we get “good” random numbers?

• For security applications, want “cryptographically secure pseudorandom numbers”
• Libraries include cryptographically secure pseudorandom number generators (CSPRNG)
  • Linux:
    • /dev/random: blocking: waits for enough entropy
    • /dev/urandom: nonblocking, possibly less entropy
    • getrandom() – syscall! – by default blocking
  • Internally:
    • Entropy pool: gathered from multiple sources
      • e.g.: mouse/keyboard/network timings
  • Better idea:
    • AMD/Intel’s on-chip random number generator: RDRAND
    • Hopefully no hardware bugs!
Random-number generation

Request random bits

Processing/smoothing

Request random bits
Pseudorandom (number) generators: PRG/PRNGs

• A PRG is an efficient, deterministic algorithm that expands a short, uniform seed into a longer, pseudorandom output
  • Useful whenever you have a “small” number of true random bits, and want lots of “random looking” bits

```
seed

G

output
```

seed: a small number of true random bits

G: deterministic polynomial time algorithm

output: pseudorandom bits of length n, i.e., cannot be distinguished from truly random bits, by any efficient statistical test.
Do PRNGs exist?

• We actually don’t know!
• Assume that there exist *some* functions \( G \) that are PRNG.

![Diagram]

<table>
<thead>
<tr>
<th>seed</th>
<th>( G )</th>
<th>output</th>
</tr>
</thead>
</table>

seed: a small number of true random bits

\( G \): deterministic polynomial time algorithm

output: pseudorandom bits of length \( n \), i.e., cannot be distinguished from truly random bits, by any efficient statistical test.
Applying Pseudo-randomness to the one-time pad

- n bits
- key
- G
- p bits
- “pseudo” key
- p bits
- message +
- ciphertext
Single-message secrecy

Message $m$  \hspace{1cm} Ciphertext (c) \hspace{1cm} Original $m$
Multiple message secrecy

We are not going to formally define a notion of multiple-message secrecy
• Instead, define something stronger: security against chosen-plaintext attacks (CPA-security)
• minimal notion of security an encryption scheme should satisfy
Security against Chosen Plaintext Attack: Impossible?

It really is a problem if an attacker can tell when the same message is encrypted twice!

*This attack only works if encryption is deterministic!*
Random Functions

• Functions map from some set $X$ to a set $F(X) = Y$.
  • (think of this mapping as a hash table mapping from $x \rightarrow y$)

• $\text{Func}_n$: all mappings from $X: \{0, 1\}^n \rightarrow F(X) = Y: \{0, 1\}^n$
  • i.e., for all input bit strings of length $n$, there is a mapping to an output bit string also of length $n$
  • all possible mappings? $2^n(2^n)! \approx$ astronomically large!
Random Functions

Out of all possible functions between $X$ and $Y$ we choose one uniformly at random.

- e.g. for a 2 bit string mappings between $X$: $\{0, 1\}^2$ and $Y$: $\{0, 1\}^2$
- one possible mapping that we could choose:

<table>
<thead>
<tr>
<th>$x$</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F(x)$</td>
<td>01</td>
<td>11</td>
<td>00</td>
<td>10</td>
</tr>
</tbody>
</table>

Properties of function $F(X)$ chosen uniformly at random:

- for any given $x \in X$, the probability that $F(x) = y$ is $1/2^n$
- in our example example:
  - given $x \in X$, the probability that $F(x) = 1/2^2 = 1/4 = 0.25$
- $F(x)$ property:
  - if $x$ changes by one bit to give $x'$ then
  - $F(x')$ is completely independent of $F(x)$. 
Random Permutations

• Variant of random function is random permutation
  • treat them equivalently for our purposes.

• E.g.: random permutation over bit strings of length 2
  Encryption: \(\{0, 1\}^2 \rightarrow \{0, 1\}^2\)

<table>
<thead>
<tr>
<th>x</th>
<th>F(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Important Property of the Random Permutation:

A permutation is invertible (bijective) function

Given \(F(x)\) it is impossible to determine \(x\) without resorting to a brute force attack.

If \(|X|\) is very large? brute force not possible by an efficient (probabilistic polynomial time) attacker.
What we have, ideally: Random Functions

Consider the set of all permutations $F_k : X \to X$

If you know $k$, then $F_k(x)$ is trivial to invert

If you don’t know $k$, then $F_k(x)$ is one-way

most efficient attack is a brute force attack.

Think of $X$ as all 128-bit bit strings
What we have, ideally: Random Functions

Consider the set of all permutations $F_k: X \rightarrow X$

Think of $X$ as all 128-bit bit strings

Without knowing $k$, Eve learns nothing about $m$
What we have, ideally: Random Functions

In essence, this protocol is saying “Let’s use the kth permutation function”

Infeasible to store all permutation functions – so instead cryptographers construct pseudorandom functions
In essence, this protocol is saying “Let’s use the kth permutation function”

Infeasible to store all permutation functions – so instead cryptographers construct pseudorandom functions