MTAT.07.017
Applied Cryptography

Randomness, PRNG,
One-Time Pad, Stream Cipher

University of Tartu

Spring 2021

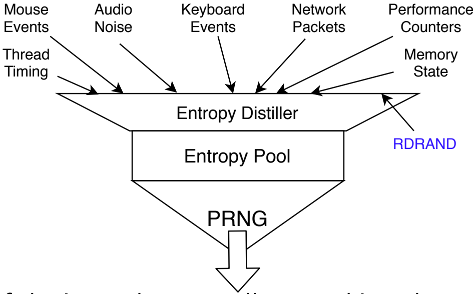
Randomness

- 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...

  ...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)

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

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

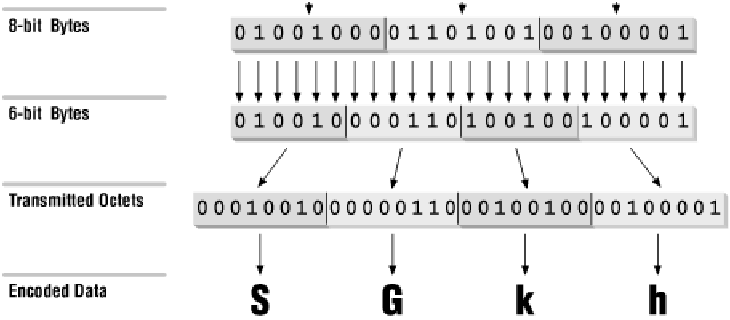
- One hex symbol represents 4 bits
- Two hex symbols needed to represent a byte

`2E = 0010 1110`

Base64 encoding

bn+ITbj/TRwcSAwT8CZnFZN0me5/AGdFIGNLBPPo7Nc07T6XTpsTw0Q
xnM++9xJXKkEEcaEn2Vo9MiAVPVUR5PsFGKZbL7coPRdHD058RokCF4
aizWv6+Dqg0lsXsmXliWusn0Q==

- 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 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:

```
>>> 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 1 1 1 1 (right shift by two)
```

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

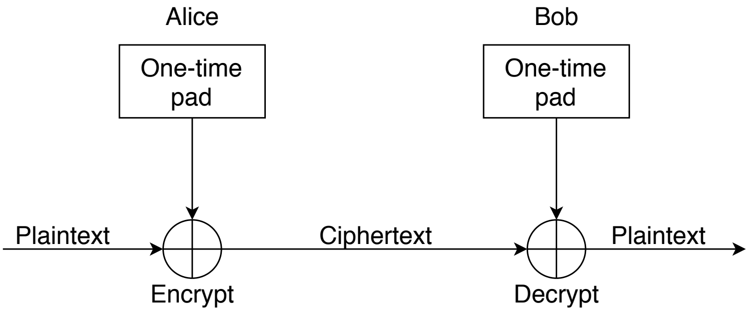
Python:

```
>>> 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



One-Time Pad (OTP)

Information-theoretically secure (unbreakable), if:

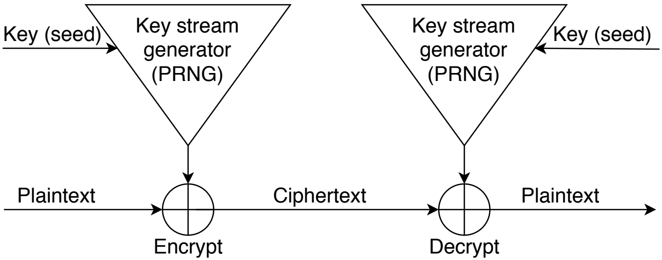
- Key (one-time pad) is truly random
- Key is never reused

$$\text{plaintext1} \oplus \text{key} = \text{ciphertext1}$$
$$\text{plaintext2} \oplus \text{key} = \text{ciphertext2} \oplus \text{plaintext2} = \text{key}$$
$$\text{key} \oplus \text{ciphertext1} = \text{plaintext1}$$

- Not used in practice

Stream cipher

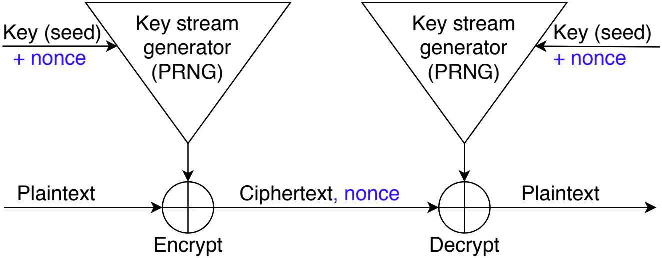
- Key generation: a small key "seeds" the PRNG to generate keystream
- Encryption operation: XOR ($\oplus$) the plaintext with the keystream
- Decryption operation: XOR ($\oplus$) the ciphertext with the keystream



- 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
  ```

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:

```
>>> s = 'Fōō'
>>> type(s), len(s)
(<class 'str'>, 3)
>>> s[0], s[1], s[2]
('F', 'ō', 'ō')
>>> b = s.encode('utf8')
```

`bytes` object stores bytes:

```
>>> b
b'F\xc5\x8d\xc5\x8d'
>>> type(b), len(b)
(<class 'bytes'>, 5)
>>> b.decode('utf8')
'Fōō'
```

```
>>> 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\n'
>>> codecs.encode(b, 'hex')
b'616243'
>>> codecs.encode(b, 'hex').decode('ascii')
'616243'
>>> b.hex()
'616243'
```

Python: bytes to integer

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

```
$ 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!