CS 419: Computer Security

Recitation: week of 2020-10-19

Project 3 Discussion

TA: Shuo Zhang
Paul Krzyzanowski

Assignment 11 (Project 3)

- **This assignment has three parts**
- **This is an individual assignment**
- **Goal: implement three simple ciphers**
  These will include using:
  - Polyalphabetic cipher using table-driven substitutions
  - Stream cipher using
    - A linear congruential pseudorandom keystream generator
    - Simple password hashing for seed generation
  - Block cipher derived from the stream cipher
    - Keystream-based byte swapping
    - Cipher block chaining (CBC) for diffusion

Environment

- **You should be able implement this on any platform**
  - You may use Go, Python, Java, C, C++
- **But you are responsible to make sure it works on the Rutgers iLab machines with no extra software**

Part 1: Binary Vigenère Cipher

Review: Vigenère polyalphabetic cipher

- **Repeat keyword over text: (e.g., key=FACE)**

  Keystream: `FA CEF ACE FACEF ....`

  Plaintext: `MY CAT HAS FLEAS`

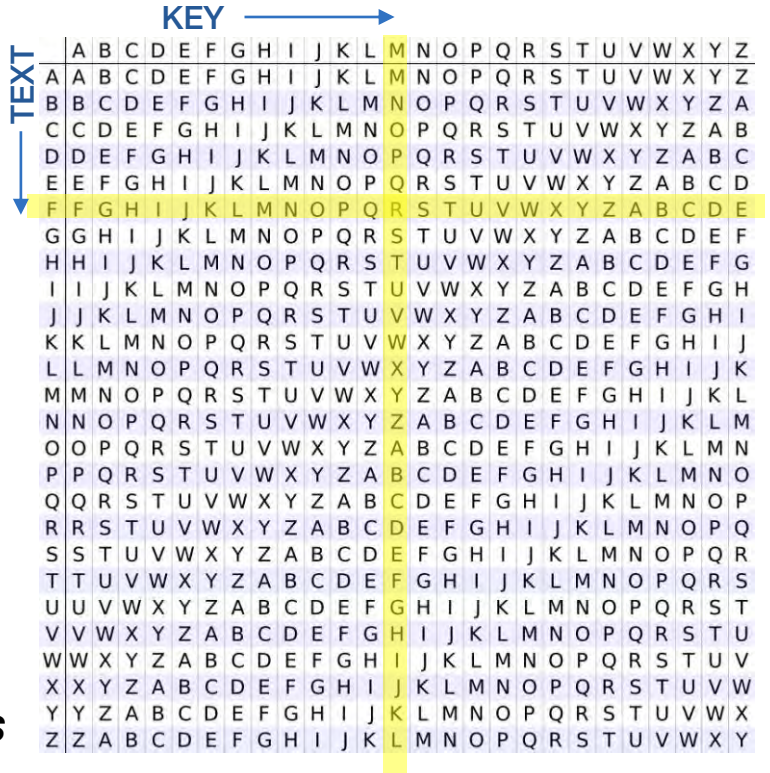
- **Encrypt: find intersection:**

  row = keystream letter

  column = plaintext (message) letter

- **Decrypt: find column**
  - Row = keystream letter, search for ciphertext
  - Column heading = plaintext letter

Message is encrypted with as many substitution ciphers as there are unique letters in the keyword

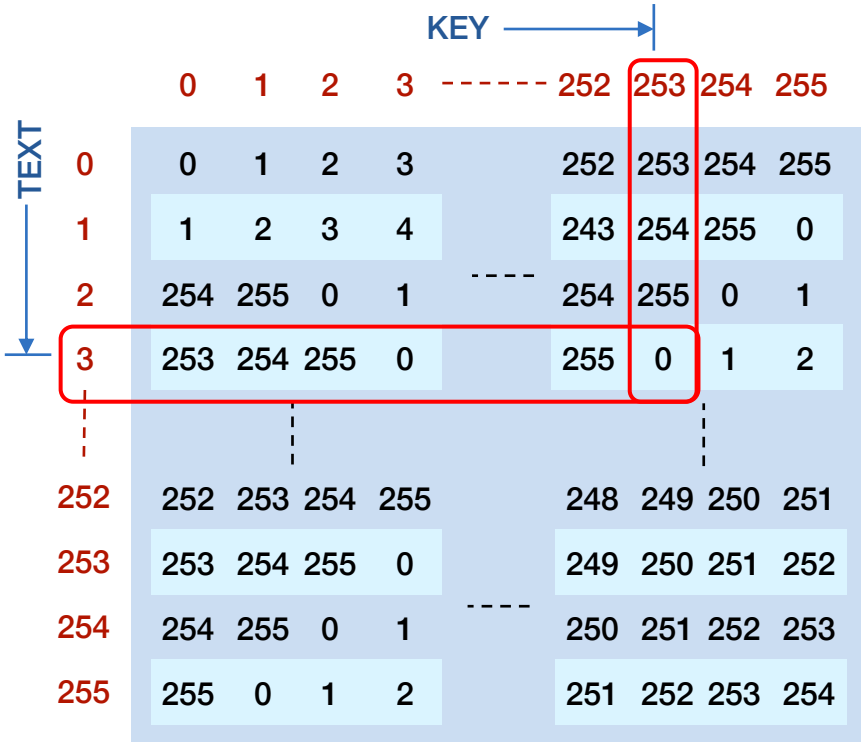


Part 1: Binary Vigenère Cipher

- **The Vigenère cipher was designed for pencil-and-paper cryptography**
  - It's designed for use with text only
- **You will modify the cipher to work with binary data**
  - Any file
  - Arbitrary binary key file

Binary Vigenère Cipher

- **Instead of a text-based table we use a byte table**
  - 256 rows & 256 columns
- **Arbitrary plaintext file data**
  - Not just text
- **Arbitrary key**
  - Data stored in a keyfile
- **Compute ciphertext**
  - Column = next key byte
  - Row = next plaintext byte
  - Ciphertext = intersection

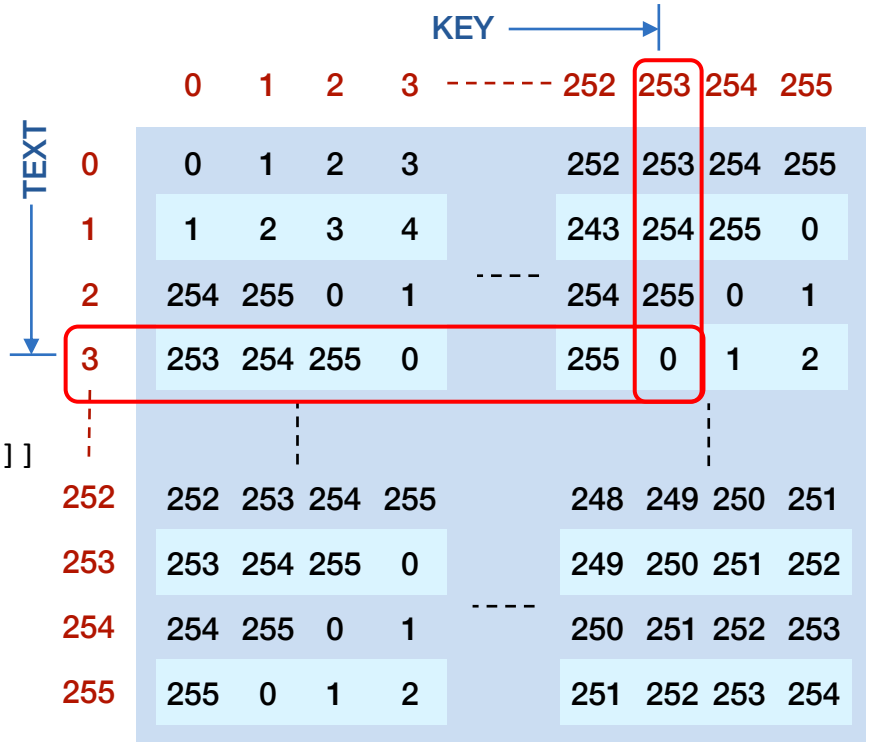


Binary Vigenère Cipher

- **Use a repeating key**
  - Just as in the Vigenère cipher
- **Wrap back to the start of the key when you run out of key data**

To encrypt a byte of plaintext:

1. **Look up ciphertext**
   ```
   ciphertext[n] =
   table[row=message[n]][column=ciphertext[i]]
   ```
2. **Go to the next position of plaintext**
   ```
   n = n+1
   ```
3. **Go to the next position of the key**
   ```
   i = (i+1) % length(ciphertext)
   ```



Implementation

- **Create two programs – one to encrypt and another to decrypt**
  - `vencrypt keyfile message ciphertext`
  - `vdecrypt keyfile ciphertext plaintext`

Implementation Hints

- **Test thoroughly!**
  - Come up with various test cases
  - A key with bytes of 0 will always produce plaintext
  - A key with bytes of 1 will produce shifted data (e.g., “ABC” ⇒ “BCD”)
  - Printing input & output of data (as hex #s, for example) can help you test
- **Hints**
  - The *od* command dumps binary data:

    `od –t xC keyfile` dumps contents of `keyfile` as hex bytes
  - If you think about the problem, you don’t need a table
    - The entire encryption can be one `while` loop with one line of code within it!

Validate your program

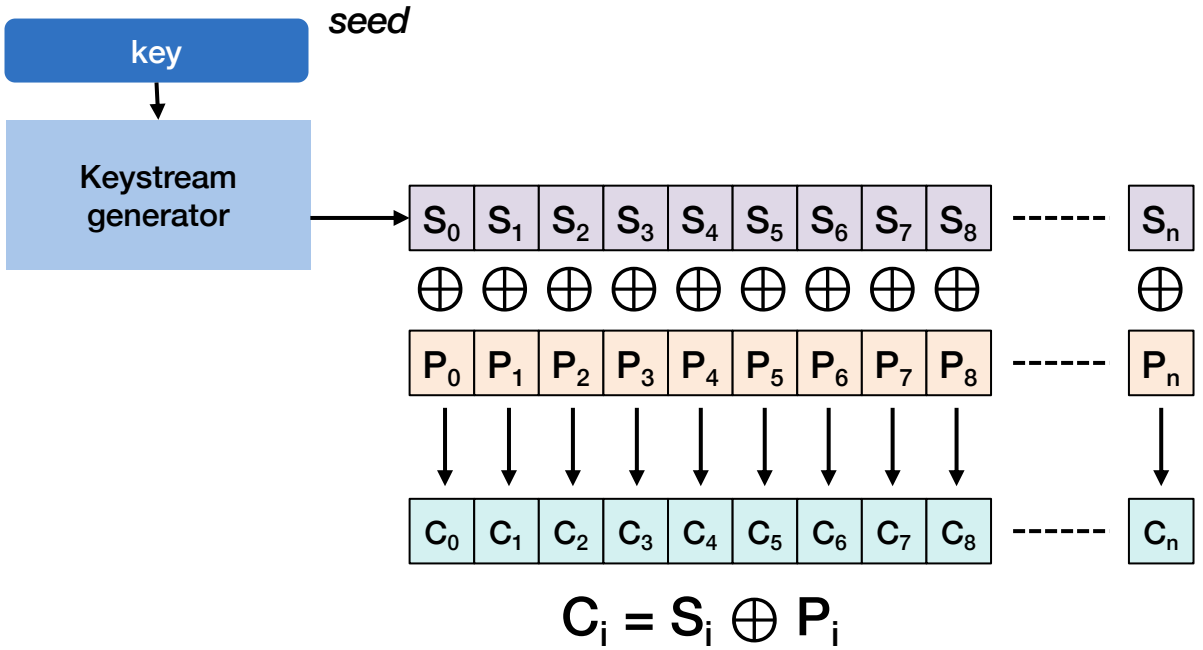
- **You will be provided with:**
  - Reference versions of the programs: *vencrypt*, *vdecrypt*
  - Sample keys
  - Small sample content
- **Your program should produce identical output**

Part 2: Stream Cipher

Stream ciphers

Key stream generator produces a sequence of pseudo-random bytes

Simulates a one-time pad



Keystream Generator

- **Stream ciphers work by creating a key sequence that is as long as the message**
- **They do this by using a keystream generator**
  - This is a pseudorandom number generator
  - We want the sequence to have a statistically random distribution
  - But it needs to be reproducible so we can get the same encryption & decryption if we use the same key
- **In this assignment, we will use a very simple pseudorandom number generator**

Linear congruential keystream generator

- **The cipher will use a linear congruential generator**
- **One of the best-known pseudorandom number generators**
- **Each value is f(previous value):**

$$X_{n+1} = (aX_n + c) \bmod m$$

- **Where**
  - X_{n+1} = next pseudorandom number
  - X_n = last pseudorandom number
  - m = modulus – we will use 256 (2^8) to get a byte stream
  - a, c = magic parameters, some produce better data than others
    - a = 1103515245
    - c = 12345

These are used by ANSI C, C90 ,C99, etc.
See the Wikipedia article

Seed: hash

- **We need a seed for the pseudorandom number generator**
- **This is just a number**
- **Instead of asking users to enter a number, we will use a password string:**
  - seed = hash(password)
- **For this assignment, we will not use a cryptographic hash function but one that is trivial to implement:**
  - sbdm – used in gawk, sdbm database, Berkeley DB, etc

```
static unsigned long
sdbm(unsigned char *str) {
     unsigned long hash = 0;
     int c;
     while (c = *str++)
          hash = c + (hash << 6) + (hash << 16) - hash;
     return hash;
}
```

Test your keystream generator

- **Before implementing the cipher, test your seed generation and keystream against the reference implementation provided**
  - Cipher implementations need to work across different platforms and different implementations
- **You are provided with a program called prand-test**

```
$ ./prand-test
usage: ./prand-test [-p password | -s seed] [-n num]
```

Test your keystream generator

Test password → seed

```
$ ./prand-test -p monkey01
using seed=5423267027848090132 from password="monkey01"
```

Test keystream generator from seed

```
$ ./prand-test -s 123 -n 5
using seed=123
152
241
214
87
68
```

Test your keystream generator

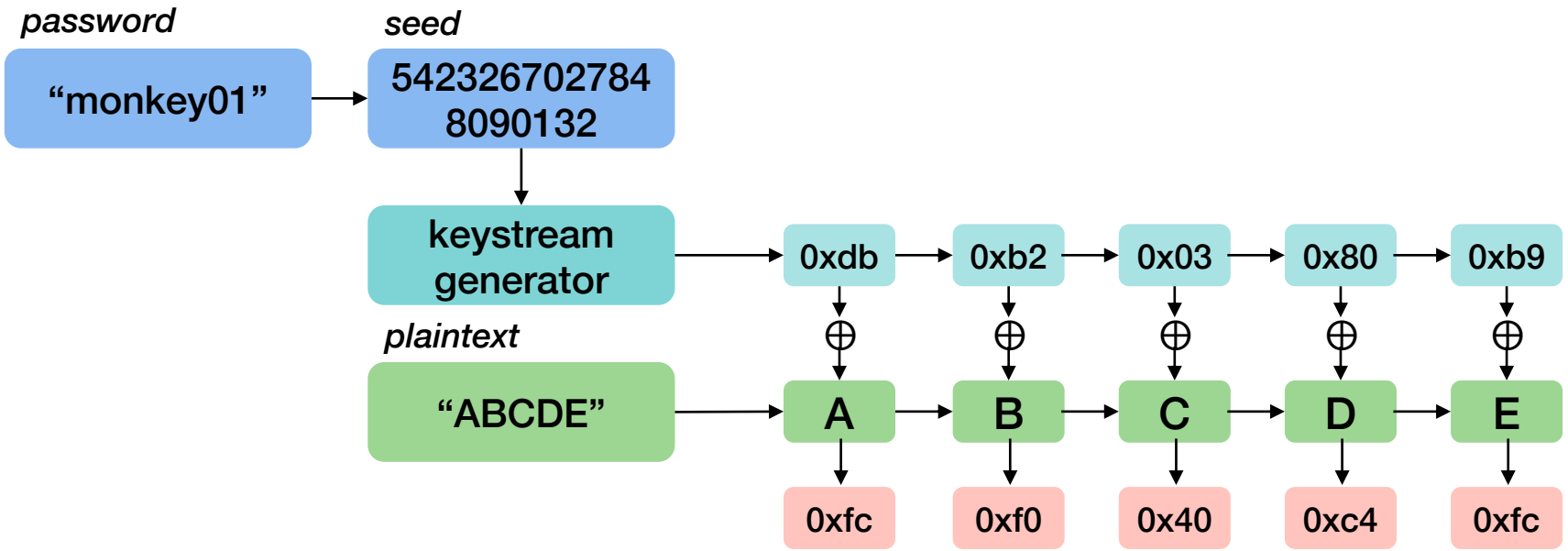
Test keystream generator from password

```
$ ./prand-test -p monkey01 -n 10
using seed=5423267027848090132 from password="monkey01"
189
178
3
128
185
254
95
172
117
10
```

The program

Write the program

```
scrypt password plaintextfile ciphertextfile
```



Validate your program

- **You will be provided with:**
  - Reference versions of the program: *scrypt*
  - Small sample content
- **Your program should produce identical output**
- **Note: there is no encrypt/decrypt**
  - XOR of the ciphertext with the same keystream produces plaintext

```
scrypt password plaintextfile ciphertextfile
scrypt password ciphertextfile plaintextfile
```

Part 3: Block Cipher With CBC

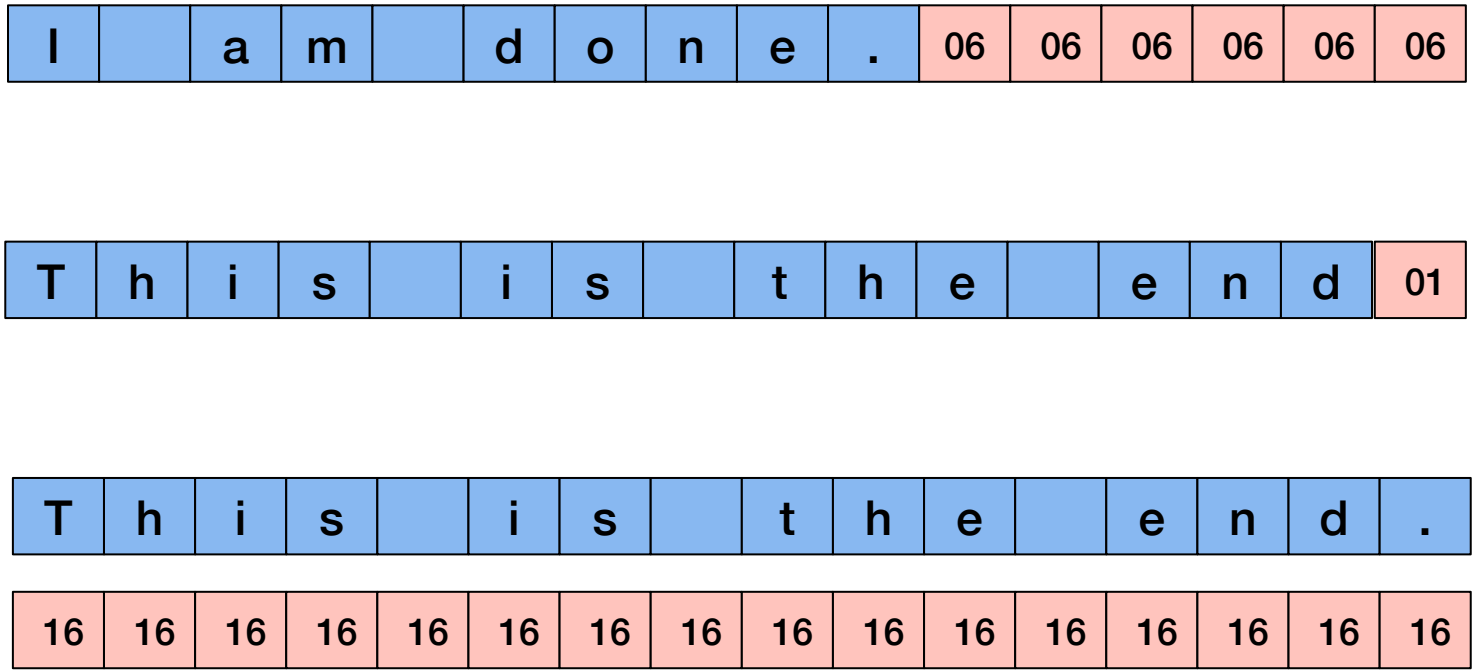
Simple Block Cipher

- **Symmetric block ciphers apply an SP network in multiple rounds**
  - This provides confusion & diffusion within the block
- **Cipher Block Chaining (CBC)**
  - Adds diffusion across multiple blocks
- **We will take a different approach and turn the stream cipher from Part 2 into a simple block cipher**
  - Read data in 16-byte blocks (128 bits)
  - Apply CBC (adds diffusion)
  - Exchange random pairs of bytes in the block (enhances confusion)
  - XOR result with the keystream (this adds confusion)

Padding

- **Block ciphers work on a block of data (16 bytes for us)**
- **The last part of a file might be a partial block**
  - We will add padding at the end … and remove it when decrypting
- **Padding: 1-16 extra bytes**
  - If the file was an even # of blocks, padding adds an extra block
  - Otherwise, if just fills up the block
  - Each byte of the padding is simply the # of bytes of padding that were added

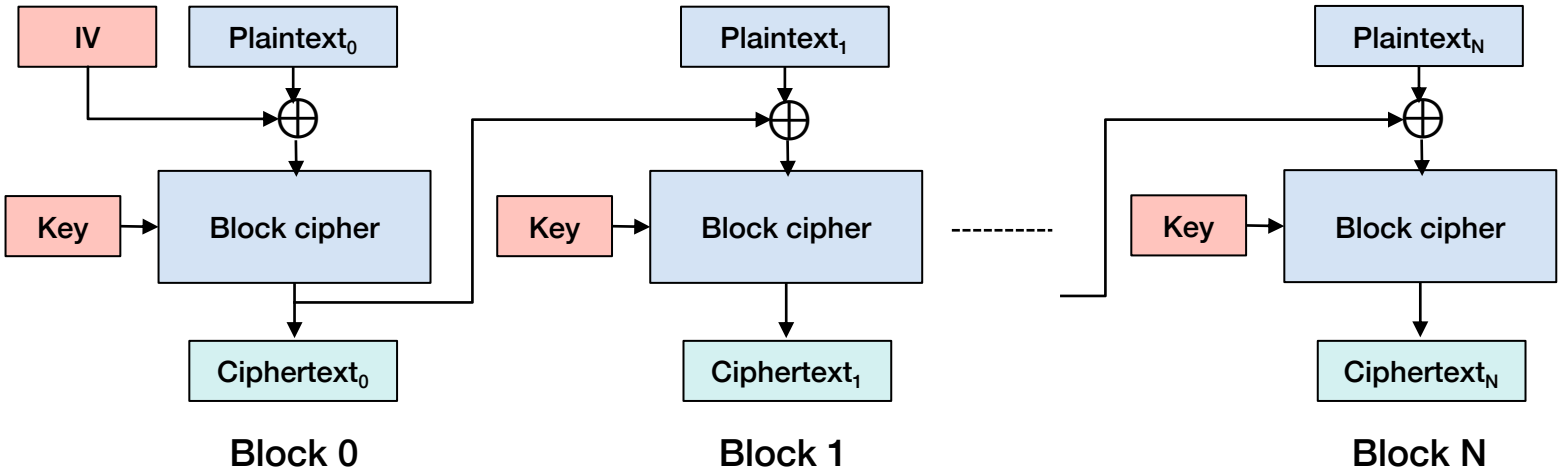
Padding Examples



Reminder: Cipher Block Chaining (CBC) mode

- Random initialization vector (IV) = bunch of *k* random bits
- Exclusive-or with first plaintext block – then encrypt the block

$$c_i = E_K(m_i) \oplus c_{i-1}$$



Byte Swapping

- **We add a step where we move bytes around within a block**
- **This removes the positional dependency of each byte**
  - You cannot identify the correspondence of a byte of plaintext with a block of ciphertext
- **Get 16 bytes of key from the keystream generator**
  - Each byte of the keystream will identify two bytes that will be swapped in the block

```
for (i=0; i < blocksize; i=i+1)
    first = key[i] & 0xf   lower 4 bits of the keystream
    second = (key[i] >> 4) & 0xf   top 4 bits of the keystream
    swap(block[first], block[second])      exchange the bytes
```

How the program works

- **Create an initialization vector (IV)**
  - 16 bytes – obtained by reading 16 bytes of data from the keystream generator
- **For each 16-byte block of plaintext**
  1. If it's the last block, add padding
  2. XOR the data with the previous 16 byte-block of ciphertext (the first time, XOR with the IV)
  3. Read 16 bytes of keystream data
  4. Swap 16 pairs of bytes based on the keystream data
  5. Ciphertext_block = result $\oplus$ keystream data (from step 2)
  6. Write the ciphertext

Your programs

- **Two programs – one to encrypt & one to decrypt**

```
sbencrypt password plaintextfile ciphertextfile
sbdecrypt password ciphertextfile plaintextfile
```

- **You will be provided with:**
  - Reference versions of the program: *sbencrypt, sbdecrypt*
  - Small sample content
- **Your program should produce identical output**

Test & Submission

You don't need anything to get started beyond the instructions

Download `a-11.zip` (see assignment) and unzip it

This will provide reference programs and keys

You should test your programs with your own data too!

Submission

- Create a Makefile to create the executables:
  - vencrypt, vdecrypt, scrypt, sbencrypt, sbdecrypt
  - We will not try to figure out how to run your program
- Create a zip file containing the source code & Makefile
  - No executables, no libraries, no test data!

The End