INTRODUCTION TO CRYPTOGRAPHIC ATTACKS

EXERCISE LOGISTICS

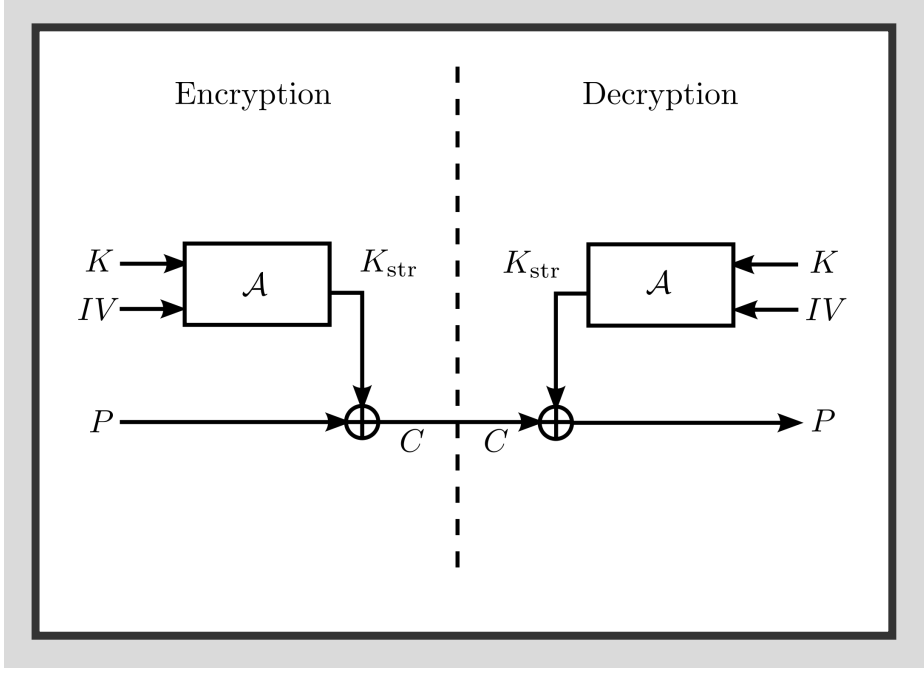
- Workshop Challenges
- Username: workshop
- Password: workshop
- Startup code in PyCharm Projects

STREAM CIPHERS

ONE TIME PAD

Plaintext:	A	T	T	A	C	K	A	T	D	A	W
Numeric Plaintext:	0	19	19	0	2	10	0	19	3	0	22
OTP:	M	K	Q	I	V	S	O	F	G	S	J
Numeric OTP:	12	10	16	8	21	18	14	5	6	18	9
Numeric Ciphertext:	12	3	9	8	23	2	14	24	9	18	5
Ciphertext:	M	D	J	I	X	C	O	Y	J	S	F

STREAM CIPHER OPERATION



XOR

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

KEYSTREAM REUSE

$$Ctext1 = Ptext1 \oplus KeyStream$$

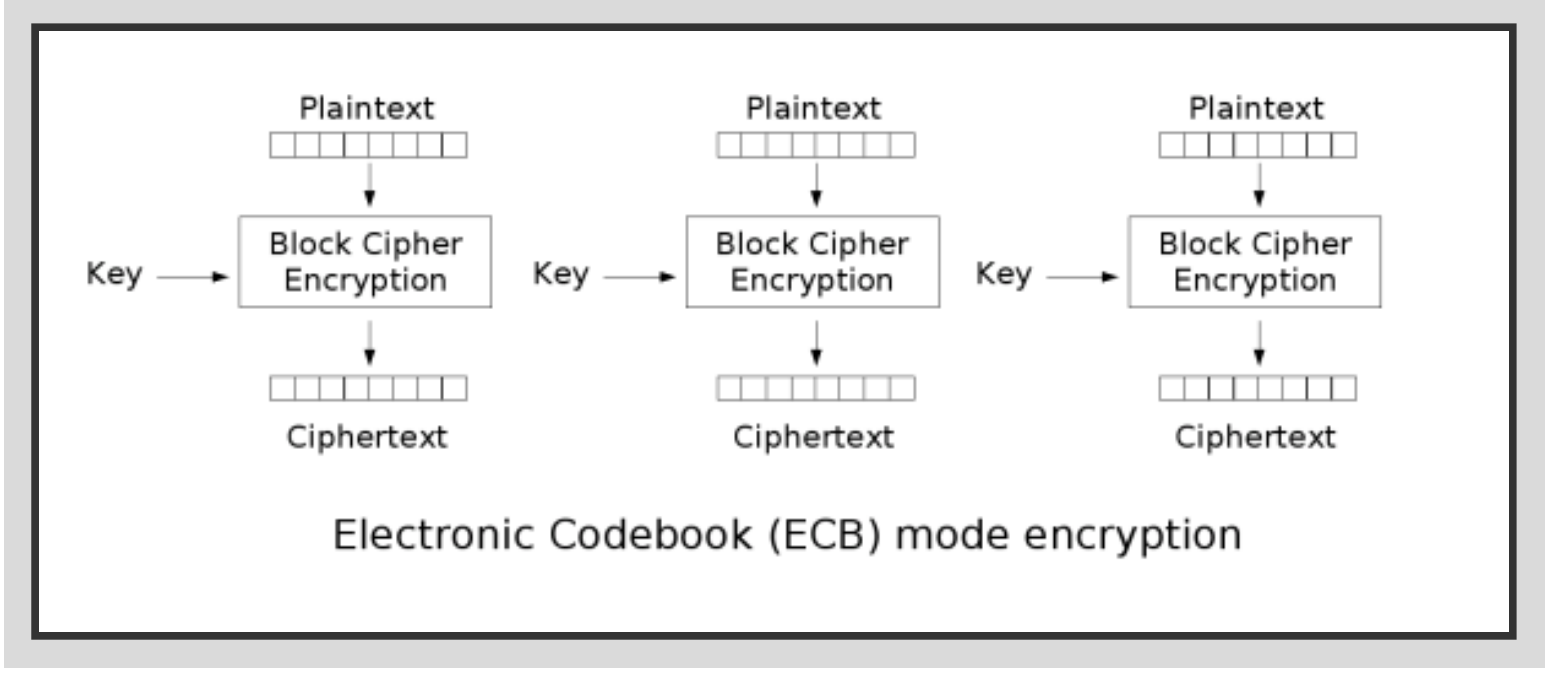
$$Ctext2 = Ptext2 \oplus KeyStream$$

$$Ctext1 \oplus Ctext2 = Ptext1 \oplus Ptext2$$

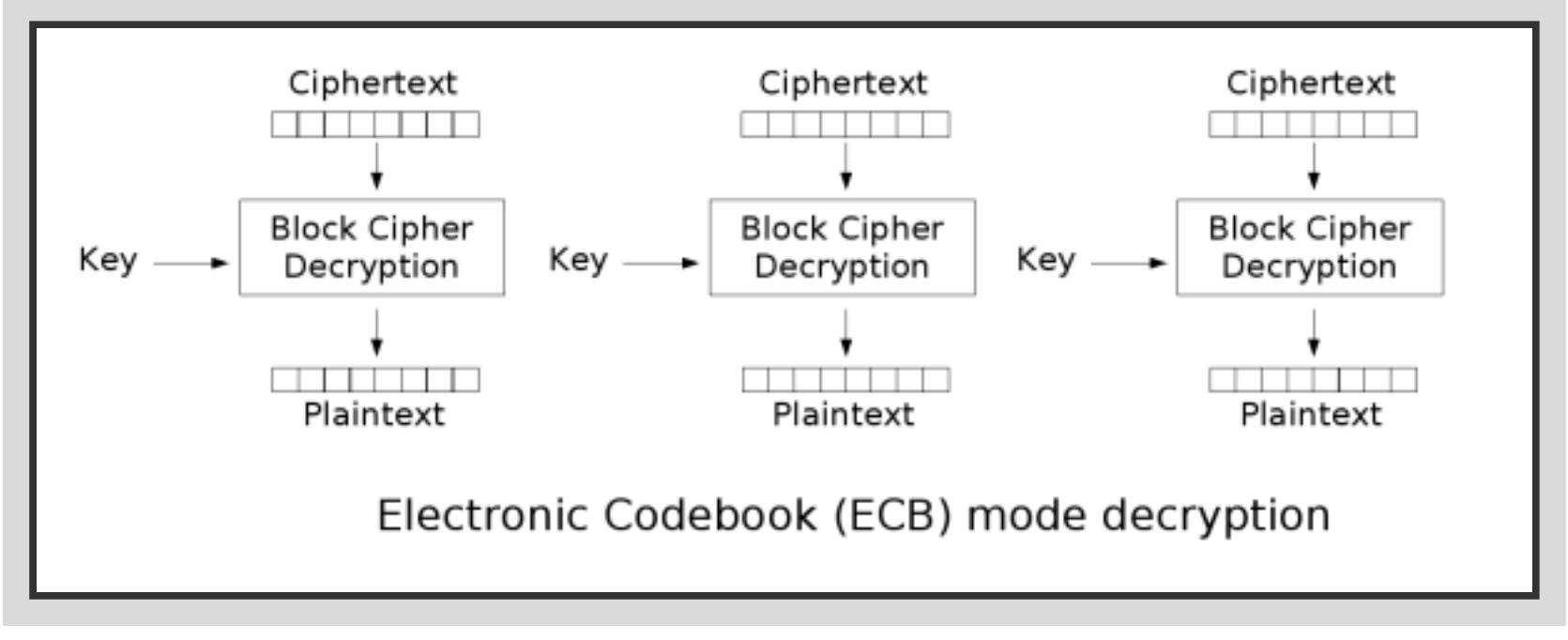
BLOCK CIPHERS

- Fixed sized input
- Random looking output for each message and key
- Block Cipher Modes

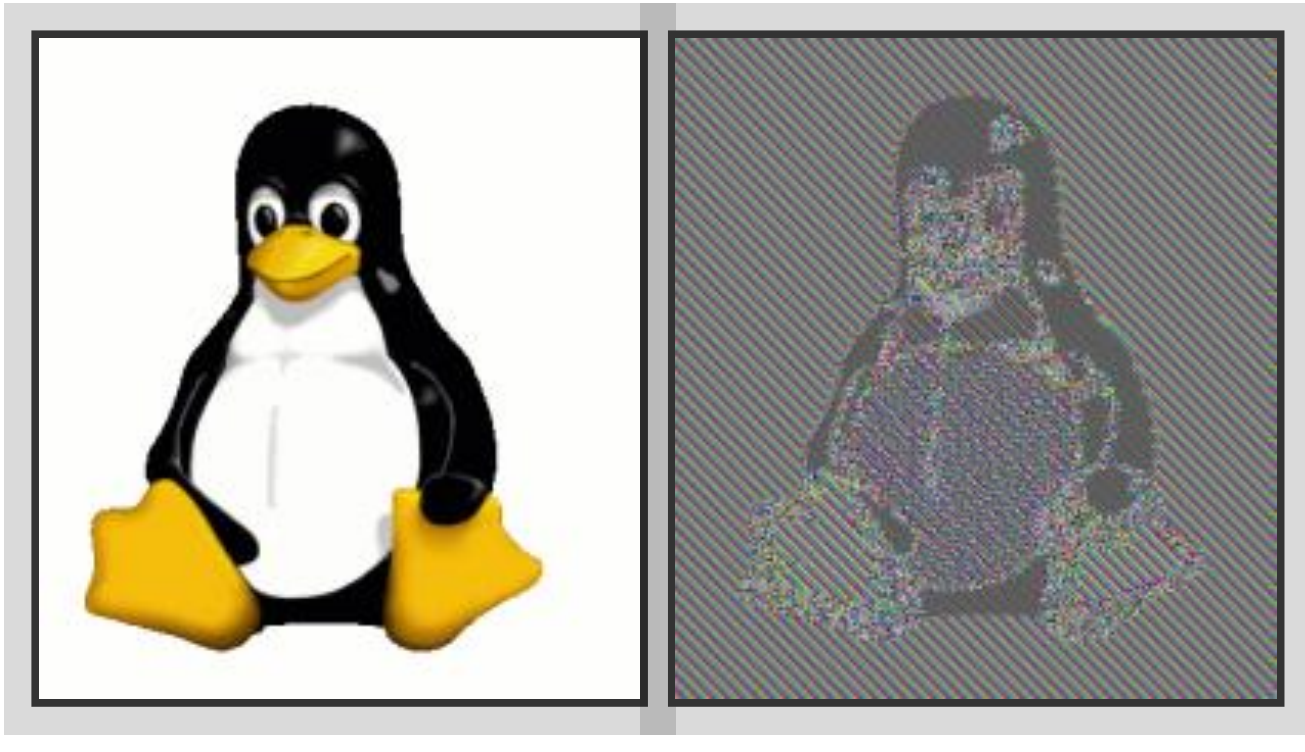
ECB MODE ENCRYPTION



ECB MODE DECRYPTION

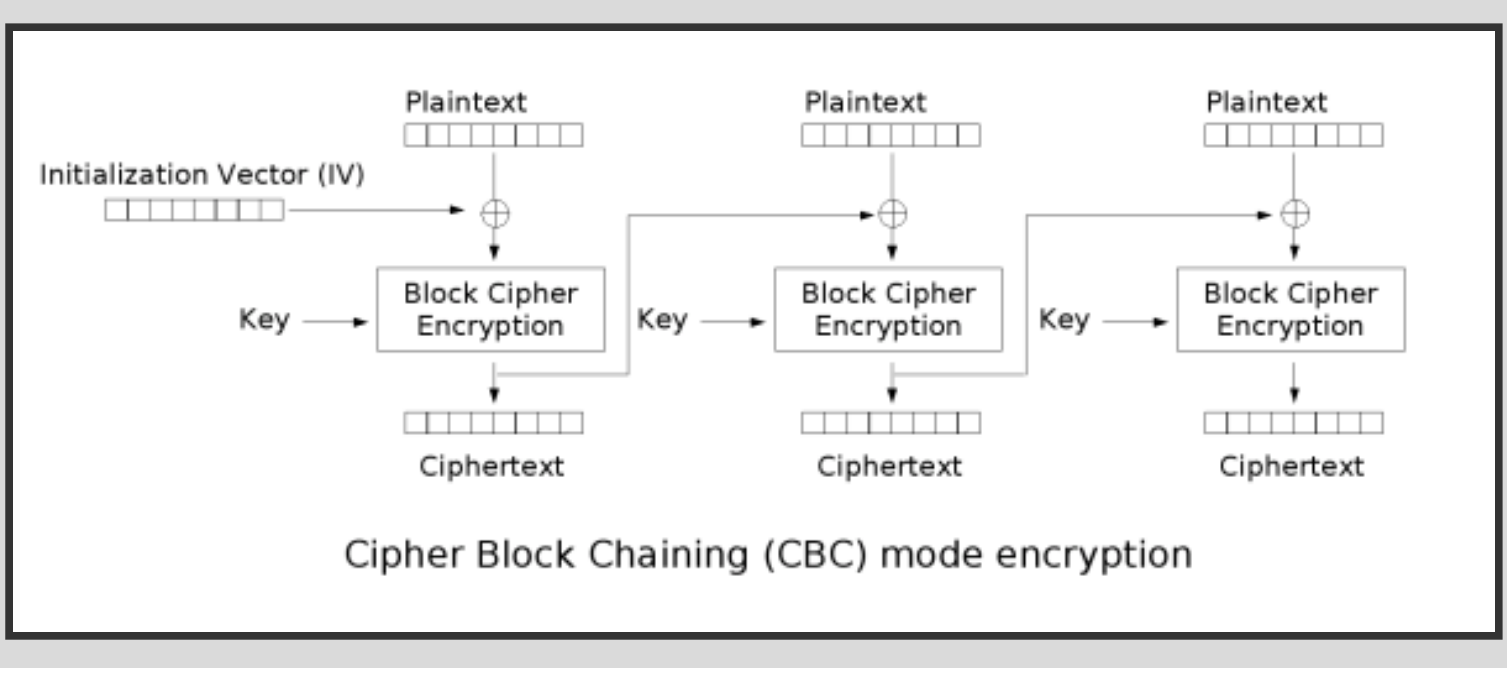


PENGUINS AND ELECTRONIC CODE BOOK

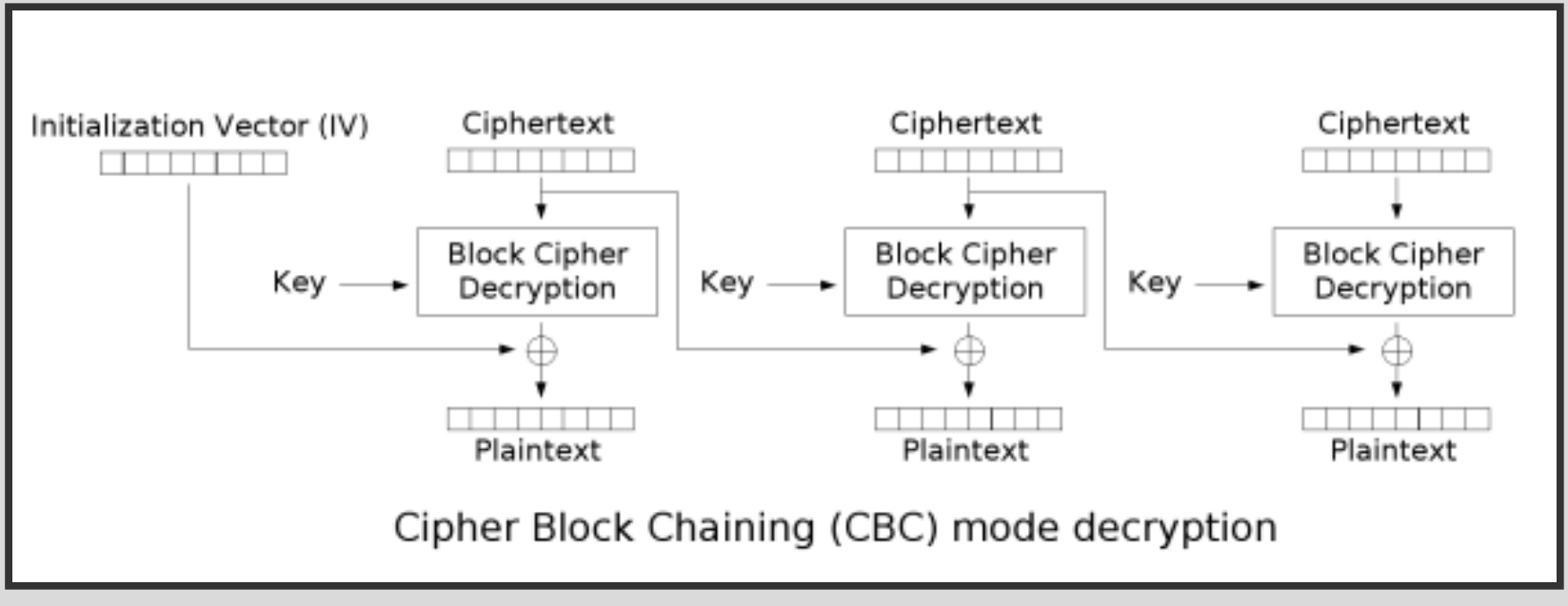


CIPHER BLOCK CHAINING

CBC MODE ENCRYPTION



CBC MODE DECRYPTION



NOTES ON BLOCK CIPHER CHALLENGES

- Using AES which has 16 byte block size
- In hex that equals 32 characters

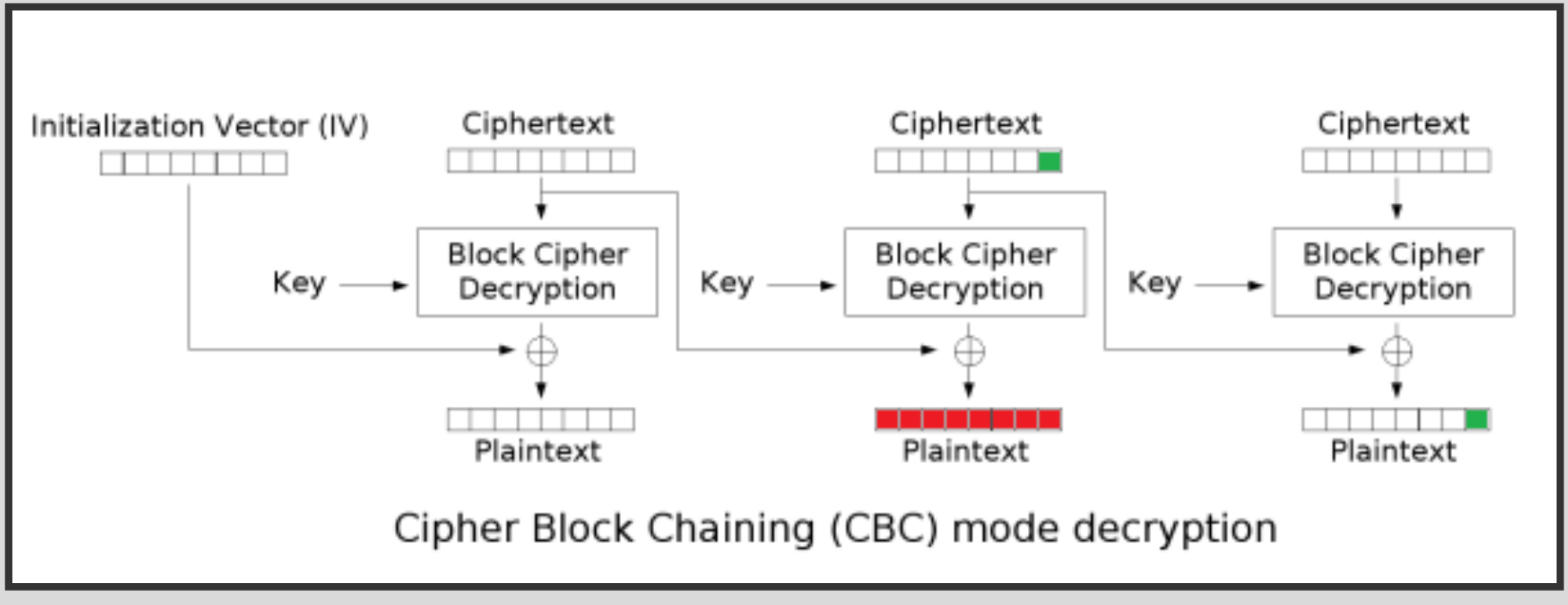
PADDING ORACLE ATTACK

PKCS#7 PADDING

- Count how many bytes needed to finish block
- Append that many bytes to the last block
- Set the value of each of those bytes to the number added

```
|dd dd dd dd dd dd dd dd dd dd dd dd dd dd dd dd|
|dd dd dd dd 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c|
```

AFFECTING DECRYPTION



DESCRIPTION OF ATTACK

- Changing the second to last block produces predictable change to pad
- Guess byte at a time starting with the last byte
- If you guess is correct you have a valid pad

EXAMPLE GUESS FOR THE LAST BYTE

```
guess_and_pad = xor_string(b'\x00'*15+guess,b'\x00'*15+b'\x01')
modified_ciphertext = xor_string(ciphertext, guess_and_pad)
```

EXAMPLE

- Take the same plaintext and key
- IV:

```
c0103a601101dec2acfb3d869bb96985
```

- ciphertext:

```
ef7b84991fa8fda6b8d429ffdf1091b9
22E2B24B804911958545E79E75B6966B
```

- New pad:

```
00000000000000000000000000000001
```

- Guess:

```
00000000000000000000000000000004
```

MODIFICATION OF PENULTIMATE BLOCK

$$
\begin{array}{rl}
 & ef7b84991fa8fda6b8d429ffdf1091b9 \\
 & 00000000000000000000000000000001 \\
\oplus & 00000000000000000000000000000004 \\
\hline
 & ef7b84991fa8fda6b8d429ffdf1091bc
\end{array}
$$

DECRYPTION ORACLE

- Decryption:

```
787C41849A2158240F6C1C4D96F3023C
206D65737361676508080808080808D
```

- Return bad pad

CORRECT GUESS

$$
\begin{array}{rl}
 & ef7b84991fa8fda6b8d429ffdf1091b9 \\
 & 00000000000000000000000000000001 \\
\oplus & 00000000000000000000000000000008 \\
\hline
 & ef7b84991fa8fda6b8d429ffdf1091b0
\end{array}
$$

DECRYPTION ORACLE

- Decryption:

```
B7F19E2E7035EE6E2BBB0CDDEDCBE971
206D6573736167650808080808080801
```

- Return good pad

CORRECT GUESS

$$
\begin{array}{rl}
 & ef7b84991fa8fda6b8d429ffdf1091b9 \\
 & 00000000000000000000000000000202 \\
\oplus & 00000000000000000000000000000808 \\
\hline
 & ef7b84991fa8fda6b8d429ffdf1091b0
\end{array}
$$

DECRYPTION ORACLE

- Decryption:

```
2EF5F8B2E68E2F48873AD78530553FF3
206D65737361676508080808080202
```

- Return good pad

FINDING PADDING ORACLES

- Find ciphertext that's multiple blocks long
- Change the first byte of the first block and observe response
- Compare to response from changing the last byte of the second to last block
- Possible signs of invalid/valid pads
  - Timing differences
  - Differences in error messages/status codes

REAL WORLD EXAMPLES

- POODLE CVE-2014-3566
- Apache mod_session_crypto CVE-2016-0736

(EC)DSA

HOW DOES IT WORK?

FOR ALL PUBLIC KEY SLIDES

- Numbers will be written in base 16
- Modulus of an equation may not always be explicit

SIGNING

- Generate per message value k where 0<k<q
- Calculate $r = \left(g^k \mod p\right) \mod q$. If $r = 0$ choose new k.
- Calculate $s = k^{-1}(H(m) + xr) \mod q$. If $s = 0$, start over.
- Signature is (r, s)

VERIFICATION

- Reject if $0 < r < q$ or $0 < s < q$ are not satisfied
- Calculate $w = s^{-1} \mod q$
- Calculate $u_1 = H(m) \cdot w \mod q$
- Calculate $u_2 = r \cdot w \mod q$
- Calculate $v = (g^{u_1} \cdot y^{u_2} \mod p) \mod q$
- Signature is invalid unless $v = r$

KNOWN NONCE ATTACK

$$x = r^{-1}(sk - H(m)) \mod q$$

EXAMPLE KEY PARAMETERS

- $p = \mathrm{dabf4205f2e14301f33b037e5374800a8c5$
- $q = \mathrm{2000cd8327}$
- $g = \mathrm{bb10df09c69f69efe9e0c692068252c1fd9c$

SIGNATURE

- $y = \text{a848d9e77be54ce255ac07f73f7820b1190}$
  $h = \text{sha256('This is a test message')} \quad \text{mod}$
- $= 15\text{ed}55355\text{e} \quad \text{mod}$
- $k = 15\text{ed}55355\text{e}$
- $r = 1175\text{d}72735$
- $s = 55\text{ca}12\text{fb}$

COMPUTING PRIVATE KEY

- $r^{-1} = \mathrm{d06b6e353}$
- $sk = \mathrm{17b8055023}$
- $sk - h = \mathrm{1cab01ac5}$
- $r^{-1}(sk - h) = \mathrm{114e3b383f} = x$
- $g^x = y$

REPEATED NONCE ATTACK

Suppose (r, s_1) and (r, s_2) with $s_1 \neq s_2$ are valid signatures

$$\begin{aligned} k &= s_1^{-1}(H(m_1) + xr) \mod q \\ &= s_2^{-1}(H(m_2) + xr) \mod q \end{aligned}$$

$$x = (r(s_2 - s_1))^{-1}(s_1 h_2 - s_2 h_1) \mod q$$

EXAMPLE: NEW PUBLIC KEY

- Same p, q, g as before
- $y = 864\text{d}327\text{e}65175220\text{bc}37\text{d}25\text{bf}6\text{b}57\text{d}89\text{f}48$

EXAMPLE: FIRST SIGNATURE

- $h_1 = \text{sha256}(\text{'This is a test message'}) \quad \text{mod}$
  $= 15\text{ed}55355\text{e} \quad \text{mod}$
- $r = \text{c}486\text{e}38\text{ce}$
- $s_1 = 1\text{bbf}1\text{ecfb}9$

EXAMPLE: SECOND SIGNATURE

- $h_2 = \text{sha256}(\text{'This is a new message'})$ mod
  $= \text{6ca2d6ff7}$ mod
- $r = \text{c486e38ce}$
- $s_2 = \text{1cf9e14eb3}$

COMPUTING THE PRIVATE KEY

- $s_2 - s_1 = 13\text{ac}27\text{efa}$
- $r(s_2 - s_1) = 16\text{fe}8911\text{b}1$
- $(r(s_2 - s_1))^{-1} = 126\text{b}09\text{fe}7$
- $s_1 h_2 = 1\text{e}6\text{d}804\text{ea}6$
- $s_2 h_1 = 174\text{d}764767$
- $s_1 h_2 - s_2 h_1 = 7200\text{a}073\text{f}$
- $(r(s_2 - s_1))^{-1}(s_1 h_2 - s_2 H(m_1)) = 1\text{a}25\text{c}0\text{c}8$

EXAMPLES FROM THE REAL WORLD

- Some Bitcoin Implementations
- SSH servers typically on embedded systems
- Sony's Playstation 3 ECDSA implementation



RSA

RSA ENCRYPTION AND DECRYPTION

- To encrypt message 0<m<n, compute
  $c = m^e \mod n$
- To decrypt c, compute $m = c^d \mod n$

RSA SIGNATURES

- To sign message m, $s = m^d \mod n$
- Verify signature by verifying $m == s^e \mod n$

HOMOMORPHIC PROPERTIES OF RSA

- $c_1 = m_1^e \mod n$
- $c_2 = m_2^e \mod n$
- $c_1 c_2 = (m_1 m_2)^e \mod n$
- $(c_1 c_2)^d \mod n = m_1 m_2 \mod n$

ADAPTIVE CHOSEN CIPHERTEXT ATTACK

- Consider an oracle that decrypts ciphertext it hasn't seen before
- Consider an unknown ciphertext c and corresponding plaintext message m
- Generate random $1 < r < n$ and compute $c' = r^e c \mod n$
- Get decryption m'
- $m = r^{-1} m' \mod n$

EXAMPLE RSA KEY

- $n = 4000001320\text{ca}68\text{d}6629986\text{a}0\text{d}8\text{a}86\text{cf}88\text{f}$
- $e = 10001$

EXAMPLE CIPHERTEXT

- $m = 546869732069732061207465737374$

  (ASCII encoding of "This is a test")
- $c = 144e2ad0e54a707c703da9c9e821e25909$

EXAMPLE RANDOM CIPHERTEXT

- $r = 2\text{c}0\text{aca}049446\text{ae}92\text{fdda}3\text{d}0\text{d}35\text{d}848\text{a}11\text{c}$
- $c_r = 47\text{b}4\text{f}96\text{ae}98\text{aa}933\text{c}4\text{e}6\text{e}17\text{e}6\text{b}3\text{d}1\text{bfae}$

SUBMITTING TO THE ORACLE

$$
\begin{aligned}
c' &= c_r * c \\
&= \mathrm{e4faf5884371e14767f7f9ad1731dba8b}
\end{aligned}
$$

RESPONSE FROM THE ORACLE

$$m' = 25ac321f583b6b15760e4eb569207f2142$$

$$m = r^{-1}m'$$

$$= 546869732069732061207465737 4$$

BLEICHENBACHER '06

RSA SIGNATURES IN PRACTICE

- Don't sign the message, but hash of message
- Hashes are small so pad hash before signing

PKCS1.5 PADDING

00 01 FF FF ... FF 00 ASN.1 HASH

IMPROPER VERICATION

00 01 FF FF ... FF 00 ASN.1 HASH GARBAGE

DESCRIPTION OF VULERNABILITY

- Improper parsing during verification
- Choosing small public exponent such as 3
- Can exploit this in one of two ways
  - Through clever choice of values A, B and the fact that
    $(A - B)^3 = A^3 - 3A^2B + 3AB^2 - B^3$
  - Take cube root of 00 01 FF 00 ASN.1 HASH GARBAGE and round

EXAMPLE MODULUS

```
9016dafb596855c95c0d01e95d2ada94
e5f702b7d081fd708ea205e0811cf4a2
9e8ffa1e64ca1bc3cdf7d21512b654b9
f4bac42c52fb99595968f11ec901b8b3
eb86ed0d96d4c8d1adbcf2fdc5a2bb6f
2254a406c62a627bf348d572dc284f02
ab3e0b5f9f00c217a1d97b15e3bf0399
b403b54f857f54d0dc57831cc717c427
dde4aeccaa9af05b4bffce39b4c63a8b
922dfd82320c2873facdf7c6e4ca092f
a540c9db77c1d176f27b2d89a40f7041
320581056ba95bf18aac584c92b9ade6
2a80bfae78e6d79078d5f2bfefb89e94
765745c95770fcdb3c690301002a1acb
f9fecfbfe95e625a6ef9c87a97edb5ea
57b713736f57596ef3c79ecbe7f89a0b
```

FORGING A SIGNATURE

- Forge a signature for the message "This is a test message"
- Public exponent $e = 3$

EXAMPLE TAG

```
32c38624845911817bf125c3cf15343cee
1389f8cf00f9ae3781a29e611d939748cf
6f7578fc47f53fa6a24e9126607f37741f
78a7a418741a849eb4877450eefe609eb8
16213789bb9ccafa891e5d5aa38e68945d
```

VERIFICATION BY CUBING



EXAMPLES FROM THE REAL WORLD

- CVE-2007-6721 Bouncy Castle Java API
- CVE-2016-1494 python-rsa (Bleichenbacher '06 variant)

BAD SIGNATURE WHEN USING SUNZI'S THEOREM

SUNZI'S THEOREM AND RSA

- Knowing private exponent d as well as public exponent e and modulus n equivalent to knowing and q
- $m \mod n \mapsto (m \mod p, m \mod q)$ so $s = m^d \mod n \mapsto (m^{d_p} \mod p, m^{d_q} \mod$
- $d_p = d \mod p - 1$ and likewise $d_q = \mod q - 1$
- Much more efficient than working mod n

FAULT IN ONE COMPONENT

- Suppose $m^{d_p} \mod p$ is correct, but $m^{d_q} \mod q$ is incorrect
- $m \neq s^e \mod n$
- $m - s^e \mod n$ is divisible by p, but not q
- $p = \gcd(m - s^e \mod n, n)$

EXAMPLE: RSA PUBLIC KEY

- $e = 10001$
- $n =$

```
cdd806846f7a82de994760411d44e45e
ae6a99da3f25eb28cc1a44d5269bb2cc
dc5cd89afe7df8c55b0ca4747757d8d3
897881616e462b8b3d0124464e5efcd2
12a4e82ba75c9a8b49623ac675e10e12
68c0f54bd0ee154bf1843f7b23e56f7b
5ff33579a25e1f0c36a4b0346573d795
47f99d411089f638e8cc7fa00c596d5b
```

BAD SIGNATURE

- Message: "This is a test message"
- Using SHA-256
- Signature:

```
330a05472cd5c6188c78f85444efed2b
45d1b7930afc815a69c09948ea4d7991
9b6b610a680e19dce936ab56b937f522
0888b624cbd8c8a2e78f47b84fdb9cc1
15b6205d50326507b7e3ce59b17fa9fd
8436b2946a089a01f96a372c0734b647
c874381f7a04e7a243c10633bdf51c7a
559e5d918b17dbe8d34e806715e1201f
```

VERIFICATION PRODUCES

```
56a526f2975e45d80a35b3f2e1480217
83004e5dfb5499bcfb403b662a4d87cc
fc9c210158bf32a8ff228570bae7069e
6f6cbea49c8a716ef0d7d4ea0a68ae61
65034d58bdf8a62463a1c02a72caaa12
9f04758d8d60c5e1a9bf721eee492d8f
d754e7e3d75b19cb0839a515db297e66
42051384db00f47f49da3b712aab4ebe
```

SHOULD BE

```
0001ffffffffffffffffffffffffff
ffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffff
ffffffffffffffffffffff00303130
0d0609608648016503040201050004206f3438001129a90c5b1637928bf38bf26e39e57c6e9511005682048bedbef906
```

DIFFERENCE


56a326f2975e45d80a35b3f2e1480217
83004e5dfb5499bcfb403b662a4d87cc
fc9c210158bf32a8ff228570bae7069e
6f6cbea49c8a716ef0d7d4ea0a68ae61
65034d58bdf8a62463a1c02b729a78e2
91fe6c2d0718c47ca6bb701de949296f
6820afe3c63170bead236d834f35f273
d3cb2e086c6be37ef35836e53cec55b8

COMPUTE GCD

- gcd=p=

```
d988c222a820f802390e94e7580e7c9c
7a1b04239b6b73c6f321fdc7b0fc19bb
1c5bbfc05f2b74a5886d90db9b9a87d3
bf12888238555a9f94f42a61cf3208e3
```

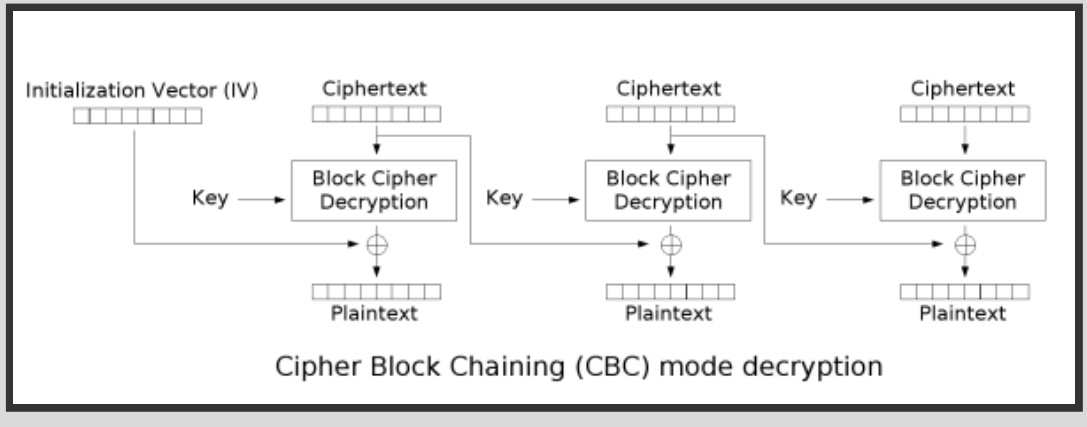
- q=

```
f23e12af65391aee2efabbb76d8d5ec8
d477772c1594be55d894debe1039b8a5
70070009fc48d50fcc8e4aefd3c8c17c
fb7af77c906972c7e5c12af322aecb29
```

BONUS SECTION

WARM-UP: KEY = IV IN CBC MODE

- Ciphertext blocks: $C_1 C_2 .. C_n$
- Submit $C_1 \mathbf{00} C_1$ to a decryption oracle
- Recover IV=key



EXAMPLE

- Key = IV = "YELLOW SUBMARINE"
- Message = "This is a sample message"
- Ciphertext =

```
e9c0fce699eef3477423750e4f29db71
4919caca7f6be14b131d677c0159c8e8
```

- Decrypt:

```
e9c0fce699eef3477423750e4f29db71
00000000000000000000000000000000
e9c0fce699eef3477423750e4f29db71
```

RESPONSE FROM ORACLE

- Decryption:

```
546869732069732061207361 6d706c65
84309b42189a8aebdb905f5fae271f33
0d2d253f6f3e537334623e203f392220
```

- Compute XOR of:

```
546869732069732061207361 6d706c65
0d2d253f6f3e537334623e203f392220
```

WIENER'S ATTACK

PRELIMINARIES

- $ed = 1 \mod \operatorname{lcm}((p-1),(q-1))$
- $ed = K \cdot \operatorname{lcm}((p-1)(q-1)) + 1$
- If we let $G = \gcd((p-1)(q-1))$ then $ed = \frac{K}{G}(p-1)(q-1)+1$
- Let $k = \frac{K}{\gcd(K,G)}$ and $g = \frac{G}{\gcd(K,G)}$
- $ed = \frac{k}{g}(p-1)(q-1)+1$
- $\frac{e}{pq} = \frac{k}{dg}(1-\delta)$ with $\delta = \frac{p+q-1-\frac{g}{k}}{pq}$

OUTLINE OF THE ATTACK

- $\frac{e}{pq}$ is made up of public values and approximates $\frac{k}{dg}$
- Use Continued Fractions algorithm to find $\frac{k}{dg}$
- When checking our guesses we will find d, p, q when d is sufficiently small
- See handout for algorithm details

DETECTING THE VULNERABILITY

Very large public exponent indicates a likely small private exponent

BATCH GCD

ENTROPY REUSE

- During key generation one prime may be chosen that was chosen for another key
- Computing the GCD of such modulii will find the common prime and thus factor each
- Batch GCD efficiently computes GCD of each modulus and the product of the rest in a large list

THE ALGORITHM

- Compute product $P = \prod_i n_i$ of modulii using product tree
- Compute $z_i = P \mod n_i^2$ using remainder tree
- Compute $\gcd(n_i, z_i/n_i)$

REFERENCES

- https://en.wikipedia.org/wiki/Block_cipher_mode_o
- https://en.wikipedia.org/wiki/Digital_Signature_Alg
- http://www.nilsschneider.net/2013/01/28/recoverin private-keys.html
- https://en.wikipedia.org/wiki/RSA_(cryptosystem)
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.581.5957&rep=rep1&type=pdf
- https://factorable.net/weakkeys12.extended.pdf

APPENDIX

DSA KEY GENERATION

- Choose cryptographic hash function H and bit lengths N and L
- Choose N-bit prime q where N is less than the length of the hash function
- Choose L-bit prime p where q divides $p-1$
- Choose number g whose order mod p is q
- Choose random x with $0<x<q$
- Compute $y = g^x \mod p$
- Public key is (p, q, g, y)
- Private key is x

RSA KEY GENERATION

- Choose two distinct primes p and q at random and similar in magnitude, but differ in length by a few digits
- Compute $n = pq$
- Compute $\varphi(n) = (p-1)(q-1) = n - (p+q-1)$
- Choose an integer e such that 1<e< $\varphi(n)$
- Find $d = e^{-1} \mod \varphi(n)$
- Public Key is (e, n), private key is d