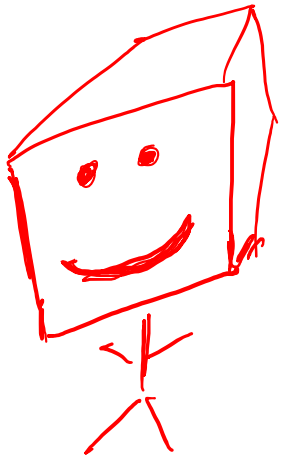
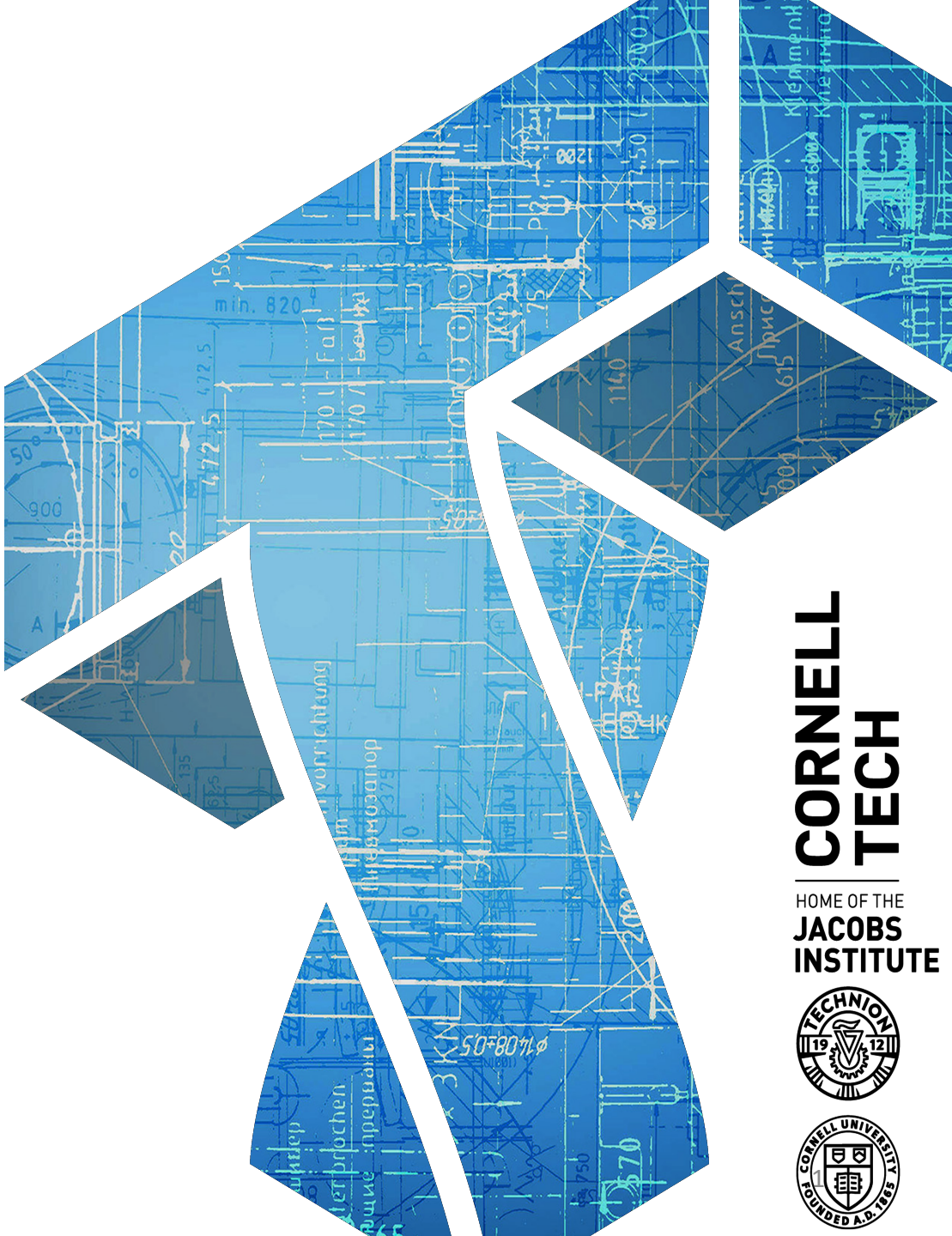
CS 5830
Cryptography



Instructor: Tom Ristenpart
TAs: Yan Ji, Sanketh Menda



CORNELL TECH
HOME OF THE JACOBS INSTITUTE

Administrivia

- Homework 1 will be released today
  - Plaintext recovery attacks against biased stream cipher, ECB mode encryption, incorrectly implemented CTR mode
  - Python coding, not a lot of lines of code
  - Short answers explaining what you did
- I did it over the weekend, lots of fun!

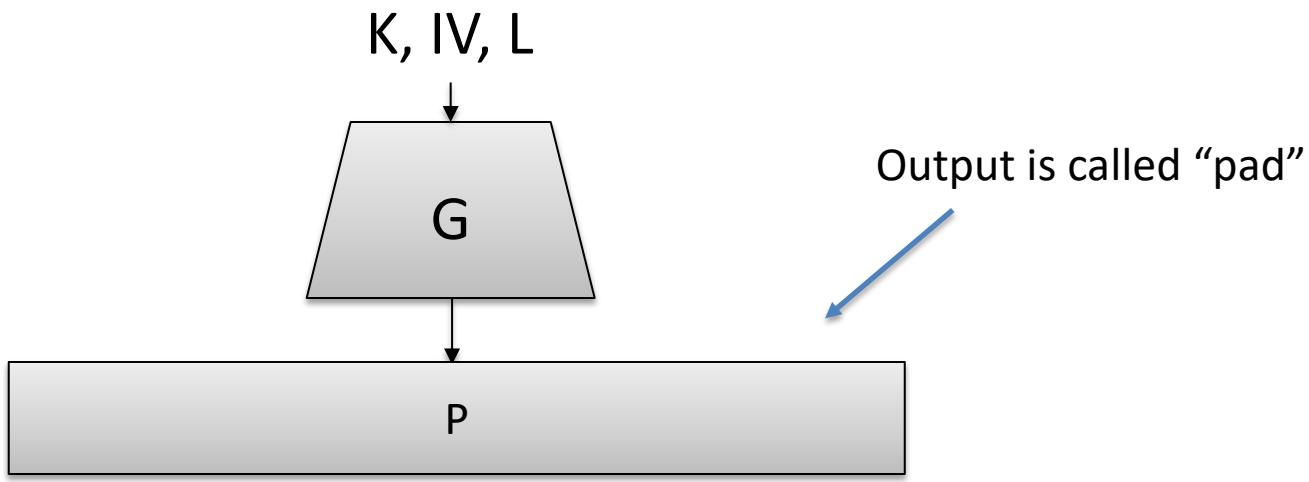
Recap and where we're at

- Shannon and perfect secrecy
  - Good confidentiality against passive adversarys, but impractical
- Computational security
  - Relax to allow tiny success probability or limited adversarial resources
- Stream ciphers as computationally secure one-time pads

Stream ciphers

Stream cipher (aka pseudorandom generator) is pair of algorithms:

- Kg outputs a random key K
- G(K,IV,L) takes K, optionally an additional random value IV (initialization vector, also called nonce), desired length L
  - Outputs bit string P with |P| = L



You'll see PRGs used in different ways

- No nonce / initialization vector
  - PRG(K)
  - Can only ever use K to encrypt a single message
  - Otherwise two-time pad
- With nonce / initialization vector
  - PRG(K,IV)
  - Can use K many times, with distinct IV (usually randomly chosen, sometimes counters)
  - Repeat K, IV = two-time pad
- Length L is sometimes implicit and meant to be clear from context
  - PRG(K) $\oplus$ M means use PRG to generate |M| bits

RC4 stream cipher

- Rivest Cipher 4 (1987)
  - No IV: can't reuse K
  - L is implicit: call generate L times
  - All addition mod 256
- Originally proprietary secret
- Leaked to CypherPunks mail list (1994)
  - "Alleged RC4" or ARCFOUR
  - Rivest confirmed history later
- Very simple, fast to implement in software
- Used widely, only recently deprecated

Algorithm 1: RC4 key scheduling (KSA)

input : key K of l bytes
output : initial internal state st_0
begin
  for $i = 0$ **to** 255 **do**
    $\mathcal{S}[i] \leftarrow i$
  $j \leftarrow 0$
  for $i = 0$ **to** 255 **do**
    $j \leftarrow j + \mathcal{S}[i] + K[i \bmod l]$
    $\text{swap}(\mathcal{S}[i], \mathcal{S}[j])$
  $i, j \leftarrow 0$
  $st_0 \leftarrow (i, j, \mathcal{S})$
  return st_0

Algorithm 2: RC4 keystream generator (PRGA)

input : internal state st_r
output : keystream byte Z_{r+1} updated internal state st_{r+1}
begin
  parse $(i, j, \mathcal{S}) \leftarrow st_r$
  $i \leftarrow i + 1$
  $j \leftarrow j + \mathcal{S}[i]$
  $\text{swap}(\mathcal{S}[i], \mathcal{S}[j])$
  $Z_{r+1} \leftarrow \mathcal{S}[\mathcal{S}[i] + \mathcal{S}[j]]$
  $st_{r+1} \leftarrow (i, j, \mathcal{S})$
  return (Z_{r+1}, st_{r+1})

From:
http://www.isg.rhul.ac.uk/tls/RC4passwords.pdf

RC4 stream cipher

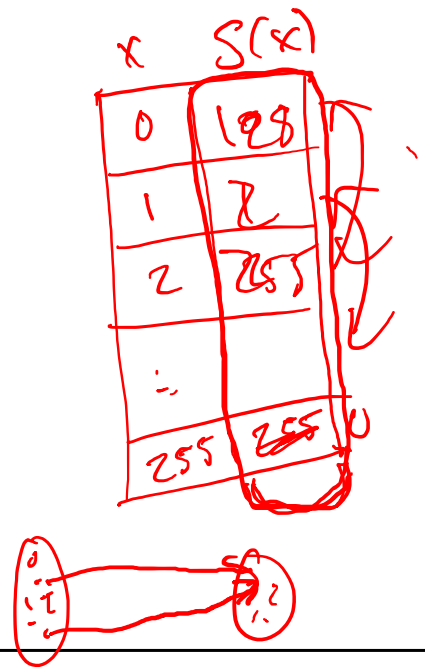
- Key scheduling algorithm uses K to build permutation *S* on {0,…,255} and initialize internal state (i,j,S)

  Interesting discussion of permutation generation:
  https://blog.codinghorror.com/the-danger-of-naivete/

- Keystream generator outputs byte Z_{r+1} and updated internal state
  - *S* remains permutation throughout
  - i is a counter
  - j varies as function of i, *S*, previous j

Algorithm 1: RC4 key scheduling (KSA)

input : key K of l bytes
output : initial internal state st_0
begin
  for $i = 0$ **to** 255 **do**
    $\mathcal{S}[i] \leftarrow i$
  $j \leftarrow 0$
  for $i = 0$ **to** 255 **do**
    $j \leftarrow j + \mathcal{S}[i] + K[i \bmod l]$
    $\text{swap}(\mathcal{S}[i], \mathcal{S}[j])$
  $i, j \leftarrow 0$
  $st_0 \leftarrow (i, j, \mathcal{S})$
  return st_0



Algorithm 2: RC4 keystream generator (PRGA)

input : internal state st_r
output : keystream byte Z_{r+1} updated internal state st_{r+1}
begin
  parse $(i, j, \mathcal{S}) \leftarrow st_r$
  $i \leftarrow i + 1$
  $j \leftarrow j + \mathcal{S}[i]$
  $\text{swap}(\mathcal{S}[i], \mathcal{S}[j])$
  $Z_{r+1} \leftarrow \mathcal{S}[\mathcal{S}[i] + \mathcal{S}[j]]$
  $st_{r+1} \leftarrow (i, j, \mathcal{S})$
  return (Z_{r+1}, st_{r+1})

From:
http://www.isg.rhul.ac.uk/tls/RC4passwords.pdf

RC4 stream cipher

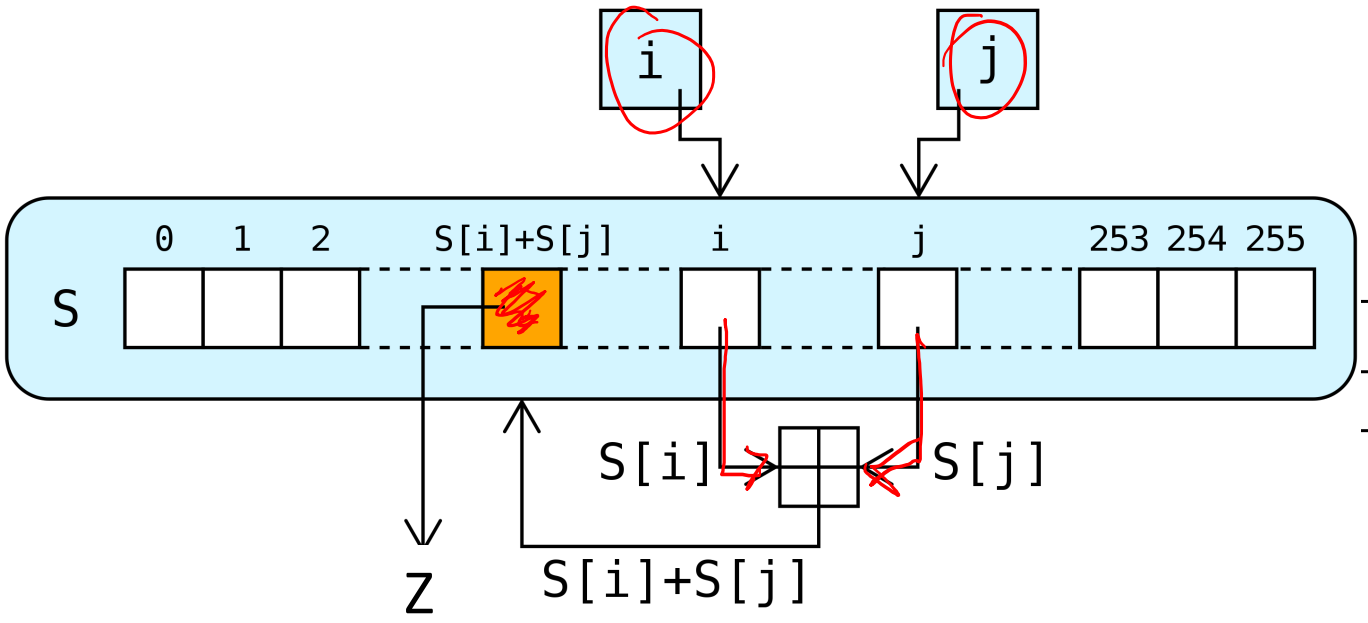


Diagram courtesy of https://en.wikipedia.org/wiki/RC4

Algorithm 1: RC4 key scheduling (KSA)

input : key K of l bytes
output : initial internal state st_0
begin
  for $i = 0$ **to** 255 **do**
    $\mathcal{S}[i] \leftarrow i$
  $j \leftarrow 0$
  for $i = 0$ **to** 255 **do**
    $j \leftarrow j + \mathcal{S}[i] + K[i \bmod l]$
    $\text{swap}(\mathcal{S}[i], \mathcal{S}[j])$
  $i, j \leftarrow 0$
  $st_0 \leftarrow (i, j, \mathcal{S})$
  return st_0

Algorithm 2: RC4 keystream generator (PRGA)

input : internal state st_r
output : keystream byte Z_{r+1} updated internal state st_{r+1}
begin
  parse $(i, j, \mathcal{S}) \leftarrow st_r$
  $i \leftarrow i + 1$
  $j \leftarrow j + \mathcal{S}[i]$
  $\text{swap}(\mathcal{S}[i], \mathcal{S}[j])$
  $Z_{r+1} \leftarrow \mathcal{S}[\mathcal{S}[i] + \mathcal{S}[j]]$
  $st_{r+1} \leftarrow (i, j, \mathcal{S})$
  return (Z_{r+1}, st_{r+1})

From:
http://www.isg.rhul.ac.uk/tls/RC4passwords.pdf

RC4 stream cipher

- How do evaluate RC4 security?
  - Manual analysis:

    walk through code and analyze probabilities

[Mantin-Shamir 2001]: $\Pr[Z_2 = 0x00] \approx 1/128$

  - Statistical tests:

    for a random key, do you get expected distribution of output bytes?

    for many random keys, do you get expected distribution for each byte?

[AlFardan et al. 2013] calculated outputs for 2^{44} keys and reported on empirical distributions

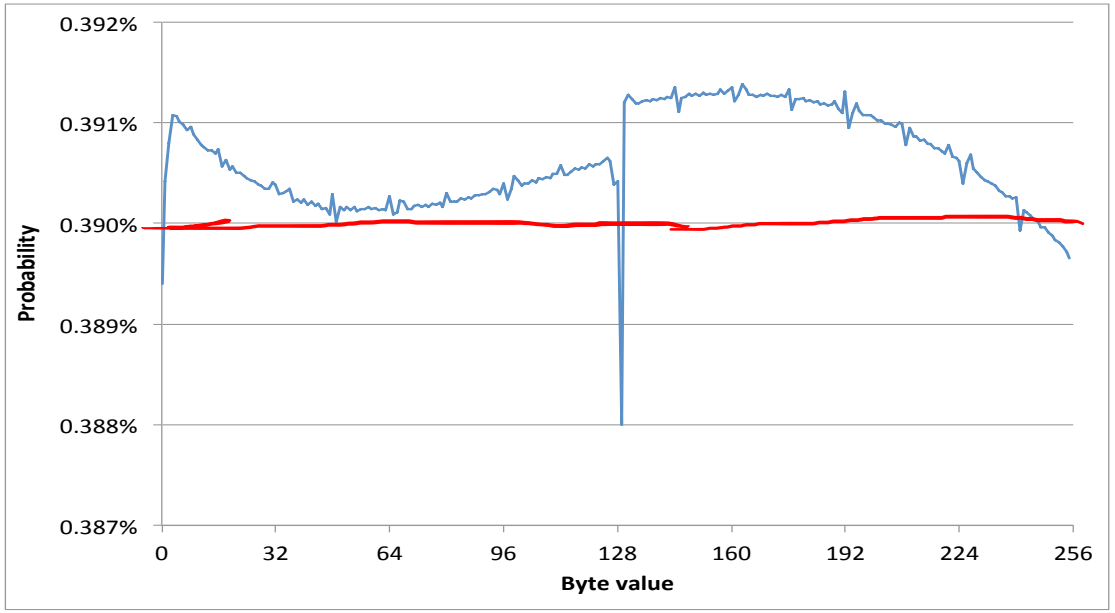
Algorithm 1: RC4 key scheduling (KSA)

input : key K of l bytes
output : initial internal state st_0
begin
  for $i = 0$ **to** 255 **do**
    $\mathcal{S}[i] \leftarrow i$
  $j \leftarrow 0$
  for $i = 0$ **to** 255 **do**
    $j \leftarrow j + \mathcal{S}[i] + K[i \bmod l]$
    $\text{swap}(\mathcal{S}[i], \mathcal{S}[j])$
  $i, j \leftarrow 0$
  $st_0 \leftarrow (i, j, \mathcal{S})$
  return st_0

Algorithm 2: RC4 keystream generator (PRGA)

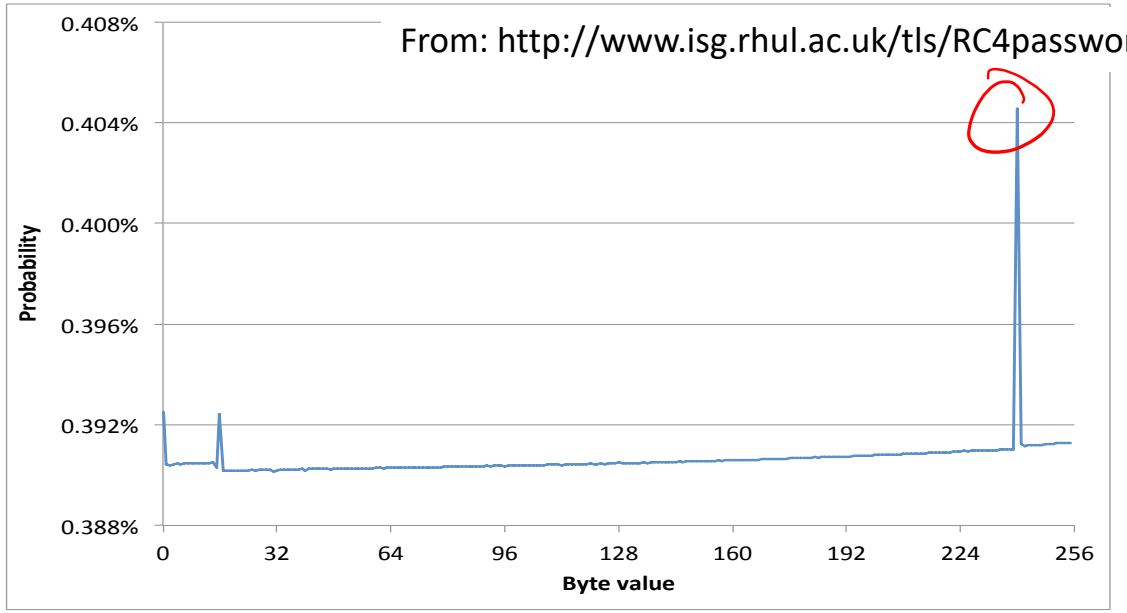
input : internal state st_r
output : keystream byte Z_{r+1} updated internal state st_{r+1}
begin
  parse $(i, j, \mathcal{S}) \leftarrow st_r$
  $i \leftarrow i + 1$
  $j \leftarrow j + \mathcal{S}[i]$
  $\text{swap}(\mathcal{S}[i], \mathcal{S}[j])$
  $Z_{r+1} \leftarrow \mathcal{S}[\mathcal{S}[i] + \mathcal{S}[j]]$
  $st_{r+1} \leftarrow (i, j, \mathcal{S})$
  return (Z_{r+1}, st_{r+1})

From:
http://www.isg.rhul.ac.uk/tls/RC4passwords.pdf

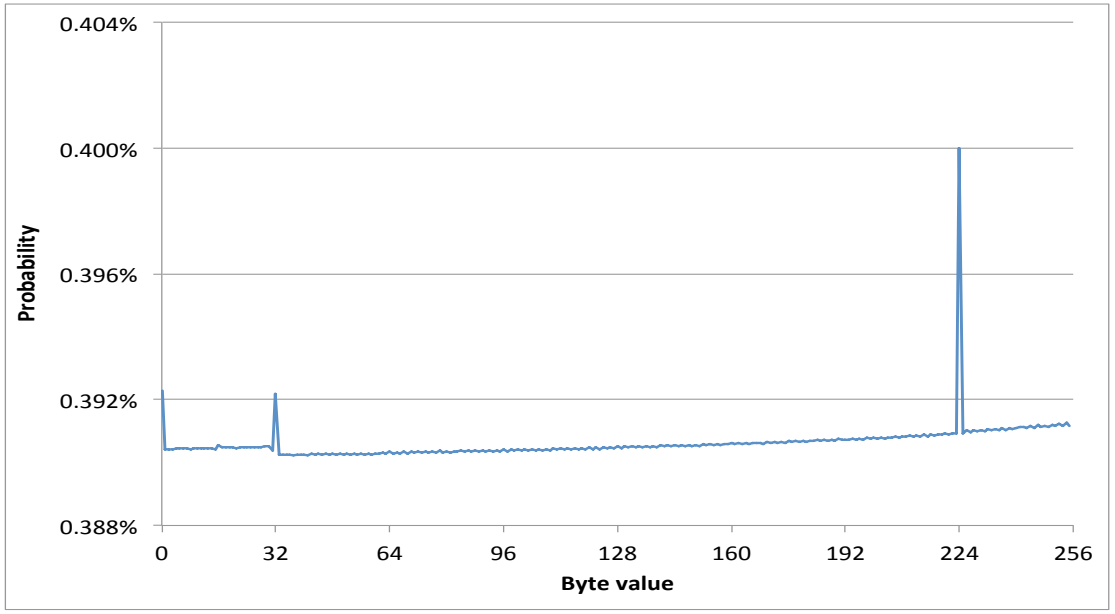


(a) Byte Z_1

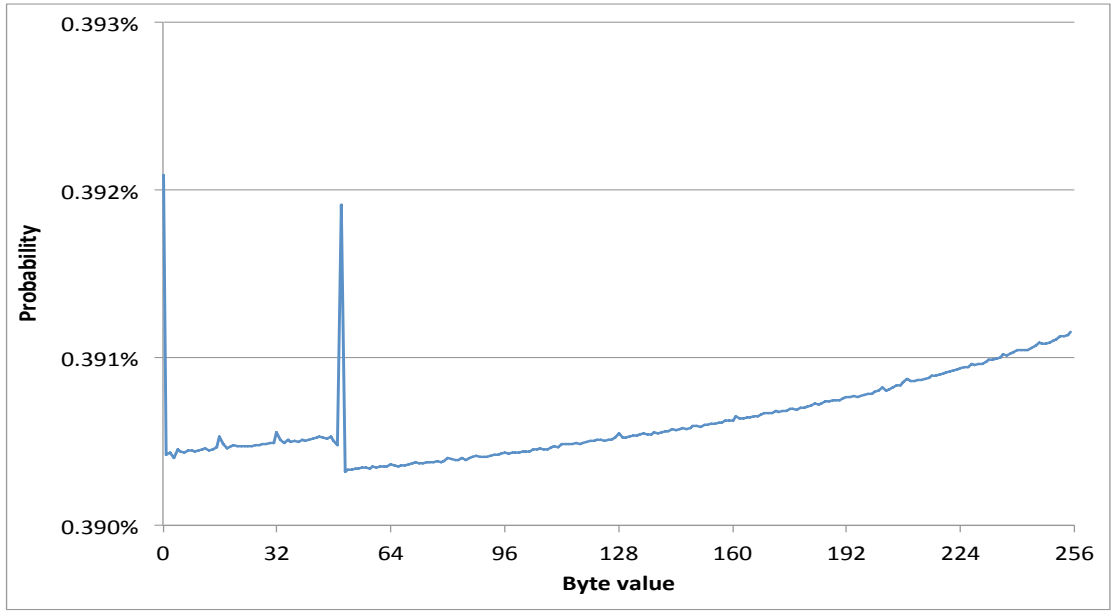
From: http://www.isg.rhul.ac.uk/tls/RC4passwords.pdf



(b) Byte Z_{16}



(c) Byte Z_{32}



(d) Byte Z_{50}

Bias-abusing attacks

$C_1 = \mathrm{PRG}(K_1) \oplus M$

$C_2 = \mathrm{PRG}(K_2) \oplus M$

$\vdots$

$C_q = \mathrm{PRG}(K_q) \oplus M$

Broadcast attack:

Unknown message M

Encrypted q times under q different K (or different K,N pairs)

Use ciphertexts and biases to recover (portions of) M

E.g., M is password
and this is login-over-TLS
Each encryption is fresh session

Say we know bias, e.g. $\Pr[\ \mathrm{PRG}(K)[0] = 0\cancel{x00}\ ] > \delta$

- What settings would give rise to broadcasts?
- How could you estimate δ ?
- How do you exploit?

Bias-abusing with shifting

$$\text{PRG}(K_1) \oplus (M_1 \,||\, M)$$

$$\text{PRG}(K_2) \oplus (M_2 \,||\, M)$$

$$\vdots$$

$$\text{PRG}(K_q) \oplus (M_q \,||\, M)$$

Broadcast attack with plaintext shifting:

Unknown message M

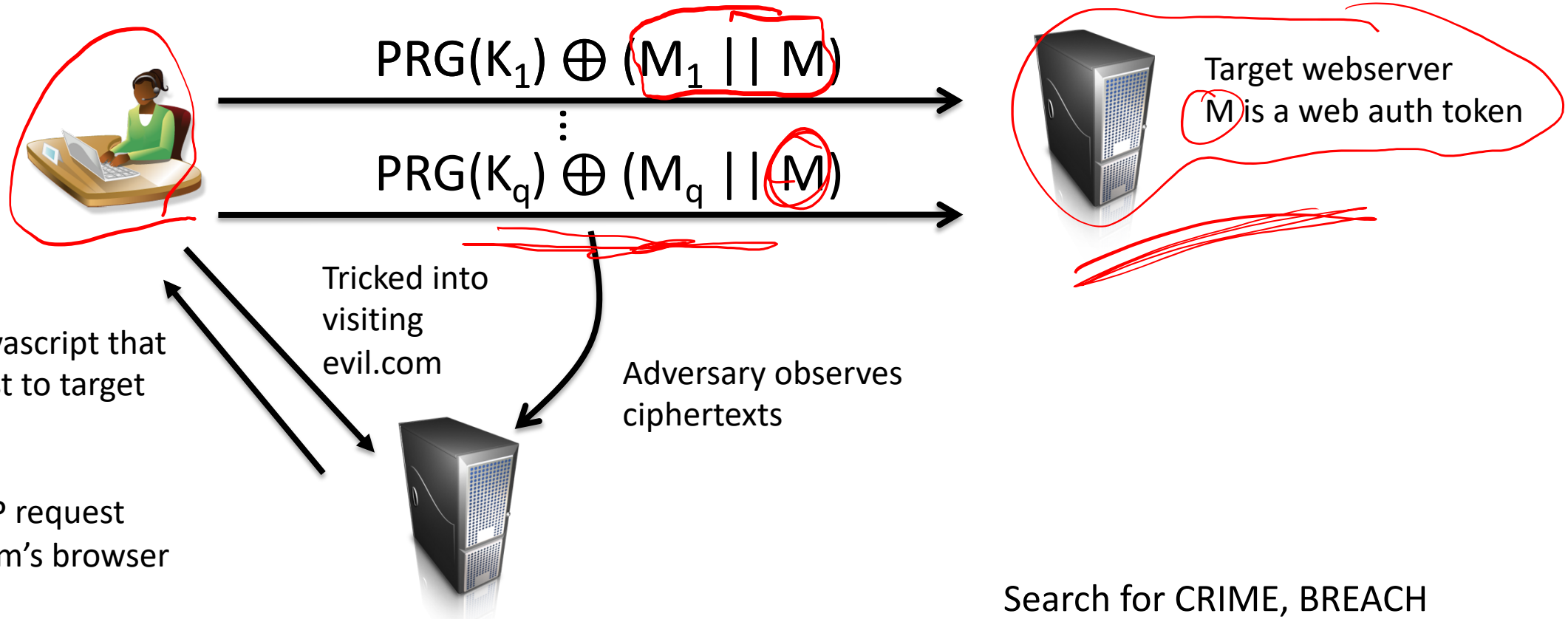
Attacker controls message prefix M_i

Encrypted q times under q different K (or different K,N pairs)

Use ciphertexts and biases to recover all of M

In what kind of situations could this arise?

Bias-abusing with shifting



Controls HTTP request headers, victim's browser inserts secret authentication cookie M

M is enough to authenticate as user to target webserver

Search for CRIME, BREACH attacks against TLS for more discussion of setting & demonstrations

Fallout from 2013 RC4 attacks

- RC4 was the most widely used encryption method for TLS at the time (circa 2013)
- Attack required about q = 2^{26} (67 million) to start recovering plaintexts
  - Not quite practical, but within realm of feasibility
  - Enough to be considered ***significant problem***, and potentially within reach of intelligence agencies
- Needed to move on from RC4, but all other TLS encryption methods had even more severe security problems (stay tuned)
- Have replaced now with encryption based on ***block ciphers***

Block ciphers

Family of permutations, one permutation for each key

$$E : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$$

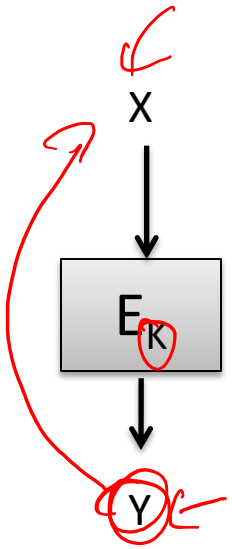
Use notation $E(K,X) = E_K(X) = Y$

Define inverse $D(K,Y) = D_K(Y) = X$

$$\text{s.t. } D(K,E(K,X)) = X$$

E,D must be efficiently computable

Pick K uniformly at random from $\{0,1\}^k$



X	Y
00	10
01	11
10	01
11	00

X	Y
00	11
01	10
10	00
11	01

X	Y
00	11
01	10
10	00
11	01

Block cipher design

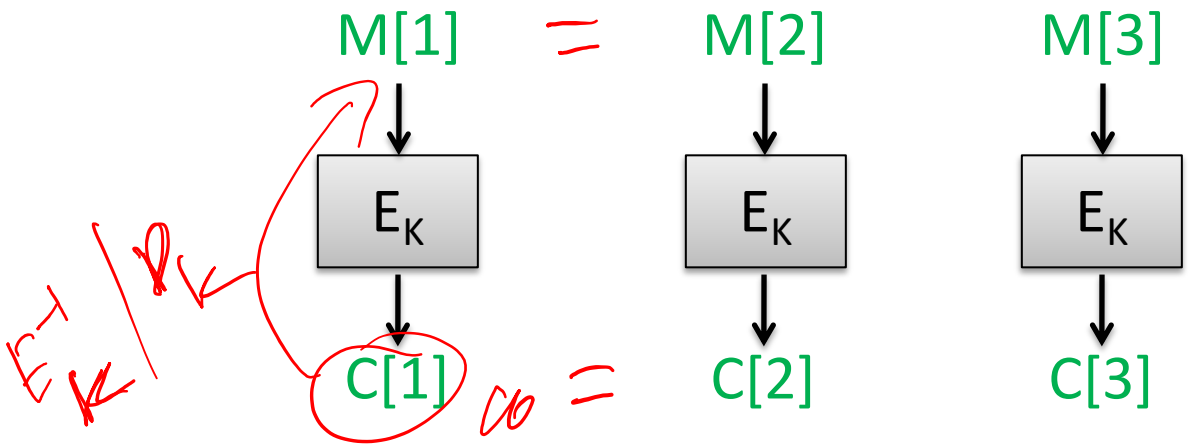
- A big topic with ~50 year history
- Data Encryption Standard (DES) designed in 1970s
  - Uses Feistel network structure
- Advanced Encryption Standard (AES) designed in 1990s
  - Uses substitution-permutation network structure
- We will dig into this more next lecture
- First: How to encrypt messages with block ciphers?

Block cipher modes of operation

Electronic codebook (ECB) mode

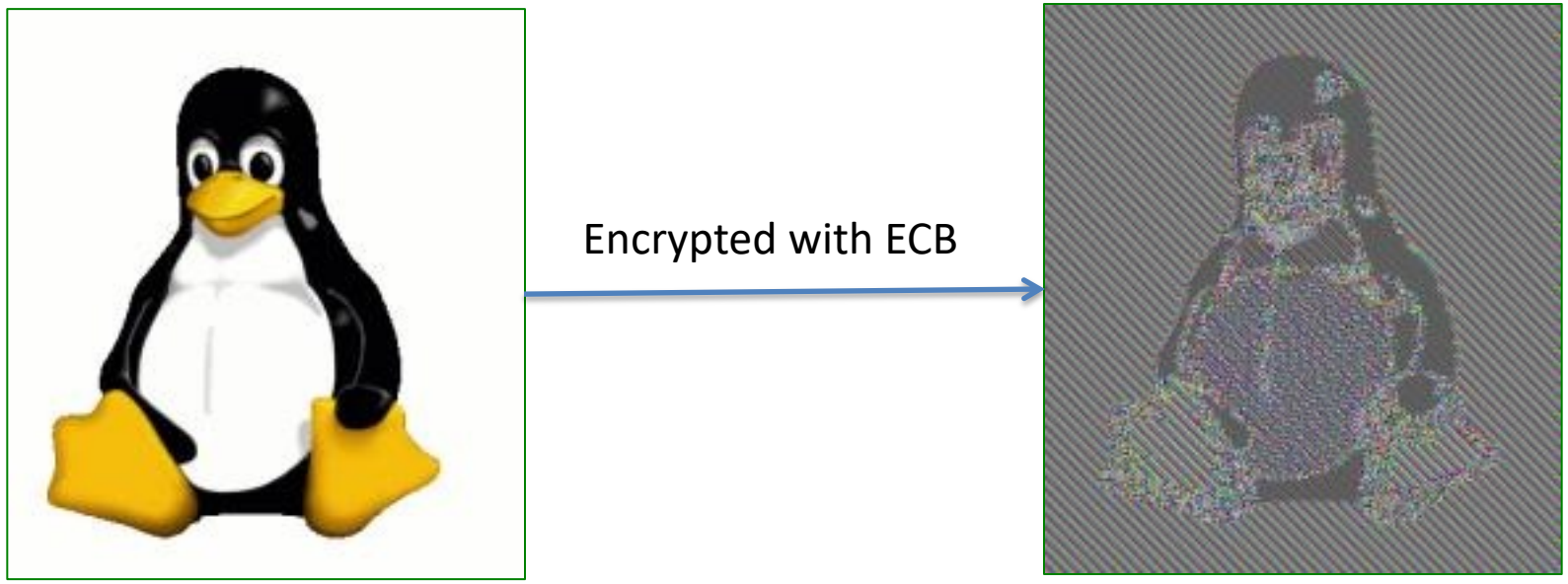
Pad message M to M[1],M[2],M[3],... where each block M[i] is n bits

Then:



How can we decrypt?

ECB mode is a more complicated looking substitution cipher

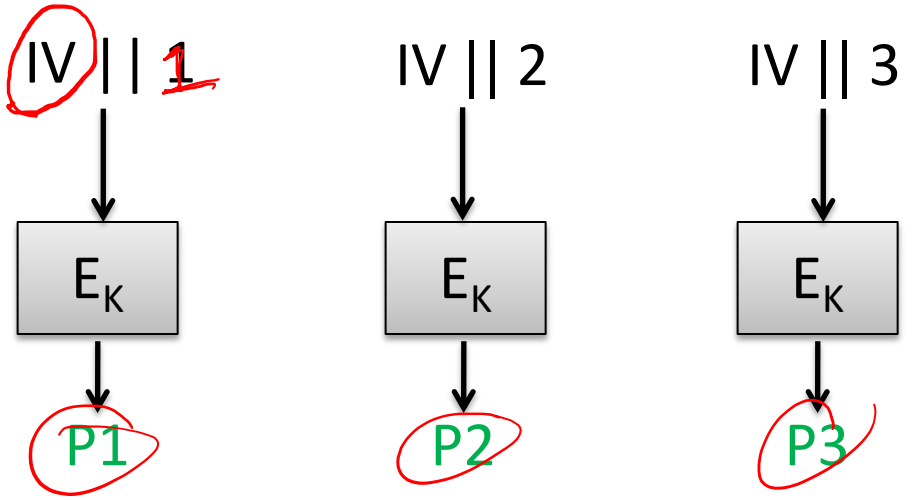


Images courtesy of
http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

CTR mode stream cipher

Counter mode stream cipher:

- Kg outputs random k-bit key for block cipher
- $G(K,IV,L) = E_K(IV \,||\, 1) \,\|\, E_K(IV \,||\, 2) \,\|\, \ldots \,\|\, \text{trunc}(E_K(IV \,||\, m))$
  where $m = \text{ceil}(L / n)$
- Here |IV| smaller than n bits, say n/2 bits



Truncate P3
to get L total bits

CTR-mode SE scheme

Kg():
K <-$ $\{0,1\}^k$

Enc(K,M):
L <- |M| ; m <- ceil(L/n)
IV <-$ $\{0,1\}^{n/2}$
P <- $E_K(IV \,||\, 1) \,\|\, \cdots \,\|\, \text{trunc}(E_K(IV \,||\, m))$
Return (IV, $P \oplus M$)

Dec(K,(IV,C)):
L <- |C| ; m <- ceil(L/n)
P <- $E_K(IV \,||\, 1) \,\|\, \cdots \,\|\, \text{trunc}(E_K(IV \,||\, m))$
Return (IV, $P \oplus C$)

Pick a random key

Should be able to call Enc with same K for many messages. What could go wrong?

What security properties do we need from the block cipher?

Assume ciphertext can be parsed into IV and remaining ciphertext bits

Summary

- Target security against computational attackers
- Stream ciphers are computational equivalent of OTP
  - Keys must be large enough to avoid brute force
  - Even moderate biases in output stream must be avoided
- Stream ciphers give efficient symmetric encryption secure against passive adversaries
- None achieve security against active adversaries (stay tuned)

Next up

- Block cipher security goals
  - Pseudorandom functions and permutations
- Building & analyzing block ciphers
  - Feistel networks and DES
  - AES cipher