MTAT.07.017
Applied Cryptography

Introduction, Randomness, PRNG,
One-Time Pad, Stream Ciphers

University of Tartu

Spring 2020
Who am I?

Arnis Paršovs

MSc in Cyber Security
TalTech, 2012

Computer Science PhD student at UT

Applied Cyber Security Group: https://acs.cs.ut.ee/
Who are you?

- MSc (Cyber Sec.) - 22
- BSc (Computer Sci.) - 5
- MSc (Computer Sci.) - 4
- PhD (Computer Sci.) - 3
- PhD (Math) - 1
- BSc (IT Systems) - 1
- MSc (Software Eng.) - 1
This course

- Practical course
- No proofs – just intuition
- Learn by implementing
Course timeline

16 weeks

- Lecture: will be published by every Monday 23:59
- Practice: Thursdays 14:15–16:00 (Narva mnt 18-1019)

6 ECTS – 10 hours weekly

- 2 hours for lectures
- 8 hours on homework (may vary)
Grading

• Homework every week
• Homeworks give maximum 70% of the final grade
• Deadlines are strict!
  • Homework deadline – beginning of the next lecture
  • Late submissions get 50% of the grade
  • Homeworks submitted later than 1 week after the deadline are not accepted!
• Exam gives another 30% of the final grade
  • Should be easy if you follow the lectures
Homework submission

• Homeworks must be implemented in Python 3
  • Test environment: Ubuntu 19.10, Python 3.6.x
  • Python packages from Ubuntu package repository (not pip)
• Create a private Bitbucket repository and grant me ‘read’ privileges:
  https://bitbucket.org/appcrypto/2020/src/master/setup/
• Add your repository to the course grading page at
  https://cybersec.ee/appcrypto2020/
• Homework templates will be published at course repository:
  https://bitbucket.org/appcrypto/2020/
• Feedback will be given using code comment feature
• Teaching assistance over e-mail not available
• Do not look on homework solutions of others!
  • Plagiarism cases will be handled in accordance with UT Plagiarism Policy
Academic fraud

- It is an academic fraud to collaborate with other people on work that is required to be completed and submitted individually.

- The homeworks in Applied Cryptography course are required to be completed and submitted individually!

- You can help your peers to learn by explaining concepts, but don’t provide them with answers or your own work!
  - If you don’t see the borders – work alone.

- Copying code samples from internet resources (e.g., stackoverflow.com) may be considered plagiarism:
  - the most basic building blocks may be OK
  - combination (composition) of building blocks is NOT OK
- If you don’t see the borders – limit yourself to Python API documentation.
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 the real life?
• Why do we need randomness in crypto?
  • For keys, passwords, nonces, etc.

• Where we can 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 endless stream of numbers which are 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 to predict anything about the output
- 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...
  ...

  ...000000.......

• 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)

Byte - basic addressable element
### ASCII Table

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;NULL&gt;</td>
</tr>
<tr>
<td>1</td>
<td>&lt;SOH&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;STX&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;ETX&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;EOT&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;ENQ&gt;</td>
</tr>
<tr>
<td>6</td>
<td>&lt;ACK&gt;</td>
</tr>
<tr>
<td>7</td>
<td>&lt;BEL&gt;</td>
</tr>
<tr>
<td>8</td>
<td>&lt;BS&gt;</td>
</tr>
<tr>
<td>9</td>
<td>&lt;TAB&gt;</td>
</tr>
<tr>
<td>10</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>11</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>12</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>13</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>14</td>
<td>&lt;SO&gt;</td>
</tr>
<tr>
<td>15</td>
<td>&lt;SI&gt;</td>
</tr>
<tr>
<td>16</td>
<td>&lt;DLE&gt;</td>
</tr>
<tr>
<td>17</td>
<td>&lt;DC1&gt;</td>
</tr>
<tr>
<td>18</td>
<td>&lt;DC2&gt;</td>
</tr>
<tr>
<td>19</td>
<td>&lt;DC3&gt;</td>
</tr>
<tr>
<td>20</td>
<td>&lt;DC4&gt;</td>
</tr>
<tr>
<td>21</td>
<td>&lt;NAK&gt;</td>
</tr>
<tr>
<td>22</td>
<td>&lt;SYN&gt;</td>
</tr>
<tr>
<td>23</td>
<td>&lt;ETB&gt;</td>
</tr>
<tr>
<td>24</td>
<td>&lt;CAN&gt;</td>
</tr>
<tr>
<td>25</td>
<td>&lt;EM&gt;</td>
</tr>
<tr>
<td>26</td>
<td>&lt;SUB&gt;</td>
</tr>
<tr>
<td>27</td>
<td>&lt;ESC&gt;</td>
</tr>
<tr>
<td>28</td>
<td>&lt;FS&gt;</td>
</tr>
<tr>
<td>29</td>
<td>&lt;GS&gt;</td>
</tr>
<tr>
<td>30</td>
<td>&lt;RS&gt;</td>
</tr>
<tr>
<td>31</td>
<td>&lt;US&gt;</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

bn+ITbj/TRwcSAwT8CZnFZN0me5/AGdFIGNLBPp07Nc07T6XTpsTw0QxnM++9xJXKkEEcaEn2Vo9MiAVPVUR5PsFGKZbL7coPRdHD058RokCF4aizWv6+Dqg0lsXsmXliWusn0Q==

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

AND:

• extract partion of bit string

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

\[
\begin{array}{llllllllll}
\text{---} \\
4 \\
\end{array}
\]

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

OR:

• set specific bits

\[
\begin{array}{llllllllll}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]
\[
\begin{array}{llllllllll}
\text{---} \\
62 \\
\end{array}
\]

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

XOR:

• flip specific bits

\[
\begin{array}{llllllllll}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]
\[
\begin{array}{llllllllll}
\text{---} \\
58 \\
\end{array}
\]

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

Shift:

• shift and pad with 0

\[
\begin{array}{llllllllll}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]
\[
\begin{array}{llllllllll}
\text{---} \\
15 \\
\end{array}
\]

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

(right shift by two)
Bitwise operation: AND

• Extract bits we are interested in

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)
```

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

- Set specific bits

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)
```

Python:

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

- Flip specific bits

Example:

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

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

Python:

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

- Shift (right or left) and pad with zeros

Example:

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

Python:

>>> 60 >> 2
15
>>> 15 << 1
30

- Fast multiplication and division by 2
One-Time Pad (OTP)

- 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

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

- Not used in practice
Stream cipher

- Key generation: a small key “seeds” the PRNG
- Encryption operation: XOR (⊕) the plaintext with the key
- Decryption operation: XOR (⊕) the ciphertext with the key

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 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?
  - Counter value
  - Random value
  - Current time
Questions

• Where we can get (true) random numbers?
• Why pseudo-random number is not as good as random number?
• What are the properties of random sequence?
• Can we tell whether the provided sequence is random?
• What happens to data if we XOR it with random data?
• Why brute-force attacks are ineffective in breaking one-time pad?
• Why unbreakable one-time pad is not used in enterprise products?
• How is stream cipher different from one-time pad?
Task: One-Time Pad (OTP) – 3p

Implement 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 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()
str object stores Unicode characters:

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

bytes object stores bytes:

```python
>>> bytes([97,98])
b'ab'
>>> b = b'F\xc5\x8d\xc5\x8d'
>>> b = s.encode()
>>> type(b), len(b)
(<class 'bytes'>, 5)
>>> b[0], b[1], b[2], b[3], b[4]
(70, 197, 141, 197, 141)
>>> b.decode()
'Fōō'
```

```python
>>> import codecs
>>> codecs.encode(b, 'hex')
b'46c58dc58d'
>>> codecs.encode(b, 'base64')
b'RsWNxY0=\n'
>>> codecs.encode(b, 'base64').decode()
'RsWNxY0=\n'
```
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 contents 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

$ 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.
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 nonrequired 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!