MTAT.07.017
Applied Cryptography

Introduction, Randomness, One-Time Pad, Stream Ciphers

University of Tartu

Spring 2014
Who am I

Arnis Paršovs

MSc in Cyber Security
Tallinn University of Technology, 2012

Computer Science PhD student at UT
Crypto courses in UT

- Cryptology I
- Cryptographic protocols
- Cryptology II
- Research seminar in cryptography
- Quantum cryptography
- Topics of mathematics in cryptology
- Applied cryptography
This course

- Practical course
- Implementing low level crypto
- No proofs – just intuition
Course timeline

16 weeks

- Lecture: Thu 12:15 – 14:00
- Practice: Thu 16:15 – 18:00
  - Overlaps with Cryptology I practice session

6 ECTS – 10 hours weekly

- 2 hours in class
- 8 hours on homework (may vary)
Grading

- Homework every week
- Homeworks give 70% of final grade
- Deadlines are strict!
  - May be extended if you have a good excuse
  - Late submission gets 50% of grade
- Exam gives another 30% of final grade
  - Should be easy if you follow the lectures
Programming Environment

Course history:

- 2010: C, C++, Java
- 2011: Java
- 2012: Java
- 2013: Python
- 2014: Python

Reason: focus on cryptography, not language

Test environment: Ubuntu 13.10, Python 2.7.x
• Create a private Bitbucket repository and grant me ’read’ privileges:
  https://bitbucket.org/appcrypto/2014/src/tip/setup/
  • Send me an e-mail if your identity is not obvious from your Bitbucket account name.

• Homework templates will be published at course repository:  https://bitbucket.org/appcrypto/2014/

• Homework deadline – beginning of next lecture (next Thursday 14:15 if not stated otherwise)
  • I might check your solution and give a feedback before deadline
  • Feedback will be given using Bitbucket code comment feature
Python Cheat Sheet


```python
>>> "abraca" + 'dabra'
'abracadabra'
>>> "abraca" + "\x31\x05\x77"
'abraca1\x05w'
>>> for character in "foo":
...     print "char=%s" % (character)
char=f
c char=o
c char=o
>>> "abraca"[2:5]
'rac'
>>> "abracadabra".encode('hex')
'6162726163616461627261'
>>> "abracadabra".encode('base64')
'YWJyYWNhZGFicmE=
' >>> "abracadabra".encode('base64').decode('base64')
'abracadabra'
```
Randomness

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

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

- Where can we get random numbers?

- Can we tell whether the sequence is random?
  
  \[ \ldots 41592653589 \ldots \]
  
  \[ 3.141592653589793 \ldots \]
Pseudo-Random Number Generator (PRNG)

Deterministic algorithm that produces endless stream of numbers which are indistinguishable from truly random. PRNG is initialized using (hopefully) random 'seed' value.

Linux /dev/random implementation:

In Python: `os.urandom(n)`
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

|   | <NULL> | <SPC> | @ | ` | Â | † | ç | i | Ņ | Š | % | Æ | è | ™ | Å | ± |
|---|------|------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 |     |     | 64 | 96 | 128 | 160 | 192 | 224 |
| 1 | <SOH> | ! | 65 | a | 129 | 161 | 193 | 225 |
| 2 | <STX> | " | 66 | b | 130 | 162 | 194 | 226 |
| 3 | <ETX> | # | 67 | c | 131 | 163 | 195 | 227 |
| 4 | <EOT> | $ | 68 | d | 132 | 164 | 196 | 228 |
| 5 | <ENQ> | % | 69 | e | 133 | 165 | 197 | 229 |
| 6 | <ACK> | & | 70 | f | 134 | 166 | 198 | 230 |
| 7 | <BEL> | ' | 71 | g | 135 | 167 | 199 | 231 |
| 8 | <BS> | ( | 72 | h | 136 | 168 | 200 | 232 |
| 9 | <TAB> | ) | 73 | i | 137 | 169 | 201 | 233 |
| 10 | <LF> | * | 74 | j | 138 | 170 | 202 | 234 |
| 11 | <VT> | + | 75 | k | 139 | 171 | 203 | 235 |
| 12 | <FF> | , | 76 | l | 140 | 172 | 204 | 236 |
| 13 | <CR> | - | 77 | m | 141 | 173 | 205 | 237 |
| 14 | <SO> | . | 78 | n | 142 | 174 | 206 | 238 |
| 15 | <SI> | / | 79 | o | 143 | 175 | 207 | 239 |
| 16 | <DLE> | 0 | 80 | p | 112 | 144 | 176 | 208 |
| 17 | <DC1> | 1 | 81 | q | 113 | 145 | 177 | 209 |
| 18 | <DC2> | 2 | 82 | r | 114 | 146 | 178 | 210 |
| 19 | <DC3> | 3 | 83 | s | 115 | 147 | 179 | 211 |
| 20 | <DC4> | 4 | 84 | t | 116 | 148 | 180 | 212 |
| 21 | <NAK> | 5 | 85 | u | 117 | 149 | 181 | 213 |
| 22 | <SYN> | 6 | 86 | v | 118 | 150 | 182 | 214 |
| 23 | <ETB> | 7 | 87 | w | 119 | 151 | 183 | 215 |
| 24 | <CAN> | 8 | 88 | x | 120 | 152 | 184 | 216 |
| 25 | <EM> | 9 | 89 | y | 121 | 153 | 185 | 217 |
| 26 | <SUB> | ; | 90 | z | 122 | 154 | 186 | 218 |
| 27 | <ESC> | | 91 | { | 123 | 155 | 187 | 219 |
| 28 | <FS> | | 92 | | 124 | 156 | 188 | 220 |
| 29 | <GS> | | 93 | | 125 | 157 | 189 | 221 |
| 30 | <RS> | | 94 | | 126 | 158 | 190 | 222 |
| 31 | <US> | | 95 | | 127 | 159 | 191 | 223 |

# Hexadecimal (Base16) Encoding

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

- One hex character can represent 4 bits
- Two hex characters needed to represent a byte

\[ 2E = 0010 \ 1110 \]

\[
>>> 46
\]

\[\begin{align*}
46 \\
>>> 0x2e \\
46 \\
>>> 0x2e + 1 \\
47
\end{align*}\]
Python

Python’s `str` data type can store any byte:

```python
>>> s = 'Abc\x00\x61'
>>> len(s)
5
>>> s[0], s[1], s[2], s[3], s[4]
('A', 'b', 'c', '\x00', 'a')
>>> ord(s[0])
65
>>> chr(65)
'A'
```

- `ord()` can be used to convert byte to integer
- `chr()` can be used to convert integer to byte
Base64 encoding

bn+ITbj/TRwcSAwT8CZnFZN0me5/AGdFIGNLBP07Nc07T6XTpsTw0QxnM++9xJXKkEEcaEn2Vo9MiAVPVUR5PsFGKZbL7coPRdHD058RokCF4aizWv6+Dqg0lsXsmXliWusn0Q==

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

AND:
- extract partition of bit string

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)

OR:
- set specific bits

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)

XOR:
- flip specific bits

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)

Shift:
- shift and pad with 0

0 0 1 1 1 1 0 0
-------------
0 0 0 0 1 1 1 1 (right shift by two)
Bitwise operations: 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 operations: OR

- Set specific bits

Example:

\[
\begin{array}{cccccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\hline
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0
\end{array}
\]

Python:

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

• Flip specific bits

Example:

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

0 0 1 1 1 0 1 0 (XOR)

Python:

```python
>>> 60 ^ 6
58
```
Bitwise operations: 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
Bytes to Integer

>>> s = 'abC'
>>> i = ord(s[0])
>>> i
97
>>> bin(i)
'0b1100001'
>>> i = i << 8
>>> bin(i)
'0b1100001000000000'
>>> i = i | ord(s[1])
>>> bin(i)
'0b110000101100010'
>>> i = i << 8
>>> bin(i)
'0b110000101100010000000000'
>>> i = i | ord(s[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
- ...

Endianness (byte order):

- Big Endian (network byte order)
  - MSB left-most byte
- Big Endian order used in most standards
One-Time Pad (OTP)

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

Information-theoretically secure, if:

- Key (one-time pad) is truly random
- Key has the same length as plaintext
- Key is never reused
Stream Ciphers

- Key generation: key is a 'seed' for PRNG
- Encryption operation: XOR (⊕) the key with plaintext
- Decryption operation: XOR (⊕) the key with ciphertext

Why is it less secure than one-time pad?
- Malleability attacks
- The same keystream must never be reused!

plaintext1 ⊕ keystream = ciphertext1
plaintext2 ⊕ keystream = ciphertext2 ⊕ plaintext2 = keystream
keystream ⊕ ciphertext1 = plaintext1
Stream Ciphers

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

- The same *nonce* must never be reused!
- How to generate *nonce*?
  - Random value
  - Counter value
  - Current time
Task: One-Time Pad (OTP)

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 “otp.py” to your repository
#!/usr/bin/env python
import os, sys # do not use any other imports/libraries
# specify here how much time your solution required

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

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

def usage():
    print "Usage:
    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()
Task: One-Time Pad (OTP)

- **Encrypter:**
  - Read the file contents into byte string (e.g., `s = open('file.txt').read()`)
  - Convert plaintext byte string to one big integer
  - Obtain random byte string the same length as plaintext (use `os.urandom()`)
  - Convert random byte string to one big integer
  - XOR plaintext integer and key integer (*please, use this approach*)
  - Save the key (one-time pad) and XOR’ed result (ciphertext) to file:
    - Convert key integer to byte string:
      - Use bit masking and left shift
      - Store in network byte order

- **Decrypter:**
  - Perform the operations in reverse order
$ 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.key file.enc fileorig.plain
$ hexdump -C fileorig.plain
  00000000  00 00 61 62 43 00 |..abC.|
Please!

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

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

- Give a feedback about parts that were hard to grasp or implement

- Make a note if something was wrong on slides or you have an idea for improvement

Thank you!