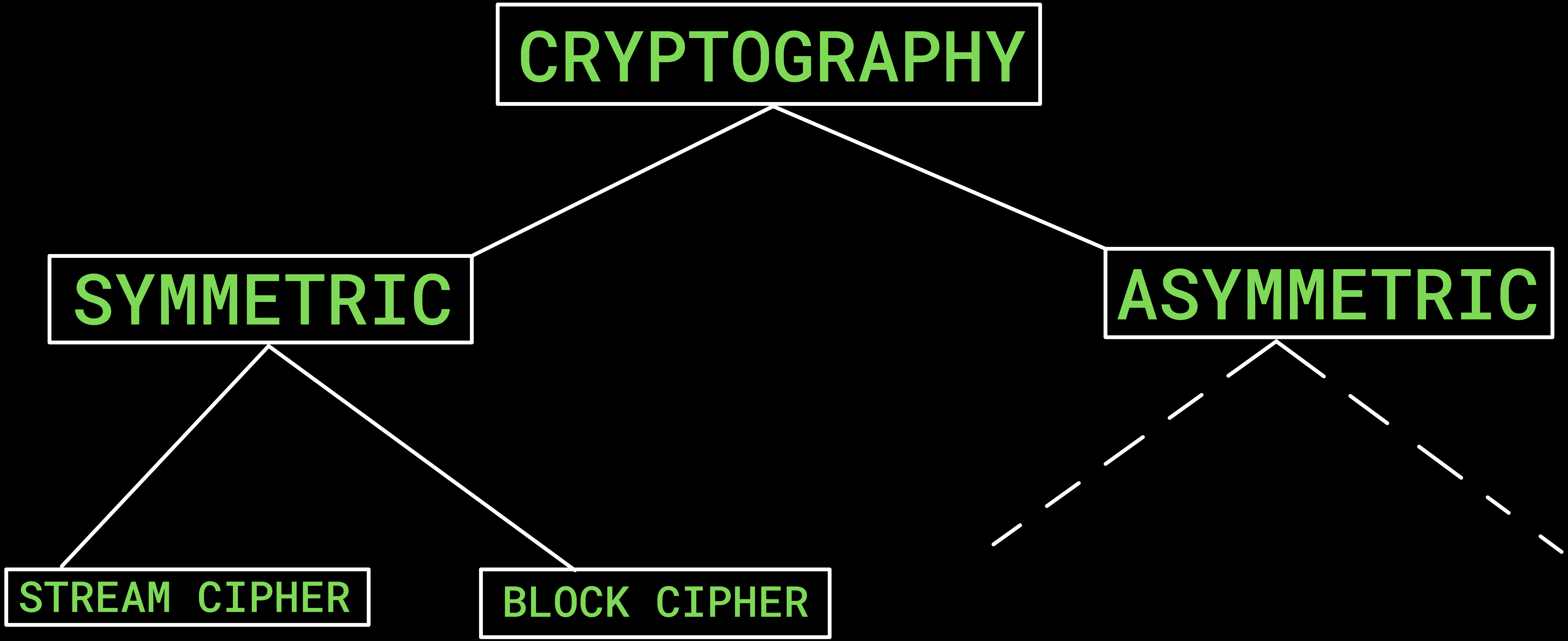
CRYPTOGRAPHY

PART II

MODERN CRYPTOGRAPHY

- Modern-day cryptography uses bits and bytes, and runs on our computers, allowing us to encrypt and send all kinds of things!

- Comparatively speaking, modern cryptography allows us to design and use cryptosystems that are much more secure than classical era .


CRYPTOGRAPHY
SYMMETRIC
ASYMMETRIC
STREAM CIPHER
BLOCK CIPHER

Symmetric cryptography uses the same key for encryption as for decryption. In order to communicate, two parties simply share the same secret key, and each can encrypt / decrypt the same messages.

Asymmetric cryptography (or public-key cryptography) uses different keys for encryption and decryption. The key to encrypt is “public”, so anyone can encrypt messages to owners of the private key, which is required to decrypt.

BITWISE OPERATIONS

- You might be familiar with the following logical operations
    - AND (&)
    - OR (|)
    - NEGATION (~)
    - XOR (^)

- They work on bits too!

TRUTH TABLE

X	Y	X ^ Y
0	0	0
0	1	1
1	0	1
1	1	0

# We simply treat 1 as “true” and “0” as “false”.

BITWISE OPERATIONS TRUTH TABLE

You might be familiar with the following logical operations:

AND (&)

X	Y	X & Y
0	0	
0	1	
1	0	
1	1	

OR (|)

X	Y	X | Y
0	0	
0	1	
1	0	
1	1	

XOR (^)

X	Y	X & Y
0	0	
0	1	
1	0	
1	1	

BITWISE OPERATIONS TRUTH TABLE

You might be familiar with the following logical operations:

AND (&)

X	Y	X & Y
0	0	0
0	1	0
1	0	0
1	1	1

OR (|)

X	Y	X | Y
0	0	
0	1	
1	0	
1	1	

XOR (^)

X	Y	X & Y
0	0	
0	1	
1	0	
1	1	

BITWISE OPERATIONS TRUTH TABLE

You might be familiar with the following logical operations:

AND (&)

X	Y	X & Y
0	0	0
0	1	0
1	0	0
1	1	1

OR (|)

X	Y	X | Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR (^)

X	Y	X & Y
0	0	
0	1	
1	0	
1	1	

BITWISE OPERATIONS TRUTH TABLE

You might be familiar with the following logical operations:

AND (&)

X	Y	X & Y
0	0	0
0	1	0
1	0	0
1	1	1

OR (|)

X	Y	X | Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR (^)

X	Y	X & Y
0	0	0
0	1	1
1	0	1
1	1	0

BITWISE OPERATIONS

- All of the above operations operate on two values, except for negation, which simply changes the truthiness of its one argument

- All of these operations associate and commute with themselves.

XOR with Constant

- For key, choose a single byte constant : K
- Then, encrypt each byte of plaintext with :

$$\boxed{C = M \wedge K}$$

- Decrypt ciphertext bytes with :

$$\boxed{M = C \wedge K}$$

How secure?

XOR with Constant

Not very ! Because the core problem is that the keyspace is tiny there's only 256 possible keys.

# let's see the python implementation

Repeated XOR

For key, choose several bytes :

K = K0 K1 K2 … Kn-1

Then, repeat our key to match the length of our plaintext or ciphertext.

Line up the plaintext / ciphertext with the repeated key, and XOR each byte with each byte.

```
rep(K) = K0 K1 K2  … Kn-1 K0 K1 …

M  = M0 M1 M2  … Mn-1    Mn     Mn+1
```

C = (Ci); Ci = Mi ^ Ki%n

How secure?
(Repeated Xor)

BAD SECURITY

# If key short enough

 - you can find key with known plaintext attacks, crib dragging, or frequency analysis.

- If key too long, sharing the key becomes difficult.

ONE TIME PAD

# using OTP

ONE TIME PAD

→ Randomly generated text

- Which is of the same length as the text you want to encrypt.

Typically it’s generated using :

RNG = Random Number Generator

# The more it’s ‘random’ the more it’s ‘secure/safe’

ONE TIME PAD (example)

Plaintext :	H	e	l	l	o
Ascii equivalent:	72	101	108	108	111

Ascii equivalent:	F	‘	:	@	X
RNG generated : Pad	102	39	58	64	88

ONE TIME PAD (example)

72 ^ 102 = 46	→	.
101 ^ 39 = 66	→	B
108 ^ 58 = 86	→	V
108 ^ 64 = 44	→	,
111 ^ 88 = 55	→	7

Ascii equivalent

How secure?

One time pad has perfect secrecy since key is exactly same as the length of message which is encrypted and key is truly random.

CONTACT US

@nitdgplug

/nitdgplug

@lugnitdgp

DEV @nitdgplug

GNU/Linux Users' Group NIT-Dgp

nitdlug@gmail.com

GNU/Linux Users' Group NIT-Dgp

https://nitdgplug.org/