Week 3: Lecture A
Improved Cipher Designs
Tuesday, September 5, 2023
Announcements

- **Project 1: Crypto** released
  - **Deadline:** Thursday, September 21st by 11:59PM

---

**Project 1: Cryptography**

**Deadline:** Thursday, September 21 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of at most two and submit one project per team. If you have difficulties forming a team, post on Piazza's Search for Teammates forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). *Don’t risk your grade and degree by cheating!*

Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any external dependencies. Use only default Python 3 libraries and/or modules we provide you.

**Helpful Resources**
- The CS 4440 Course Wiki
- VM Setup and Troubleshooting
- Terminal Cheat Sheet
- Python 3 Cheat Sheet
- PyMDS Module Documentation
- PyRoots Module Documentation

**Table of Contents:**
- Helpful Resources
- Introduction
- Objectives
- Part 1: Hash Collisions
  - Prelude: Collisions
  - Prelude: FastColl
  - Collision Attack
  - What to Submit
- Part 2: Length Extension
  - Prelude: Merkle-Damgård
  - Length Extension Attacks
  - What to Submit
- Part 3: Cryptanalysis
  - Prelude: Ciphers
  - Cryptanalysis Attack
  - Extra Credit
  - What to Submit
- Part 4: Signature Forgery
  - Prelude: RSA Signatures
  - Prelude: Bleichenbacher
  - Forgery Attacks
  - What to Submit
- Submission Instructions
Project Tips

- Projects are challenging—**you’re performing real-world attacks!**
  - Build off of lecture concepts
  - Make sure you understand the lectures
  - Prepare you to defend in the real world

- **Suggested strategy:** *get high-level idea down, then start implementing*
  1. Go through assignment and start sketching-out your approach
  2. **Come to Office Hours and ask if you’re on the right track!**
  3. Then start building your program

- Don’t get discouraged—**we are here to help!**
  - Most issues are cleared up in a few minutes of white-boarding
See Discord for meeting info!

www.utahsec.com
Questions?
Last time on CS 4440...

Message Confidentiality
Substitution Ciphers
Frequency Cryptanalysis
Message Confidentiality

- Confidentiality: ???

Alice → Mallory → Bob
- **Confidentiality**: ensure that only *trusted parties* can read the message
- Terminology: ???
Message Confidentiality

- **Confidentiality:** ensure that only trusted parties can read the message

**Terminology:**
- \( p \) plaintext: original, readable message
- \( c \) ciphertext: transmitted, unreadable message
- \( k \) secret key: known only to Alice and Bob; facilitates \( p \rightarrow c \) and \( c \rightarrow p \)
- \( E \) encryption function: \( E(p, k) \rightarrow c \)
- \( D \) decryption function: \( D(c, k) \rightarrow p \)
Confidentiality via Ciphers

- We define a key as ???
Confidentiality via Ciphers

- We define a key as a set of shifts
- Each shift represented by a letter
  - Relative position in the alphabet
We define a key as a set of **shifts**

Each shift represented by a **letter**
- Relative position in the alphabet

Shift goes past end of alphabet?

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

???
Confidentiality via Ciphers

- We define a key as a set of **shifts**
- Each shift represented by a **letter**
  - Relative position in the alphabet
- Shift goes past end of alphabet?
  - **Wrap around** to beginning!

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
</tbody>
</table>
Caesar Ciphers

- Really old school cryptography
  - First recorded use: Julius Caesar (100–144 B.C.)
- Replaces each plaintext letter with ???
Caesar Ciphers

- Really old school cryptography
  - First recorded use: Julius Caesar (100–144 B.C.)

- Replaces each plaintext letter with one a fixed number of places down the alphabet
  - Encryption: \( c_i := (p_i + k) \mod 26 \)
  - Decryption: \( p_i := (c_i - k) \mod 26 \)

- Example for \( k = 3 \):
  - Plain: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
  - +Shift: `33333333333333333333333333`
  - =Cipher: `DEFGHIJKLMNOPQRSTUVWXYZABC`
  
  - Plain: `go utes beat wash st`
  - +Key: `33 3333 3333 3333 33`
  - =Cipher: `jr xwhv ehdw zdvk vw`
Vigènere Ciphers

- First described by Bellaso in 1553
  - Later misattributed to Vigènere
- Encrypts successive letters via ???
Stefan Nagy

Vigènere Ciphers

- First described by Bellaso in 1553
  - Later misattributed to Vigènere

- Encrypts successive letters via **sequence of Caesar ciphers** determined by the letters of a keyword

- For an $n$-letter keyword $k$ ...
  - Encryption: $c_i := (p_i + k_{i \mod n}) \mod 26$
  - Decryption: $p_i := (c_i - k_{i \mod n}) \mod 26$

- Example for $k = \text{ABC}$ (i.e., $k_0 = 0$, $k_1 = 1$, $k_2 = 2$)
  - Plain:  $\text{bbbbbb amazon}$
  - +Key:  $012012 012012$
  - =Cipher: $\text{bcdbcd anczpp}$
Cryptanalysis

Brute-forcing every possible key
**Problem:** How can we beat **brute forcing** to break a substitution cipher?
**Cryptanalysis**

- **Problem:** How can we beat *brute forcing* to break a substitution cipher?
- **Observation:** Simple substitution ciphers don’t alter symbol frequency

[Letter frequency chart for the English language]

[Ordered by frequency chart]

Stefan Nagy
Vigènere Cipher Cryptanalysis

- Figure out how to **simplify a Vigenere cipher into a Caesar cipher**

  - Break it down into groups of letters—grouped by column (i.e., key-shift position)

```
Cipher: DEFGHIJKLMNOPQRSTUVWXYZABC
Shift: #################################################################

Cipher: BCDEFGHIJKLMNOPQRSTUVWXYZA
Shift: #################################################################

Cipher: EFGHIJKLMNOPQRSTUVWXYZABCD
Shift: #################################################################
```
Figure out how to simplify a Vigenère cipher into a Caesar cipher:
- Break it down into groups of letters—grouped by column (i.e., key-shift position).
- Then, use frequency analysis to derive the key (shift) for each letter-column.

Cipher: DEFGHIJKLMNOPQRSTUVWXYZABC
- Shift: 33333333333333333333333333
= Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Cipher: BCDEFGHIJKLMNOPQRSTUVWXYZA
- Shift: 11111111111111111111111111
= Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Cipher: EFGHIJKLMNOPQRSTUVWXYZABCD
- Shift: 44444444444444444444444444
= Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Vigènere Cipher Cryptanalysis
Finding Key Size via Kasiski Method

- **Example:**

<table>
<thead>
<tr>
<th>p</th>
<th>THERE ARE TWO WAYS OF CONSTRUCTING SOFTWARE ARE DESIGNING NEW</th>
<th>c</th>
<th>LFWKI MJCLP ISISK HJOGL KMVGU RAGKM KMXMA MJCVX WUYLGI ISW</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>ISTOM AKEIT SOSIME PLETH AT THE REAL</td>
<td>c</td>
<td>ALXAE YCXMJ KMKBQ BDCLA EFLFW KIMJC GUZUG SKECZ GBWYM OACFV</td>
</tr>
<tr>
<td>p</td>
<td>IE SAN DETHEO THERW AT IMAVIE ITSOC OMLPLI CATED THAT HERE</td>
<td>c</td>
<td>MQKYF WXTWM LADOI YQBFWE GKSDFI ULGQV SYHJA VEFWEB LAEFL FWKIM</td>
</tr>
<tr>
<td>p</td>
<td>RENO OBVIOS SDEFI CIENC IESTH EFIRS TMETH ODISF ARMOR EDIFF</td>
<td>c</td>
<td>JCFHS NNGGN WPWDA VMQFA AXWFZ CXBVE LWML AVGKY EDEMJ XHUXD</td>
</tr>
</tbody>
</table>
Finding Key Size via Kasiski Method

- Pick **realistic key lengths**; a length of two or three is probably short

<table>
<thead>
<tr>
<th>Dist.</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finding Key Size via Kasiski Method

- Then, **group letters by columns**—they received equal shifts!
Chi-square on Reverse-shifted Column Strings

Column #1 String (with a zero shift): LJSGUJKYSEKDLJGGAKWOGHLMJNLJVE}

\[
\chi^2 = \sum_{i=1}^{N} \frac{(O_i - E_i)^2}{E_i}
\]

1. \( O_L \) = observed count for letter ‘L’ = 5.0
2. \( E_L \) = expected count for letter ‘L’
   \[ E_L = \text{EnglishFreq}_L \times \text{ColumnStringLength} \]
   \[ = 0.04025 \times 34 \]
   \[ = 1.3685 \]
3. \( \chi^2 = \frac{(5.0 - 1.3685)^2}{1.3685} \]
   \[ = 9.6367 \]
4. \textbf{Lowest score = the correct shift!}
## Chi-square on Reverse-shifted Column Strings

**Column #1 String (with a zero shift):**

- A: IYMBWXIXIH: 291.39
- B: HXLAVWHWHG: 107.28
- C: GWKZUVGVGF: 236.00
- D: FVJYTUFUFE: 127.44
- E: EUIXSTETED: 77.16
- F: DTHWRSDSC: 29.73
- G: CSGVQRCRCB: 157.77
- H: BRFUPQPBQA: 487.57
- I: AQETOPAPAZ: 265.38
- J: ZPDSNOZ0ZY: 1227.21
- K: YOCRMINYX: 118.94
- L: XNBQLMXMXW: 726.79
- M: WMAPKLWLWV: 71.82

**Column #2 String (with a reverse shift in Column #1):**

- N: VLZOJKVKVU: 341.77
- O: UKYNIJUJUT: 306.11
- P: TJXMHITITS: 145.08
- Q: SIWLGHSHSR: 25.58
- R: RHVKGFRGRQ: 159.45
- S: QGUJEFQFQP: 1035.24
- T: PFTIDEPEPO: 50.52

**Chi-square Calculation:**

\[
\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}
\]

- **O_L** = observed count for letter 'L' = \(5.0\)
- **E_L** = expected count for letter 'L' = \(0.04025 \times 34\) = 1.3685

\[
\chi^2 = \frac{(5.0 - 1.3685)^2}{1.3685} = 9.6367
\]

Repeat for all other letters. Sum = \(\chi^2\) score for that shift. Repeat for the 25 other shifts. Lowest score = the correct shift!
Recap: Breaking Vigènere

1. ????
Recap: Breaking Vigènere

1. Identify the key length:
   - **Project 1**: keys will always be of length *eight*
   - **Extra Credit**: key varies—use **Kasiski method**!

2. ???
1. Identify the key length:
   - **Project 1:** keys will always be of length **eight**
   - **Extra Credit:** key varies—use **Kasiski method**!

2. Divide ciphertext into $N$ columns:
   - **Why?** Because Vigènere uses a repeating key
   - Vigènere cipher is a set of $N$ Caesar ciphers

3. ????
Recap: Breaking Vigènere

1. Identify the key length:
   - **Project 1:** keys will always be of length **eight**
   - **Extra Credit:** key varies—use **Kasiski method**!

2. Divide ciphertext into \( N \) columns:
   - **Why?** Because Vigènere uses a repeating key
   - Vigènere cipher is a set of \( N \) Caesar ciphers

3. Perform cryptanalysis on each column:
   - **Chi-square test:** find **best-fit shift** per column
   - Best-fit shift = that column’s **key** letter
   - Assemble all \( N \) column keys to get key word!
Questions?
This time on CS 4440...

Pseudo-random Keys
One-time Pads
Transposition Ciphers
Block Ciphers
Pseudo-random Keys
Recap: Confidentiality via Substitution Ciphers

- Clearly, **simple substitution ciphers** are vulnerable to frequency analysis
  - Root cause: ???

Ordered by frequency
Recap: Confidentiality via Substitution Ciphers

- Clearly, **simple substitution ciphers** are vulnerable to frequency analysis
  - **Root cause:** the key length is much smaller than the plaintext length
Recap: Confidentiality via Substitution Ciphers

- Clearly, **simple substitution ciphers** are vulnerable to frequency analysis
  - **Root cause:** the key length is **much smaller** than the plaintext length

How can we create a **better key** to improve **confidentiality**?

Ordered by frequency
How long should an ideal cipher key be?

- Half the size of the plaintext: 0%
- As long as the plaintext: 0%
- None of the above: 0%
Generating Keys

- Functions: ???
Generating Keys

- **Functions**: takes input and generates output
  - E.g., Hash functions
  - E.g., HMAC functions

- **Generators**: ???

```
“I really love CS 4440!”
```

```
a6be04fc96f03c1f45961259b0793a13
```
Generating Keys

- **Functions**: takes input and generates output
  - E.g., Hash functions
  - E.g., HMAC functions

- **Generators**: produces output out of thin air
  - E.g., number generators
  - E.g., HMAC secret keys
An ideal key is random...
What are some physical sources of randomness?

Nobody has responded yet.

Hang tight! Responses are coming in.
Generating Random Keys

- **Physical randomness:**
  - Coin flips
  - Atomic decay
  - Thermal noise
  - Electromagnetic noise
  - Physical variation
    - Clock drift
    - DRAM decay
    - Image sensor errors
    - SRAM startup-state
  - Lava Lamps
Generating Random Keys

- Harnessing physical randomness: “LavaRand”
  - True randomness from lava lamps
  - Used by CloudFlare today
Generating Random Keys

- Harnessing physical randomness: “LavaRand”
  - True randomness from lava lamps
  - Used by CloudFlare today

Highest guarantees of **security**
Generating Random Keys

- Harnessing physical randomness: “LavaRand”
  - True randomness from lava lamps
  - Used by CloudFlare today

Highest guarantees of **security**

Difficult to use, or **rate-limited**
“Pseudo” Randomness

- What is true randomness?
  - Physical process that’s inherently random
  - Secure yet impractical
    - Scarce, hard to use
    - Rate-limited
“Pseudo” Randomness

- What is true randomness?
  - Physical process that’s inherently random
  - Secure yet impractical
    - Scarce, hard to use
    - Rate-limited

- Pseudo-random generator (PRG)
  - Input: a small seed that is truly random
  - Output: long sequence that appears random
“Pseudo” Randomness

- What is true randomness?
  - Physical process that’s inherently random
  - Secure yet impractical
    - Scarce, hard to use

- Pseudo-random generator (PRG)
  - Input: a small seed that is truly random
  - Output: long sequence that appears random

PRGs offer the best of both worlds: practical (fast, easy-to-use) and secure (appear random)
We say a **PRG** is **secure** if Mallory can’t do better than random guessing.
Pseudo-random Generators (PRGs)

- We say a PRG is secure if Mallory can’t do better than random guessing.

- **Problem**: How much **true randomness** is enough?
  - **Example**: one coin flip = Mallory needs **very few tries** to guess.
Pseudo-random Generators (PRGs)

- We say a PRG is secure if Mallory can’t do better than random guessing.

- **Problem:** How much true randomness is enough?
  - Example: one coin flip = Mallory needs very few tries to guess.

- **Problem:** Is our “true randomness” truly random?
  - Example: coin flip output = one in two. Lava lamps have way more!
Pseudo-random Generators (PRGs)

- We say a PRG is **secure** if Mallory can’t do better than random guessing

- **Problem:** How much **true randomness** is **enough**?
  - **Example:** one coin flip = Mallory needs very few tries to guess

- **Problem:** Is our “true randomness” **truly random**?
  - **Example:** coin flip output = one in two. Lava lamps have way more!

- **Solutions:**
  - Generate a bunch of true randomness **over a long time** from a **high entropy source**
  - Run through a PRF to get an easy-to-work-with, **fixed-length** randomness (e.g., 256 bits)
Constructing a PRG

- **Idea:** Build a PRG using a PRF
Constructing a PRG

- **Idea:** Build a PRG using a PRF

- **Observation:** PRF, given consecutive inputs, produce outputs that are randomly distributed (hopefully)
Constructing a PRG

- **Idea:** Build a PRG using a PRF

- **Observation:** PRF, given consecutive inputs, produce outputs that are randomly distributed (hopefully)

- **Result:** For truly-random s and PRF f:
  - Pseudo-random generated string = $f_s(0) \ || \ f_s(1) \ || \ f_s(2) \ || \ f_s(3) \ ...$
### Proving a PRG is Secure

- **Theorem:** if $f$ is a **secure** PRF
  - ... and $g$ is seeded from $f$
  - ... then $g$ must be a **secure** PRG
Theorem: if \( f \) is a secure PRF
- ... and \( g \) is seeded from \( f \)
- ... then \( g \) must be a secure PRG

Proof: if \( f \) is a secure PRF, we must show that \( g \) is a secure PRG
1. Assume \( g \) actually is insecure... then Mallory can break it
2. If that were true, Mallory could also break the PRF too
3. This would contradict the fact that \( f \) is a secure PRF!
Theorem: if $f$ is a secure PRF

... and $g$ is seeded from $f$

... then $g$ must be a secure PRG

Proof: if $f$ is a secure PRF, we must show that $g$ is a secure PRG.

1. Assume $g$ actually is insecure... then Mallory can break it.
2. If that were true, Mallory could also break the PRF too.
3. This would contradict the fact that $f$ is a secure PRF!

How should we seed our PRG?

What happens if we fail?
Theorem: if $f$ is a secure PRF
- ... and $g$ is seeded from $f$
- ... then $g$ must be a secure PRG

Proof: if
1. Assume $g$ is insecure
2. If that were true, Mallory could also break the PRF too
3. This would contradict the fact that $f$ is a secure PRF!

When our assumptions hold, we transform a small amount of "true" randomness into a wealth of "apparent" randomness
Practical Randomness

- Where do you get true randomness?
- Modern OSes typically collect randomness
- They give you API calls to capture it
- e.g., Linux:
  - `/dev/random` is a device that gives random bits; it blocks until available
  - `/dev/urandom` gives output of a PRG; nonblocking; seeded from `/dev/random` eventually
Plaintext-length Keys: One-time Pads
Alice and Bob generate a plaintext-length string of random bits: the one-time pad $k$. 
Alice and Bob generate a **plaintext-length** string of **random bits**: the one-time pad $k$

- Encryption: $c_i := p_i \oplus k_i$
- Decryption: $p_i := c_i \oplus k_i$

$$a \oplus b \oplus b = a$$

$$a \oplus b \oplus a = b$$
One-time Pads

- Alice and Bob generate a **plaintext-length** string of **random bits**: the one-time pad \( k \)
  - Encryption: \( c_i := p_i \text{ XOR } k_i \)
  - Decryption: \( p_i := c_i \text{ XOR } k_i \)

- To be secure:
  - Key must be **truly random**
  - Key must never be **reused**

\[
a \text{ XOR } b \text{ XOR } b = a \\
a \text{ XOR } b \text{ XOR } a = b
\]
Suppose the key bits aren’t truly random
- E.g., generated by selecting one of three values

How would this help Mallory?
Suppose the key bits \textit{aren’t} truly random
- E.g., generated by selecting one of three values

\textbf{How would this help Mallory?}
1. She intercepts an encrypted message

\[(a \text{ XOR } k)\]
Attacking OTPs: Non-random Keys

- Suppose the key bits aren’t truly random
  - E.g., generated by selecting one of three values

- **How would this help Mallory?**
  1. She intercepts an encrypted message
  2. She guesses key values and decrypts

\[
(a \oplus k) \oplus g = g
\]
Attacking OTPs: Non-random Keys

- Suppose the key bits aren’t truly random
  - E.g., generated by selecting one of three values

- How would this help Mallory?
  1. She intercepts an encrypted message
  2. She guesses key values and decrypts
  3. She can recover parts of the plaintext!

\[ a \oplus k \oplus g = a \oplus (k \oplus g) \]
Attacking OTPs: Key Reuse

(a XOR k) ⊕ SEND CASH ⊕ (b XOR k)
Attacking OTPs: Key Reuse

\((a \text{ XOR } k) \oplus (b \text{ XOR } k)\)
Attacking OTPs: Key Reuse

$$(a \oplus k) \oplus (b \oplus k) = a \oplus b$$
Alice and Bob generate a plaintext-length string of random bits: the one-time pad $k$.

- Encryption: $c_i := p_i \oplus k_i$
- Decryption: $p_i := c_i \oplus k_i$

To be secure:
- Key must be truly random
- Key must never be reused

The XOR table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$a \text{ XOR } b \text{ XOR } b = a$

$\text{ Provably Secure (if key is random + not reused)}$
One-time Pads

- Alice and Bob generate a plaintext-length string of **random bits**: the one-time pad $k$
  - Encryption: $c_i := p_i \text{ XOR } k_i$
  - Decryption: $p_i := c_i \text{ XOR } k_i$
- To be secure:
  - Key must be **truly random**
  - Key must never be **reused**

**Provably Secure** (if key is **random** + not **reused**)

**Highly Impractical**
Impracticality of OTPs

- **Generating OTPs**
  - Slow and/or rate-limited
    - By hand, LavaRand, etc.

- **Deploying OTPs**
  - Potentially very long
  - Challenging to conceal

- **Cold War numbers stations**
  - Encrypted message sent via short-wave radio to agents
  - Agent decrypts with their OTP
    - Throw OTP away after!
  - Many remain in service today!
    - Lincolnshire Poacher
Questions?
Plaintext-length Keys: Stream Ciphers
Stream Cipher

- **Idea:** Use a **Pseudo-random Generator** instead of a truly random pad

- **Recall:** a secure PRG inputs a **true-random seed**, outputs a stream that’s **indistinguishable** from true randomness (unless attacker **knows seed**)

1. Start with a shared secret **truly random seed** (from a lava lamp, mouse clicks, etc.)
2. Alice & Bob each use this seed to seed their PRG and generate **k bits of PRG output**
3. To encrypt and decrypt, perform the same operations as the One-time Pad:
   - Encryption: $c_i := p_i \oplus k_i$
   - Decryption: $p_i := c_i \oplus k_i$
Stream Cipher

- **Idea:** Use a **Pseudo-random Generator** instead of a truly random pad.

- **Recall:** Secure PRG inputs a seed $k$, outputs a stream practically indistinguishable from true randomness (unless attacker knows $k$).

What if you **reuse** the PRG’s **random key** or its **output**?

1. Start with shared secret *truly random* number $k$ (e.g., from a lava lamp, mouse clicks, etc.).
2. Alice & Bob each use $k$ to seed their PRG.
3. To encrypt, **Alice XORs next bit** of her generator’s output with **next bit of plaintext**.
4. To decrypt, **Bob XORs next bit** of his generator’s output with **next bit of ciphertext**.
Stream Cipher

- **Idea:** Use a *Pseudo-random Generator* instead of a truly random pad

- **Recall:** Secure PRG inputs a seed $k$, outputs a stream practically indistinguishable from true randomness (unless attacker knows $k$)

1. Start with shared secret *truly random* number $k$ (e.g., from a lava lamp, mouse clicks, etc.)
2. Alice & Bob each use $k$ to seed their PRG
3. To encrypt, Alice XORs next bit of her generator’s output with next bit of plaintext
4. To decrypt, Bob XORs next bit of his generator’s output with next bit of ciphertext

- **What if you reuse the PRG’s random key or its output?**

  - Vulnerable to partial (or full) recovery of the plaintext!
Stream Cipher

- **Idea:** Use a pseudorandom generator instead of a truly random pad

- **Recall:** Secure PRG inputs a seed $k$, outputs a stream practically indistinguishable from true randomness (unless attacker knows $k$)

1. Start with shared secret key truly random number $k$
2. Alice & Bob each use $k$ to seed the PRG
3. To encrypt, Alice XORs next bit of her generator’s output with next bit of plaintext
4. To decrypt, Bob XORs next bit of his generator’s output with next bit of ciphertext

**What is the tradeoff between an OTP and Stream Cipher?**
Questions?
Transposition Ciphers
Transposition Ciphers

- **Substitution** ciphers swap-out plaintext symbols for others
  - E.g., shifting, XORing, etc.

- We’ve learned about several substitution ciphers
  - E.g., Caesar, Vigenere, one-time pad, stream cipher

- Can we come up with an alternative to substitution?
Transposition Ciphers

- **Substitution** ciphers swap-out plaintext symbols for others
  - E.g., shifting, XORing, etc.

- We’ve learned about several substitution ciphers
  - E.g., Caesar, Vigenere, one-time pad, stream cipher

- **Can we come up with an alternative to substitution?**

- **Transposition**: rearrange plaintext symbols to create ciphertext
Columnar Transposition

- **Rearrange plaintext symbols to create ciphertext**
  - Create a table with $|k|$ columns and $|p|/|k|$ rows ($k$ is the keyword)
  - Place plaintext symbols in columns (left to right), cycling around to next row of the first column when current row of last column is filled
  - Create the ciphertext by writing entire columns (as a serial stream) to the output, where the keyword determines the column order

- **Example:**
  - $k = \text{“ZEBRAS” (632415)}$
  - $p = \text{“We are discovered flee at once”}

| 6 | 3 | 2 | 4 | 1 | 5 |
Columnar Transposition

- **Rearrange plaintext symbols to create ciphertext**
  - Create a table with $|k|$ columns and $|p|/|k|$ rows ($k$ is the keyword)
  - Place plaintext symbols in columns (left to right), cycling around to next row of the first column when current row of last column is filled
  - Create the ciphertext by writing entire columns (as a serial stream) to the output, where the keyword determines the column order

- **Example:**
  - $k = “ZEBRAS”$ (632415)
  - $p = “We are discovered flee at once”

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>
Columnar Transposition

- **Rearrange plaintext symbols to create ciphertext**
  - Create a table with $|k|$ columns and $|p|/|k|$ rows ($k$ is the keyword)
  - Place plaintext symbols in columns (left to right), cycling around to next row of the first column when current row of last column is filled
  - Create the ciphertext by writing entire columns (as a serial stream) to the output, where the keyword determines the column order

- **Example:**
  - $k = \text{“ZEBRAS” (632415)}$
  - $p = \text{“We are discovered flee at once”}$
  - $c = \text{EVLN ACDE ESEA ROFO DEEC WIREE}$

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>
Columnar Transposition

- Rearrange plaintext symbols to create ciphertext
  - Create a table with $|k|$ columns and $|p|/|k|$ rows ($k$ is the keyword)
  - Place plaintext symbols in columns (left to right), cycling around to next row of the first column when current row of last column is filled
  - Create the ciphertext by writing entire columns (as a serial stream) to the output, where the keyword determines the column order

- Example:
  - $k =$ “ZEBRAS” (632415)
  - $p =$ “We are discovered flee at once”
  - $c =$ EVLNX ACDTQ ESEAM ROFOP DEECD WIREE
  - Replace null with nonsense symbol

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>
Rail Fence (aka Zig Zag or Scytale) Cipher

- Rearrange plaintext on downwards, diagonally successive “rails”

<table>
<thead>
<tr>
<th>E</th>
<th>C</th>
<th>R</th>
<th>L</th>
<th>T</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>R</td>
<td>D</td>
<td>S</td>
<td>O</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>E</td>
<td>A</td>
<td>O</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>I</td>
<td>V</td>
<td>D</td>
<td>E</td>
<td>N</td>
</tr>
</tbody>
</table>

- $c = \text{WECRLTE ERDSOEFEAOCAIVDEN}$
Rail Fence (aka Zig Zag or Scytale) Cipher

- Rearrange plaintext on downwards, diagonally successive “rails”

<table>
<thead>
<tr>
<th>W</th>
<th>E</th>
<th>C</th>
<th>R</th>
<th>L</th>
<th>T</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>R</td>
<td>D</td>
<td>S</td>
<td>O</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>A</td>
<td>I</td>
<td>V</td>
<td>D</td>
<td>E</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

- $c = \text{WECRLTE ERDSOEFEAOC AIVDEN}$

- **Decryption:** use same-diameter cylinder!
Columnar Cipher Cryptanalysis

- What does a **brute force** attack look like?
Columnar Cipher Cryptanalysis

- What does a **brute force** attack look like?
  1. Guess number of columns
  2. Rearrange ciphertext in (probably) wrong order
  3. Look for anagrams to get correct order
     - Harder if null characters are rewritten

- Weakness of a transposition cipher?
Columnar Cipher Cryptanalysis

- What does a **brute force** attack look like?
  1. Guess number of columns
  2. Rearrange ciphertext in (probably) wrong order
  3. Look for anagrams to get correct order
    - Harder if null characters are rewritten

- Weakness of a transposition cipher?
  - **Plaintext** characters end up in the ciphertext
Is it transposition or substitution?

- Given a message ciphertext, how can you determine whether a transposition or a substitution cipher encrypted the plaintext?
  - **Hint:** frequency analysis
Is it transposition or substitution?

- Given a message ciphertext, how can you determine whether a transposition or a substitution cipher encrypted the plaintext?
  - **Hint:** frequency analysis

- **Transposition:**
  - Letters have *expected* letter frequencies

- **Substitution:**
  - Letters have *different* letter frequencies

Ordered by frequency
How would you build a stronger columnar transposition cipher?

**Transpose multiple times** with same or different keywords

<table>
<thead>
<tr>
<th>6</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

\[ k_1 = \text{“ZEBRAS”} (632415) \]
\[ c_1 = \text{EVLNX ACDTQ ESEAM ROFOP DEECD WIREE} \]
How would you build a stronger columnar transposition cipher?

**Transpose multiple times** with same or different keywords

**k_1** = “ZEBRAS” (632415)

<table>
<thead>
<tr>
<th>6</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

**c_1** = EVLNX ACDTQ ESEAM ROFOP DEECD WIREE

**k_2** = “STRIPE” (632415)

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>4</th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>V</td>
<td>L</td>
<td>N</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>T</td>
<td>E</td>
<td>S</td>
<td>E</td>
<td>A</td>
</tr>
<tr>
<td>R</td>
<td>O</td>
<td>F</td>
<td>O</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

**c_2** = CAEIX NSOIN AEDRX LEFWS EDREE VTOCG
Stronger Transposition

- How would you build a stronger columnar transposition cipher?
  - **Transpose multiple times** with same or different keywords
    - **Myszkowski Transposition** on recurring letters in key

<table>
<thead>
<tr>
<th>T</th>
<th>O</th>
<th>M</th>
<th>A</th>
<th>T</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

\[ c = \text{ROFOXACDTESEAZDEECNWIREEEVLNQ} \]
Stronger Transposition

- How would you build a stronger columnar transposition cipher?
  - **Transpose multiple times** with same or different keywords
    - **Myszkowski Transposition** on recurring letters in key

<table>
<thead>
<tr>
<th>T</th>
<th>O</th>
<th>M</th>
<th>A</th>
<th>T</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

\[ c = \text{ROFOXACDTWESAEZDEECNWIREEEVLNQ} \]

<table>
<thead>
<tr>
<th>T</th>
<th>O</th>
<th>M</th>
<th>A</th>
<th>T</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

\[ c = \text{ROFOXACDTBEDSEEEACTWWEIVRLENEQ} \]
How would you build a stronger columnar transposition cipher?

**Fractionation:** convert letters into symbols and transpose those
- E.g., morse code encoding, bits instead of letters
How would you build a stronger columnar transposition cipher?

**Fractionation:** convert letters into symbols and transpose those
- E.g., morse code encoding, bits instead of letters

Suppose $p = “We are discovered...”$

- **Morse:** 0—– 0 02— 0— 0— oo 00 000 —0—0 ——— 000— 0 0—0 0 —oo
- **Binary:** 01010111 01100101 01100001 01110010 01100101 01100100 01101001 01110011
  01100011 01101111 01110110 01100101 01110010 01100101 01100100
### Stronger Transposition

- How would you build a stronger columnar transposition cipher?

- **Combine with a substitution cipher**
  - Makes anagram discovery more difficult

<table>
<thead>
<tr>
<th>6</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

\[ c_1 = \text{EVLNB ACDTA ESEAR ROFOX DEECB WIREE} \]
Stronger Transposition

- How would you build a stronger columnar transposition cipher?
  - Combine with a substitution cipher
    - Makes anagram discovery more difficult

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>E</td>
<td>A</td>
<td>R</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>I</td>
<td>S</td>
<td>C</td>
<td>O</td>
<td>V</td>
<td>E</td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>D</td>
<td>F</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>T</td>
<td>O</td>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

\[ c_1 = \text{EVLNB ACDTA ESEAR ROFOX DEECB WIREE} \]
\[ k_s = \text{ABCAB CABCA BCABC} \]
\[ c_2 = \text{EWNNC CCEVA FUEBT RPHOY FEFEB XKRFG} \]
Questions?
Cipher Metrics
“Confusion”
- Every bit of the ciphertext should depend on several parts of the plaintext
- Hides the relationship between ciphertext and the plaintext

“Diffusion”
- A change to one plaintext bit should change 50% of the ciphertext bits
- A change to one ciphertext should change 50% of the plaintext bits
- Plaintext features spread throughout the entire ciphertext

These are cipher metrics—how we “weigh” a cipher’s security
Cipher Metrics: Transposition Ciphers

- Do **transposition ciphers** achieve confusion or diffusion?
Cipher Metrics: Transposition Ciphers

- Do **transposition ciphers** achieve confusion or diffusion?
  - **Diffusion**—they spread the plaintext around!
Cipher Metrics: Substitution Ciphers

- What level of confusion & diffusion do simple substitution ciphers have?
What level of confusion & diffusion do simple substitution ciphers have?
- None—hence why frequency analysis is useful
- Changing one plaintext or key symbol changes one ciphertext symbol
How does low diffusion impact communication across a noisy channel?
Cipher Metrics: Noisy Channels

- How does **low diffusion** impact communication across a **noisy channel**?
  - Low diffusion = **more tolerant** to corrupted symbols

```
ABCDEFGH
BCD#$!H1
```
Questions?
Next time on CS 4440...

Block ciphers, AES, secure channels