MTAT.07.017
Applied Cryptography

Randomness, PRNG,
One-Time Pad, Stream Cipher

University of Tartu

Spring 2022
Randomness

• What is a random sequence?
  • Sequence of numbers that does not follow any deterministic pattern
  • None of the numbers can be predicted based on the previous numbers
  • Has no description shorter than itself
  • Sequence of bits that cannot be compressed

• Where do we need randomness in real life?

• Why do we need randomness in crypto?
  • For keys, passwords, nonces, etc.

• Where can we get random numbers?
  • Can we flip a coin to get a random number?
  • Can a computer program generate random numbers?
  • Thermal noise, photoelectric effect, quantum phenomena
Pseudo-Random Number Generator (PRNG)

Deterministic algorithm that produces an endless stream of numbers which is indistinguishable from truly random. The output is determined by the seed value.

Linux /dev/urandom implementation:

- Knowing some part of the input does not allow anything about the output to be predicted
- PRNG is used when true-RNG is not available
- Can be used to “extend” randomness
- Entropy of the output depends on the entropy of the input
Can we tell whether some sequence is random?

\[ \ldots 41592653589 \ldots \]

\[ 3.141592653589793 \ldots \]

\[ \ldots 000000 \ldots \ldots \]

- Statistical randomness tests
  - Able to “prove” non-randomness
Bits and bytes

Bit string:
100010000011

$2^{11} + 2^7 + 2^1 + 2^0$

Most significant bit (msb) – left-most bit

Bytes - 8-bit collections (0-255)

00000000 - 0
00000001 - 1
00000010 - 2
...
11111101 - 253
11111110 - 254
11111111 - 255 (2^8-1)

Byte - basic addressable element
<table>
<thead>
<tr>
<th>ASCII Table</th>
<th>Total</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 32 <code>&lt;NULL&gt;</code></td>
<td>64 @</td>
<td>96 <code> </code></td>
<td>128 Ä</td>
<td>160 †</td>
<td>192 ĺ</td>
<td>224 ±</td>
</tr>
<tr>
<td>1 33 <code>&lt;SOH&gt;</code></td>
<td>65 A</td>
<td>97 a</td>
<td>129 Å</td>
<td>161 ¨</td>
<td>193 ı</td>
<td>225 .</td>
</tr>
<tr>
<td>2 34 <code>&lt;STX&gt;</code></td>
<td>66 B</td>
<td>98 b</td>
<td>130 Ç</td>
<td>162 ç</td>
<td>194 ń</td>
<td>226 ,</td>
</tr>
<tr>
<td>3 35 <code>&lt;ETX&gt;</code></td>
<td>67 C</td>
<td>99 c</td>
<td>131 É</td>
<td>163 É</td>
<td>195 ų</td>
<td>227 &quot;</td>
</tr>
<tr>
<td>4 36 <code>&lt;ETX&gt;</code></td>
<td>68 D</td>
<td>100 d</td>
<td>132 Ñ</td>
<td>164 ñ</td>
<td>196 ę</td>
<td>228 %</td>
</tr>
<tr>
<td>5 37 <code>&lt;ENQ&gt;</code></td>
<td>69 E</td>
<td>101 e</td>
<td>133 Î</td>
<td>165 î</td>
<td>197 ±</td>
<td>229 Â</td>
</tr>
<tr>
<td>6 38 <code>&lt;ACK&gt;</code></td>
<td>70 F</td>
<td>102 f</td>
<td>134 Ù</td>
<td>166 ù</td>
<td>198 Æ</td>
<td>230 É</td>
</tr>
<tr>
<td>7 39 <code>&lt;BEL&gt;</code></td>
<td>71 G</td>
<td>103 g</td>
<td>135 â</td>
<td>167 â</td>
<td>199 «</td>
<td>231 À</td>
</tr>
<tr>
<td>8 40 <code>&lt;BS&gt;</code></td>
<td>72 H</td>
<td>104 h</td>
<td>136 Ð</td>
<td>168 Ð</td>
<td>200 »</td>
<td>232 Ê</td>
</tr>
<tr>
<td>9 41 <code>&lt;TAB&gt;</code></td>
<td>73 I</td>
<td>105 i</td>
<td>137 Ë</td>
<td>169 Ë</td>
<td>201 «</td>
<td>233 Ë</td>
</tr>
<tr>
<td>10 42 <code>&lt;LF&gt;</code></td>
<td>74 J</td>
<td>106 j</td>
<td>138 Ì</td>
<td>170 Ì</td>
<td>202 ±</td>
<td>234 Ì</td>
</tr>
<tr>
<td>11 43 <code>&lt;VT&gt;</code></td>
<td>75 K</td>
<td>107 k</td>
<td>139 Í</td>
<td>171 Í</td>
<td>203 À</td>
<td>235 Î</td>
</tr>
<tr>
<td>12 44 <code>&lt;FF&gt;</code></td>
<td>76 L</td>
<td>108 l</td>
<td>140 Å</td>
<td>172 Å</td>
<td>204 Â</td>
<td>236 Ï</td>
</tr>
<tr>
<td>13 45 <code>&lt;CR&gt;</code></td>
<td>77 M</td>
<td>109 m</td>
<td>141 ç</td>
<td>173 ç</td>
<td>205 Ø</td>
<td>237 Î</td>
</tr>
<tr>
<td>14 46 <code>&lt;SD&gt;</code></td>
<td>78 N</td>
<td>110 n</td>
<td>142 é</td>
<td>174 Æ</td>
<td>206 Ç</td>
<td>238 Ø</td>
</tr>
<tr>
<td>15 47 <code>/</code></td>
<td>79 O</td>
<td>111 o</td>
<td>143 è</td>
<td>175 Ø</td>
<td>207 ò</td>
<td>239 Ø</td>
</tr>
<tr>
<td>16 48 <code>&lt;DEL&gt;</code></td>
<td>80 P</td>
<td>112 p</td>
<td>144 ê</td>
<td>176 ô</td>
<td>208 –</td>
<td>240 ø</td>
</tr>
<tr>
<td>17 49 <code>&lt;DC1&gt;</code></td>
<td>81 Q</td>
<td>113 q</td>
<td>145 ê</td>
<td>177 ±</td>
<td>209 –</td>
<td>241 Ŏ</td>
</tr>
<tr>
<td>18 50 <code>&lt;DC2&gt;</code></td>
<td>82 R</td>
<td>114 r</td>
<td>146 i</td>
<td>178 ü</td>
<td>210 –</td>
<td>242 Ú</td>
</tr>
<tr>
<td>19 51 <code>&lt;DC3&gt;</code></td>
<td>83 S</td>
<td>115 s</td>
<td>147 i</td>
<td>179 ť</td>
<td>211 –</td>
<td>243 Ü</td>
</tr>
<tr>
<td>20 52 <code>&lt;DC4&gt;</code></td>
<td>84 T</td>
<td>116 t</td>
<td>148 ì</td>
<td>180 ì</td>
<td>212 –</td>
<td>244 Ü</td>
</tr>
<tr>
<td>21 53 <code>&lt;NUL&gt;</code></td>
<td>85 U</td>
<td>117 u</td>
<td>149 ï</td>
<td>181 µ</td>
<td>213 –</td>
<td>245 Ï</td>
</tr>
<tr>
<td>22 54 <code>&lt;SYN&gt;</code></td>
<td>86 V</td>
<td>118 v</td>
<td>150 ñ</td>
<td>182 ñ</td>
<td>214 –</td>
<td>246 Ì</td>
</tr>
<tr>
<td>23 55 <code>&lt;ETB&gt;</code></td>
<td>87 W</td>
<td>119 w</td>
<td>151 ó</td>
<td>183 Ï</td>
<td>215 –</td>
<td>247 Ï</td>
</tr>
<tr>
<td>24 56 <code>&lt;CAN&gt;</code></td>
<td>88 X</td>
<td>120 x</td>
<td>152 ô</td>
<td>184 Ï</td>
<td>216 Ï</td>
<td>248 Ï</td>
</tr>
<tr>
<td>25 57 <code>/</code></td>
<td>89 Y</td>
<td>121 y</td>
<td>153 ô</td>
<td>185 Ï</td>
<td>217 Ï</td>
<td>249 Ï</td>
</tr>
<tr>
<td>26 58 <code>&lt;SUB&gt;</code></td>
<td>90 Z</td>
<td>122 z</td>
<td>154 ô</td>
<td>186 ô</td>
<td>218 Ï</td>
<td>250 Ï</td>
</tr>
<tr>
<td>27 59 <code>&lt;ESC&gt;</code></td>
<td>91 <code> </code></td>
<td>123 <code> </code></td>
<td>155 ô</td>
<td>187 ö</td>
<td>219 Ï</td>
<td>251 Ï</td>
</tr>
<tr>
<td>28 60 <code>&lt;FS&gt;</code></td>
<td>92 \</td>
<td>124 \</td>
<td>156 ú</td>
<td>188 ø</td>
<td>220 Ï</td>
<td>252 Ï</td>
</tr>
<tr>
<td>29 61 <code>&lt;GS&gt;</code></td>
<td>93 J</td>
<td>125 J</td>
<td>157 û</td>
<td>189 Ø</td>
<td>221 Ï</td>
<td>253 Ï</td>
</tr>
<tr>
<td>30 62 <code>&lt;RS&gt;</code></td>
<td>94 ^</td>
<td>126 ^</td>
<td>158 û</td>
<td>190 æ</td>
<td>222 Ï</td>
<td>254 Ï</td>
</tr>
<tr>
<td>31 63 <code>&lt;US&gt;</code></td>
<td>95 _</td>
<td>127 _</td>
<td>159 û</td>
<td>191 ø</td>
<td>223 Ï</td>
<td>255 Ï</td>
</tr>
</tbody>
</table>

Source: [http://www.asciitable.com/](http://www.asciitable.com/)
### Hexadecimal (Base16) encoding

<table>
<thead>
<tr>
<th>Hex</th>
<th>Value</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>'1'</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>'2'</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>'3'</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>'4'</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>'5'</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>'6'</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>'7'</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>'8'</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>'9'</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>'A'</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>'B'</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>'C'</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>'D'</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>'E'</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>'F'</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

- One hex symbol represents 4 bits
- Two hex symbols needed to represent a byte

$$2E = 0010 \ 1110$$
Base64 encoding

bn+ITbj/TRwcSAwT8CZnFZN0me5/AGdFIGNLBPPo7Nc07T6XTpsTw0QxnM++9xJXKkEEcaEn2Vo9MiAVPVUR5PsFGKZbL7coPRdHD058RokCF4aizWv6+Dqg0lsXsmXliWusnOQ==

- Represent binary data using printable characters
- Base64 encoded data approximately 33% larger
Bitwise operations

AND:
• extract partition of bit string

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \text{(bit mask)} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{(AND)}
\end{array}
\]

OR:
• set specific bits

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & \text{(OR)}
\end{array}
\]

XOR:
• flip specific bits

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \text{(XOR)}
\end{array}
\]

Shift:
• shift and pad with 0

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \text{(right shift by two)}
\end{array}
\]
Bitwise operation: AND

Example:

0 0 1 1 1 1 0 0
0 0 0 0 0 1 1 0 (bit mask)
---------------
0 0 0 0 0 1 0 0 (AND)

- Extract bits we are interested in

Python:

```python
>>> 60 & 6
4
```
Bitwise operation: OR

Example:

0 0 1 1 1 1 0 0
0 0 0 0 0 1 1 0
---------------
0 0 1 1 1 1 1 0 (OR)

- Set specific bits

Python:

```python
>>> 60 | 6
62
```
Bitwise operation: XOR

Example:

```
  0 0 1 1 1 1 0 0
  0 0 0 0 0 1 1 0
----------------
  0 0 1 1 1 0 1 0 (XOR)
```

- Flip specific bits

Python:

```
>>> 60 ^ 6
58
```
Bitwise operation: Shift

Example:

0 0 1 1 1 1 0 0

---------------

0 0 0 0 0 1 1 1 1 1 (right shift by two)

- Shift (right or left) and pad with zeros
- Fast multiplication and division by 2

Python:

```python
>>> 60 >> 2
15
>>> 15 << 1
30
```
One-Time Pad (OTP)

100% secure, unbreakable encryption!

- Key generation: the key (one-time pad) is a random sequence the same length as the plaintext
- Encryption operation: XOR (⊕) the plaintext with the key
- Decryption operation: XOR (⊕) the ciphertext with the key
One-Time Pad (OTP)

Information-theoretically secure (unbreakable), if:

• Key (one-time pad) is truly random
• Key is never reused

plaintext1 \oplus key = ciphertext1
plaintext2 \oplus key = ciphertext2 \oplus plaintext2 = key
key \oplus ciphertext1 = plaintext1

• Not used in practice
Stream cipher

- Key generation: a small key “seeds” the PRNG to generate keystream
- Encryption operation: $\text{XOR (⊕)}$ the plaintext with the keystream
- Decryption operation: $\text{XOR (⊕)}$ the ciphertext with the keystream

Stream ciphers differ by the PRNG used
Why is it less secure than one-time pad?
Encryption on its own does not provide integrity!
The same key (keystream) must never be reused!
Stream cipher

Solution – on every encryption add a unique nonce to the key:

- The same nonce must never be reused!
- How to generate nonce?
  - Random value
  - Counter value
  - Current time
Questions

- Where can we get (true) random numbers?
- Why is a pseudo-random number not as good as a random number?
- What are the properties of a random sequence?
- Can we tell whether the provided sequence is random?
- What happens to data if we XOR it with random data?
- Why are brute-force attacks ineffective in breaking one-time pad?
- Why is unbreakable one-time pad not used in enterprise products?
- How is stream cipher different from one-time pad?
Task: One-Time Pad (OTP)

Implement the one-time pad cryptosystem.

Encryption should produce a random key file and encrypted output file:

$ chmod +x otp.py
$ ./otp.py encrypt datafile datafile.key datafile.encrypted

Decryption should use the key file and produce decrypted original plaintext file:

$ ./otp.py decrypt datafile.encrypted datafile.key datafile.plain

• Commit “01/otp.py” to your repository:

  $ git add 01/otp.py
  $ git commit -m "homework 01 solution" 01/otp.py
  $ git push
Task: Template

#!/usr/bin/env python3
import os, sys # do not use any other imports/libraries
# took x.y hours (please specify here how much time your solution required)

def bn(b):
    # b - bytes to encode as integer
    # your implementation here
    return i

def nb(i, length):
    # i - integer to encode as bytes
    # length - specifies in how many bytes the number should be encoded
    # your implementation here
    b = b''
    return b

def encrypt(pfile, kfile, cfile):
    # your implementation here
    pass

def decrypt(cfile, kfile, pfile):
    # your implementation here
    pass

def usage():
    print("Usage:")
    print("encrypt <plaintext file> <output key file> <ciphertext output file>")
    print("decrypt <ciphertext file> <key file> <plaintext output file>")
    sys.exit(1)

if len(sys.argv) != 5:
    usage()
elif sys.argv[1] == 'encrypt':
    encrypt(sys.argv[2], sys.argv[3], sys.argv[4])
elif sys.argv[1] == 'decrypt':
    decrypt(sys.argv[2], sys.argv[3], sys.argv[4])
else:
    usage()
Python 3 str and bytes data objects

str object stores Unicode characters:

```python
>>> s = 'Fōō'
>>> type(s), len(s)
(<class 'str'>, 3)
>>> s[0], s[1], s[2]
('F', 'ो', 'ो')
```

```python
>>> b = s.encode('utf8')
>>> type(b), len(b)
(<class 'bytes'>, 5)
```

bytes object stores bytes:

```python
>>> b
b'F\xc5\xa8\xc5\xa8'
>>> type(b), len(b)
(<class 'bytes'>, 5)
```

```python
>>> b.decode('utf8')
'Fōō'
```

```python
>>> b = 'abC'
>>> b = bytes([97, 98, 0x43])
>>> b
b'abC'
>>> b[0]
97
>>> b[0:1]
b'a'
```

```python
>>> import codecs
>>> codecs.encode(b, 'base64')
b'YWJD\n'
>>> codecs.encode(b, 'hex')
'b616243'
>>> codecs.encode(b, 'hex').decode('ascii')
'616243'
>>> b.hex()
'616243'
```
Python: bytes to integer

```python
>>> b = b'abC'
>>> i = b[0]
>>> i
97
>>> bin(i)
'0b1100001'
>>> i = i << 8
>>> bin(i)
'0b1100001000000000'
>>> i = i | b[1]
>>> bin(i)
'0b110000101100010'
>>> i = i << 8
>>> bin(i)
'0b110000101100010000000000'
>>> i = i | b[2]
>>> bin(i)
'0b11000010110001001000011'
>>> i
6382147
```

- Convert first byte to integer
- Left-shift integer 8 times
- Convert second byte to integer
- Load second integer in first 8 bits
- ...

Task: One-Time Pad (OTP)

**Encrypter:**
- Read the plaintext file content into bytes object (e.g., `b = open('file.txt', 'rb').read()`)
- Convert plaintext bytes to one big integer
- Obtain random key the same length as plaintext (use `os.urandom()`)
- Convert key bytes to one big integer
- XOR plaintext and key integers (please, use this approach)
- Save the key (one-time pad) and XOR’ed result (ciphertext) to file:
  - Convert ciphertext integer to bytes object
- Once more: use bitwise operations!
  - Banned: functions: `to_bytes()`, `from_bytes()` and operator `**`!

**Decrypter:**
- Perform the operations in reverse order
Task: Test Case

```bash
$ echo -n -e "\x85\xce\xa2\x25" > file.enc
$ hexdump -C file.enc
 00000000  85 ce a2 25  |...%|
$ echo -n -e "\xe4\xac\xe1\x2f" > file.key
$ hexdump -C file.key
 00000000  e4 ac e1 2f  |.../|
$ ./otp.py decrypt file.enc file.key file.plain
$ hexdump -C file.plain
 00000000  61 62 43 0a  |abC.|

$ echo -n -e "\x00\x00\x61\x62\x43\x00" > file.plain
$ hexdump -C file.plain
 00000000  00 00 61 62 43 00  |..abC.|  
$ ./otp.py encrypt file.plain file.key file.enc
$ ./otp.py decrypt file.enc file.key fileorig.plain
$ hexdump -C fileorig.plain
 00000000  00 00 61 62 43 00  |..abC.|  

Note that when you convert bytes to integer, you loose the most significant zero bytes.

Is this implementation secure?
Please!

• Include information of how much time the tasks took (as a comment at the top of your source code)

• Give feedback about the parts that were hard to grasp or you have an idea for improvement

• Do not waste your time on input validation

• Do not use imports/libraries that are not explicitly allowed

• The output of your solution must byte-by-byte match the format of example output shown on the slides
  • Remove any non-required debugging output before committing
  • Unless required, the solution must not create/delete any files

• Commit the (finished) solution to the main branch of your repository with the filename required

Thank you!