MTAT.07.017
Applied Cryptography

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

University of Tartu

Spring 2018

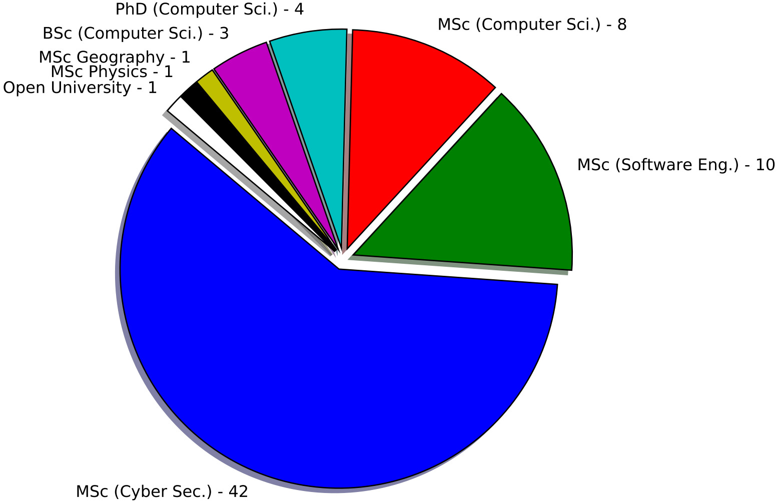
Who am I?

Arnis Paršovs

MSc in Cyber Security
Tallinn University of Technology, 2012

Computer Science PhD student at UT

Who are you?


PhD (Computer Sci.) - 4
BSc (Computer Sci.) - 3
MSc Geography - 1
MSc Physics - 1
Open University - 1
MSc (Computer Sci.) - 8
MSc (Software Eng.) - 10
MSc (Cyber Sec.) - 42

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
- Learn by implementing
- No proofs – just intuition

Course timeline

16 weeks

- Lecture: Thu 10:15 – 12:00 (room 405)
- Practice: Thu 18:15 – 20:00 (room 206)

6 ECTS – 10 hours weekly

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

Grading

- Homework every week
- Homeworks give maximum 70% of final grade
  - Bonus points help to achieve the maximum
- Deadlines are strict!
  - Homework deadline – beginning of next lecture
  - May be extended if you have a good excuse
  - Late submission gets 50% of grade
  - Homeworks submitted later than 1 week after the deadline are not accepted
- 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
- ...
- 2018: Python

Test environment: Ubuntu 16.04.3, Python 2.7.x

Python packages from Ubuntu package repository (not pip)

Homework submission

- Create a private Bitbucket repository and grant me 'read' privileges:

  `https://bitbucket.org/appcrypto/2018/src/master/setup/`

- Homework templates will be published at course repository: `https://bitbucket.org/appcrypto/2018/`
- Feedback will be given using Bitbucket's code comment feature
- Do not look on homework solutions of others!
  - Plagiarism cases will be handled in accordance with UT Plagiarism Policy

Academic Fraud

- It is 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 invididually!
- 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.

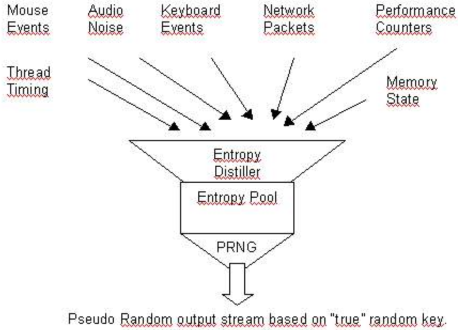
Randomness

- Why do we need randomness in real life?
- Why do we need randomness in crypto?
  - For keys, nonces, etc.
- What is random sequence?
  - Sequence of numbers that does 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 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.
PRNG is initialized using (hopefully) random 'seed' value.

Linux /dev/urandom implementation:



- Known part of the input does not allow to predict 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 the sequence is random?

  `...41592653589...`
  `3.141592653589793...`

  `...000000......`

  - Statistical randomness tests

Bits and Bytes

Bit string:

10001000011

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

Most significant bit (msb) – left-most bit

Bytes - 8-bit collections (0-255)

Byte - basic addressable element

ASCII Table

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	∆	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	“	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	”	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	‘	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	’	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	⁄	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	˚
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	ﬁ	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	ﬂ	255	ˇ

http://www.asciitable.com/

Hexadecimal (Base16) Encoding

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

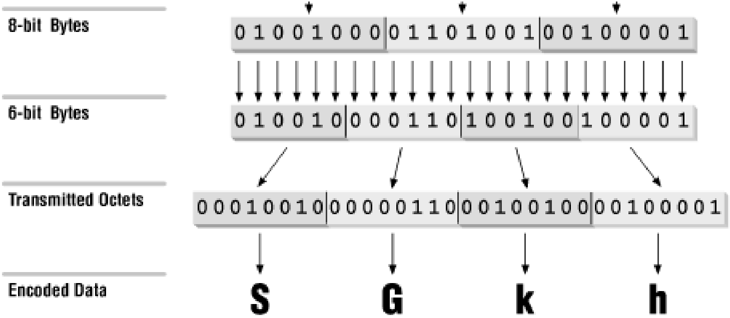
- One hex symbol represents 4 bits
- Two hex sumbols needed to represent a byte

`2E = 0010 1110`

Base64 encoding

bn+ITbj/TRwcSAwT8CZnFZN0me5/AGdFIGNLBPPo
7Nc07T6XTpsTw0QxnM++9xJXKkEEcaEn2Vo9MiAV
PVUR5PsFGKZbL7coPRdHD058RokCF4aizWv6+Dqg
0lsXsmXliWusn0Q==

- Can 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)
---------------
0 0 0 0 0 1 0 0 (AND)
```

```
>>> 60 & 6
4
```

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)
```

```
>>> 60 | 6
62
```

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)
```

```
>>> 60 ^ 6
58
```

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)
```

```
>>> 60 >> 2
15
```

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:

```
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:

```
>>> 60 | 6
62
```

Bitwise operations: XOR

- Flip specific bits

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)
```

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)
```

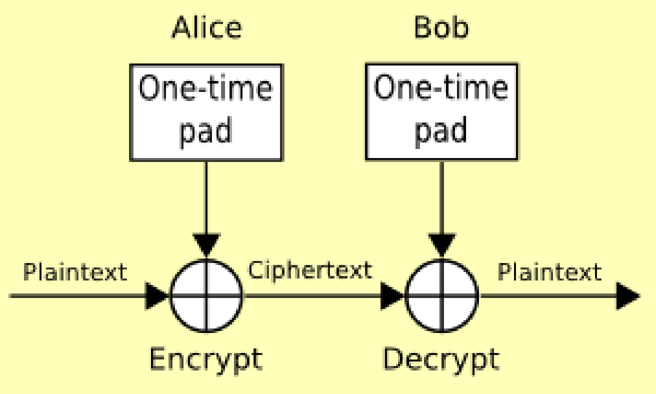
Python:

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

- Fast multiplication and division by 2

One-Time Pad (OTP)

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



One-Time Pad (OTP)

Information-theoretically secure (unbreakable), if:

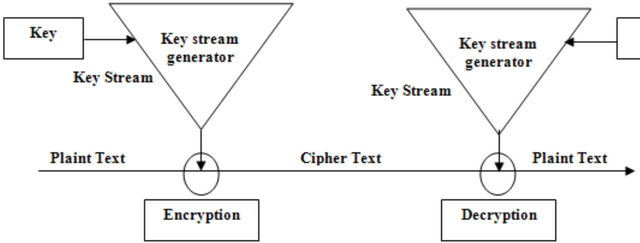
- Key (one-time pad) is truly random
- Key is never reused

  plaintext1 ⊕ key = ciphertext1
  plaintext2 ⊕ key = ciphertext2 ⊕ plaintext2 = key
  key ⊕ ciphertext1 = plaintext1

- Not used in practice

Stream Cipher

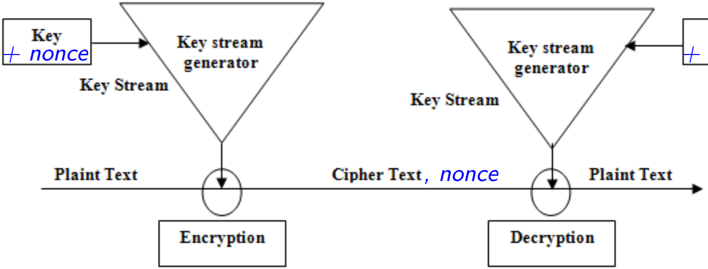
- Key generation: a small key "seeds" the PRNG
- Encryption operation: XOR ($\oplus$) the key with plaintext
- Decryption operation: XOR ($\oplus$) the key with ciphertext



- 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 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-froce 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)

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.plai
```

- Commit "01/otp.py" to your repository

```
$ git add 01/otp.py
$ git commit -m "01 homework solution" 01/otp.py
$ git push
```

Task: Template

```
#!/usr/bin/env python
import os, sys       # do not use any other imports/libraries
# took x.y hours (please 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:"
    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 Cheat Sheet

```
>>> "abraca" + 'dabra'
'abracadabra'

>>> for character in "foo":
...     print "char=%s" % (character)
char=f
char=o
char=o

>>> "abraca"[2:5]
'rac'

>>> "abracadabra".encode('hex')
'6162726163616461627261'
>>> "abracadabra".encode('base64')
'YWJyYWNhZGFicmE=\n'
>>> "abracadabra".encode('base64').decode('base64')
'abracadabra'
```

Python

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

```
>>> 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

Python: Bytes to Integer

```
>>> s = 'abC'
>>> i = ord(s[0])
>>> i
97
>>> bin(i)
'0b1100001'
>>> i = i << 8
>>> bin(i)
'0b110000100000000'
>>> i = i | ord(s[1])
>>> bin(i)
'0b110000101100010'
>>> i = i << 8
>>> bin(i)
'0b11000010110001000000000'
>>> 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
- ...

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
  - Once more: use bitwise operations!
    - Banned: functions `bin()`, `str()`, `int()`, `bytearray()` 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.|
```

Value 0 can be encoded as empty file or file full of zero bytes.

Please!

- Do not waste your time on input validation
- 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 your 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
- Do not leave any unrequired debugging output in your solution
- Commit the solution to the main branch of your repository with the filename required

Thank you!