MTAT.07.017
Applied Cryptography

Randomness, PRNG,
One-Time Pad, Stream Cipher

University of Tartu

Fall 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
Randomness

• Can we tell whether some sequence is random?
  
  . . . 41592653589 . . .
  3 . 141592653589793 . . .
  
  . . . 0000000000 . . .

• 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
# ASCII Table

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;NULL&gt;</td>
<td>32</td>
<td>@</td>
<td>64</td>
<td>$</td>
<td>128</td>
<td>À</td>
<td>160</td>
<td>Ÿ</td>
</tr>
<tr>
<td>1</td>
<td>&lt;SOM&gt;</td>
<td>33</td>
<td>!</td>
<td>65</td>
<td>A</td>
<td>97</td>
<td>a</td>
<td>129</td>
<td>À</td>
</tr>
<tr>
<td>2</td>
<td>&lt;STX&gt;</td>
<td>34</td>
<td>&quot;</td>
<td>66</td>
<td>B</td>
<td>98</td>
<td>b</td>
<td>130</td>
<td>Ç</td>
</tr>
<tr>
<td>3</td>
<td>&lt;ETX&gt;</td>
<td>35</td>
<td>#</td>
<td>67</td>
<td>C</td>
<td>99</td>
<td>c</td>
<td>131</td>
<td>É</td>
</tr>
<tr>
<td>4</td>
<td>&lt;EOT&gt;</td>
<td>36</td>
<td>$</td>
<td>68</td>
<td>D</td>
<td>100</td>
<td>d</td>
<td>132</td>
<td>Ń</td>
</tr>
<tr>
<td>5</td>
<td>&lt;END&gt;</td>
<td>37</td>
<td>%</td>
<td>69</td>
<td>E</td>
<td>101</td>
<td>e</td>
<td>133</td>
<td>Ô</td>
</tr>
<tr>
<td>6</td>
<td>&lt;ACK&gt;</td>
<td>38</td>
<td>&amp;</td>
<td>70</td>
<td>F</td>
<td>102</td>
<td>f</td>
<td>134</td>
<td>Ú</td>
</tr>
<tr>
<td>7</td>
<td>&lt;BEL&gt;</td>
<td>39</td>
<td>'</td>
<td>71</td>
<td>G</td>
<td>103</td>
<td>g</td>
<td>135</td>
<td>á</td>
</tr>
<tr>
<td>8</td>
<td>&lt;BS&gt;</td>
<td>40</td>
<td>(</td>
<td>72</td>
<td>H</td>
<td>104</td>
<td>h</td>
<td>136</td>
<td>à</td>
</tr>
<tr>
<td>9</td>
<td>&lt;TAB&gt;</td>
<td>41</td>
<td>)</td>
<td>73</td>
<td>I</td>
<td>105</td>
<td>i</td>
<td>137</td>
<td>â</td>
</tr>
<tr>
<td>10</td>
<td>&lt;LF&gt;</td>
<td>42</td>
<td>*</td>
<td>74</td>
<td>J</td>
<td>106</td>
<td>j</td>
<td>138</td>
<td>å</td>
</tr>
<tr>
<td>11</td>
<td>&lt;VT&gt;</td>
<td>43</td>
<td>+</td>
<td>75</td>
<td>K</td>
<td>107</td>
<td>k</td>
<td>139</td>
<td>ā</td>
</tr>
<tr>
<td>12</td>
<td>&lt;FF&gt;</td>
<td>44</td>
<td>,</td>
<td>76</td>
<td>L</td>
<td>108</td>
<td>l</td>
<td>140</td>
<td>ď</td>
</tr>
<tr>
<td>13</td>
<td>&lt;CR&gt;</td>
<td>45</td>
<td>-</td>
<td>77</td>
<td>M</td>
<td>109</td>
<td>m</td>
<td>141</td>
<td>č</td>
</tr>
<tr>
<td>14</td>
<td>&lt;SD&gt;</td>
<td>46</td>
<td>.</td>
<td>78</td>
<td>N</td>
<td>110</td>
<td>n</td>
<td>142</td>
<td>é</td>
</tr>
<tr>
<td>15</td>
<td>&lt;SI&gt;</td>
<td>47</td>
<td>/</td>
<td>79</td>
<td>O</td>
<td>111</td>
<td>o</td>
<td>143</td>
<td>È</td>
</tr>
<tr>
<td>16</td>
<td>&lt;DEL&gt;</td>
<td>48</td>
<td>0</td>
<td>80</td>
<td>P</td>
<td>112</td>
<td>p</td>
<td>144</td>
<td>Ñ</td>
</tr>
<tr>
<td>17</td>
<td>&lt;DC1&gt;</td>
<td>49</td>
<td>1</td>
<td>81</td>
<td>Q</td>
<td>113</td>
<td>q</td>
<td>145</td>
<td>Ñ</td>
</tr>
<tr>
<td>18</td>
<td>&lt;DC2&gt;</td>
<td>50</td>
<td>2</td>
<td>82</td>
<td>R</td>
<td>114</td>
<td>r</td>
<td>146</td>
<td>í</td>
</tr>
<tr>
<td>19</td>
<td>&lt;DC3&gt;</td>
<td>51</td>
<td>3</td>
<td>83</td>
<td>S</td>
<td>115</td>
<td>s</td>
<td>147</td>
<td>Í</td>
</tr>
<tr>
<td>20</td>
<td>&lt;DC4&gt;</td>
<td>52</td>
<td>4</td>
<td>84</td>
<td>T</td>
<td>116</td>
<td>t</td>
<td>148</td>
<td>ª</td>
</tr>
<tr>
<td>21</td>
<td>&lt;NAR&gt;</td>
<td>53</td>
<td>5</td>
<td>85</td>
<td>U</td>
<td>117</td>
<td>u</td>
<td>149</td>
<td>Í</td>
</tr>
<tr>
<td>22</td>
<td>&lt;SYN&gt;</td>
<td>54</td>
<td>6</td>
<td>86</td>
<td>V</td>
<td>118</td>
<td>v</td>
<td>150</td>
<td>Ń</td>
</tr>
<tr>
<td>23</td>
<td>&lt;ETB&gt;</td>
<td>55</td>
<td>7</td>
<td>87</td>
<td>W</td>
<td>119</td>
<td>w</td>
<td>151</td>
<td>Ó</td>
</tr>
<tr>
<td>24</td>
<td>&lt;CAN&gt;</td>
<td>56</td>
<td>8</td>
<td>88</td>
<td>X</td>
<td>120</td>
<td>x</td>
<td>152</td>
<td>Ó</td>
</tr>
<tr>
<td>25</td>
<td>&lt;EM&gt;</td>
<td>57</td>
<td>9</td>
<td>89</td>
<td>9</td>
<td>121</td>
<td>y</td>
<td>153</td>
<td>Ó</td>
</tr>
<tr>
<td>26</td>
<td>&lt;SUB&gt;</td>
<td>58</td>
<td>;</td>
<td>90</td>
<td>Z</td>
<td>122</td>
<td>z</td>
<td>154</td>
<td>ó</td>
</tr>
<tr>
<td>27</td>
<td>&lt;ESC&gt;</td>
<td>59</td>
<td>&lt;</td>
<td>91</td>
<td>[</td>
<td>123</td>
<td>{</td>
<td>155</td>
<td>ŕ</td>
</tr>
<tr>
<td>28</td>
<td>&lt;FS&gt;</td>
<td>60</td>
<td>&gt;</td>
<td>92</td>
<td>\</td>
<td>124</td>
<td></td>
<td></td>
<td>156</td>
</tr>
<tr>
<td>29</td>
<td>&lt;GS&gt;</td>
<td>61</td>
<td>?</td>
<td>93</td>
<td>]</td>
<td>125</td>
<td>}</td>
<td>157</td>
<td>ņ</td>
</tr>
<tr>
<td>30</td>
<td>&lt;RS&gt;</td>
<td>62</td>
<td>^</td>
<td>94</td>
<td>Å</td>
<td>126</td>
<td>~</td>
<td>158</td>
<td>Õ</td>
</tr>
<tr>
<td>31</td>
<td>&lt;US&gt;</td>
<td>63</td>
<td>_</td>
<td>95</td>
<td>–</td>
<td>127</td>
<td>&lt;DEL&gt;</td>
<td>159</td>
<td>Õ</td>
</tr>
</tbody>
</table>

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

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

AND:
• extract partion of bit string

```
0 0 1 1 1 1 0 0
0 0 0 0 0 1 1 0 (bit mask) >>> 60 & 6
---------------
0 0 0 0 0 1 0 0 (AND)
```

OR:
• set specific bits

```
0 0 1 1 1 1 0 0
0 0 0 0 0 1 1 0 >>> 60 | 6
---------------
0 0 1 1 1 1 1 0 (OR)
```

XOR:
• flip specific bits

```
0 0 1 1 1 1 0 0
0 0 0 0 0 1 1 0 >>> 60 ^ 6
---------------
0 0 1 1 1 0 1 0 (XOR)
```

Shift:
• shift and pad with 0

```
0 0 1 1 1 1 0 0 >>> 60 >> 2
---------------
0 0 0 0 1 1 1 1 (right shift by two)
```
Bitwise operation: AND

Example:

\[
\begin{align*}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \quad \text{(bit mask)} \\
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \quad \text{(AND)}
\end{align*}
\]

- 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:

```
>>> 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:

```
>>> 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 ($\oplus$) the plaintext with the key
- Decryption operation: XOR ($\oplus$) the ciphertext with the key

Why it works? Why is it secure?
One-Time Pad (OTP)

Information-theoretically secure (unbreakable), if:

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

\[
\begin{align*}
\text{plaintext}_1 \oplus \text{key} &= \text{ciphertext}_1 \\
\text{plaintext}_2 \oplus \text{key} &= \text{ciphertext}_2 \oplus \text{plaintext}_2 = \text{key} \\
\text{key} \oplus \text{ciphertext}_1 &= \text{plaintext}_1
\end{align*}
\]

- Not used in practice
Stream cipher

• Key generation: a small key “seeds” the PRNG to generate keystream
• Encryption operation: XOR ($\oplus$) the plaintext with the keystream
• Decryption operation: XOR ($\oplus$) 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
#!/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 bi(b):
    # b - bytes to encode as an integer
    # your implementation here
    return i

def ib(i, length):
    # i - an 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 = 'Fôô'
>>> type(s), len(s)
(<class 'str'>, 3)
>>> s[0], s[1], s[2]
('F', 'ô', 'ô')
>>> b = s.encode('utf8')
```

bytes object stores bytes:

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

>>> import codecs
>>> codecs.encode(b, 'base64')
b'YWJD

>>> codecs.encode(b, 'hex')
b'616243'
>>> 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!  

- **Decrypter:**
  - Perform the operations in reverse order  

- Banned functions: `int()`, `str()`, `bin()`, `hex()`, `to_bytes()`, `from_bytes()`, operators `*`, `**`, `/`, `%!`
Task: Test Case

```
$ 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!