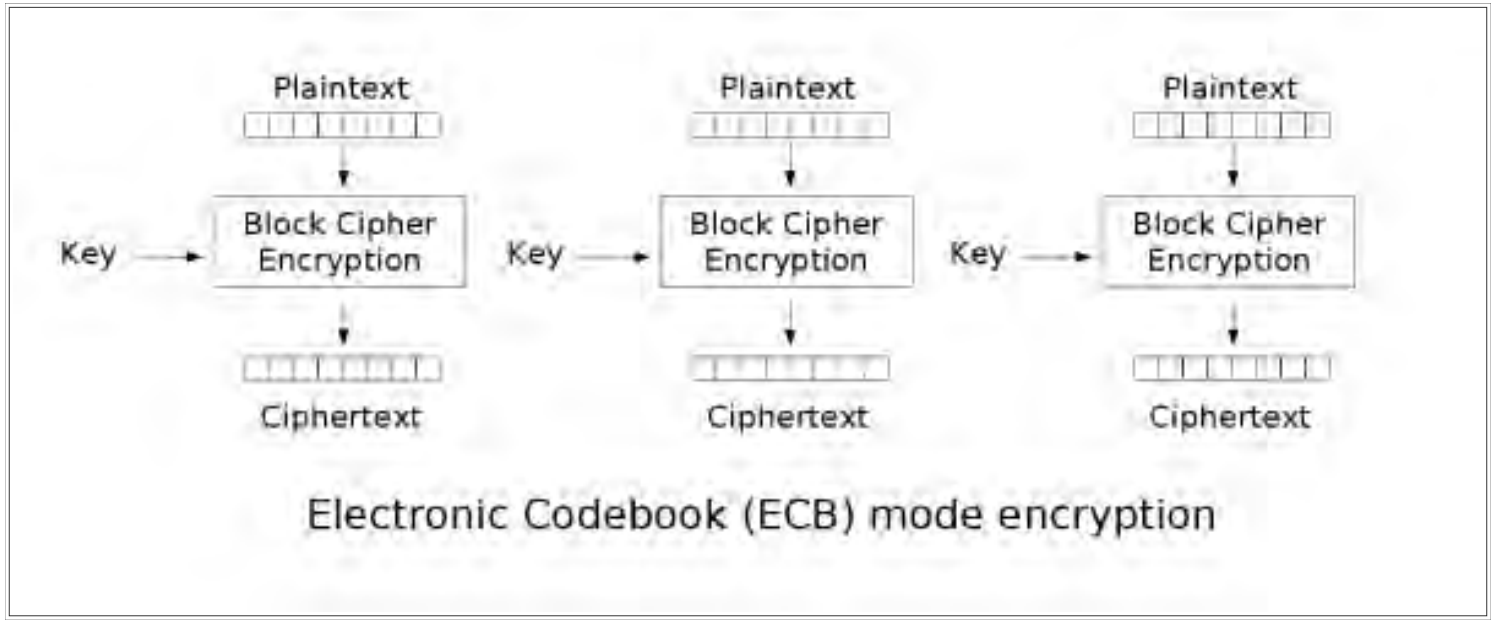
# INTRODUCTION TO CRYPTOGRAPHIC ATTACKS

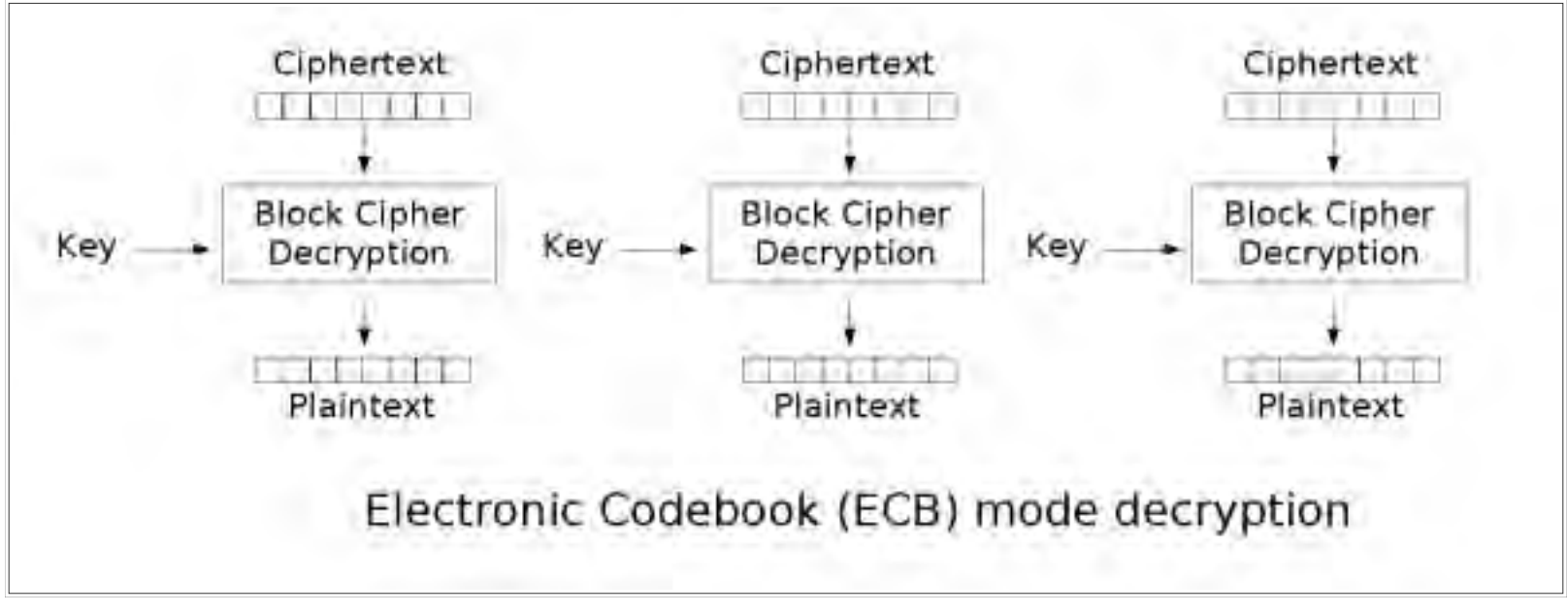
# BLOCK CIPHERS

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

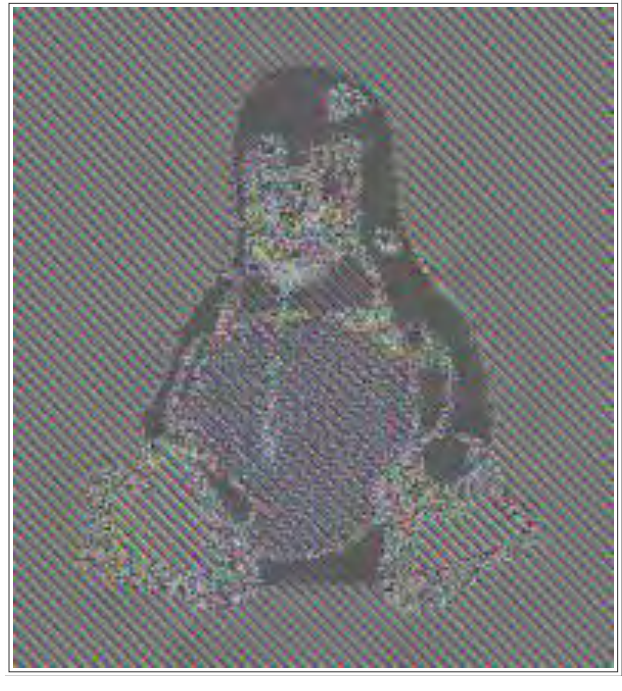
# ECB MODE ENCRYPTION



# ECB MODE DECRYPTION

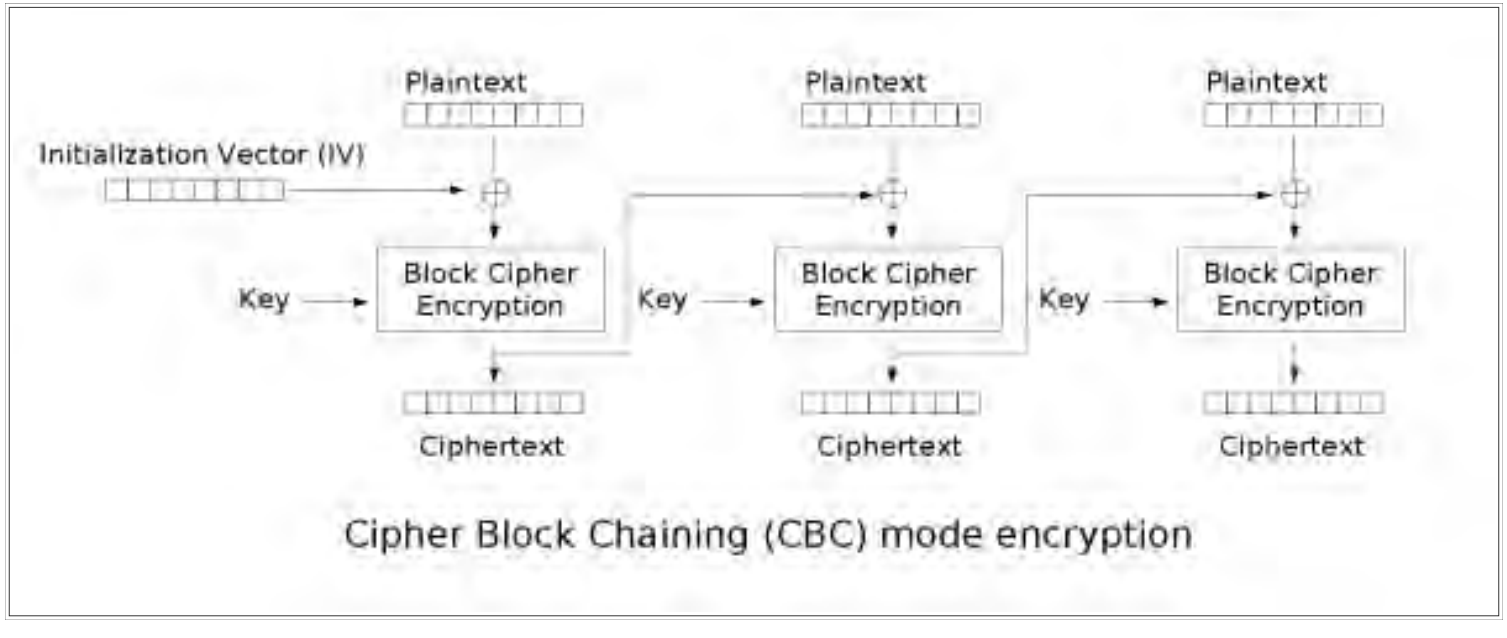


# PENGUINS AND ELECTRONIC CODE BOOK

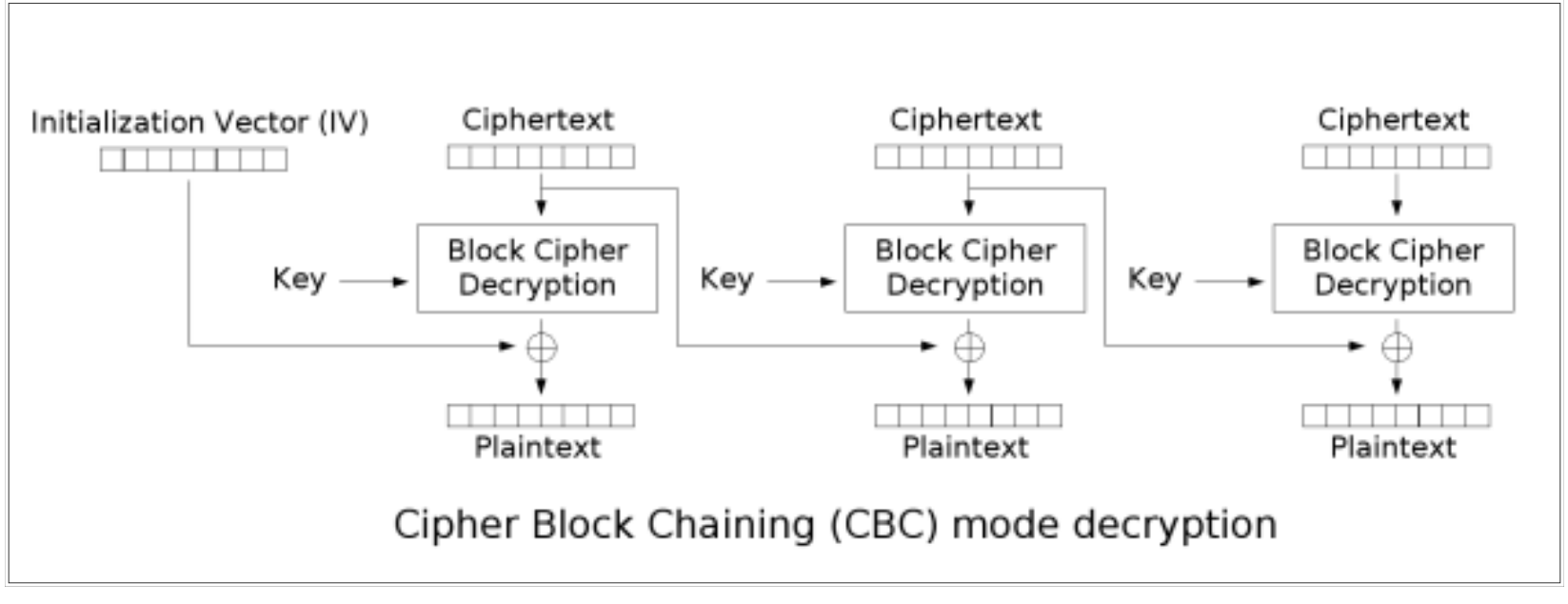


# CIPHER BLOCK CHAINING

# CBC MODE ENCRYPTION

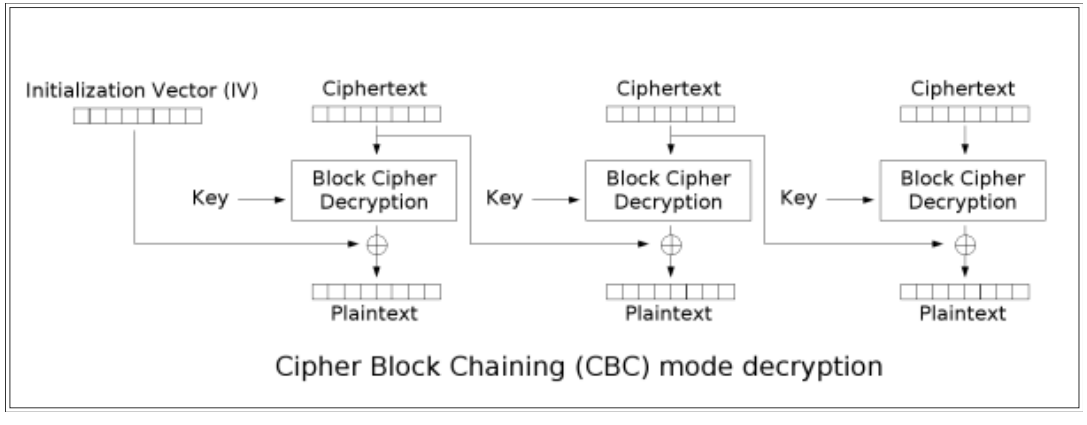


# CBC MODE DECRYPTION



# 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



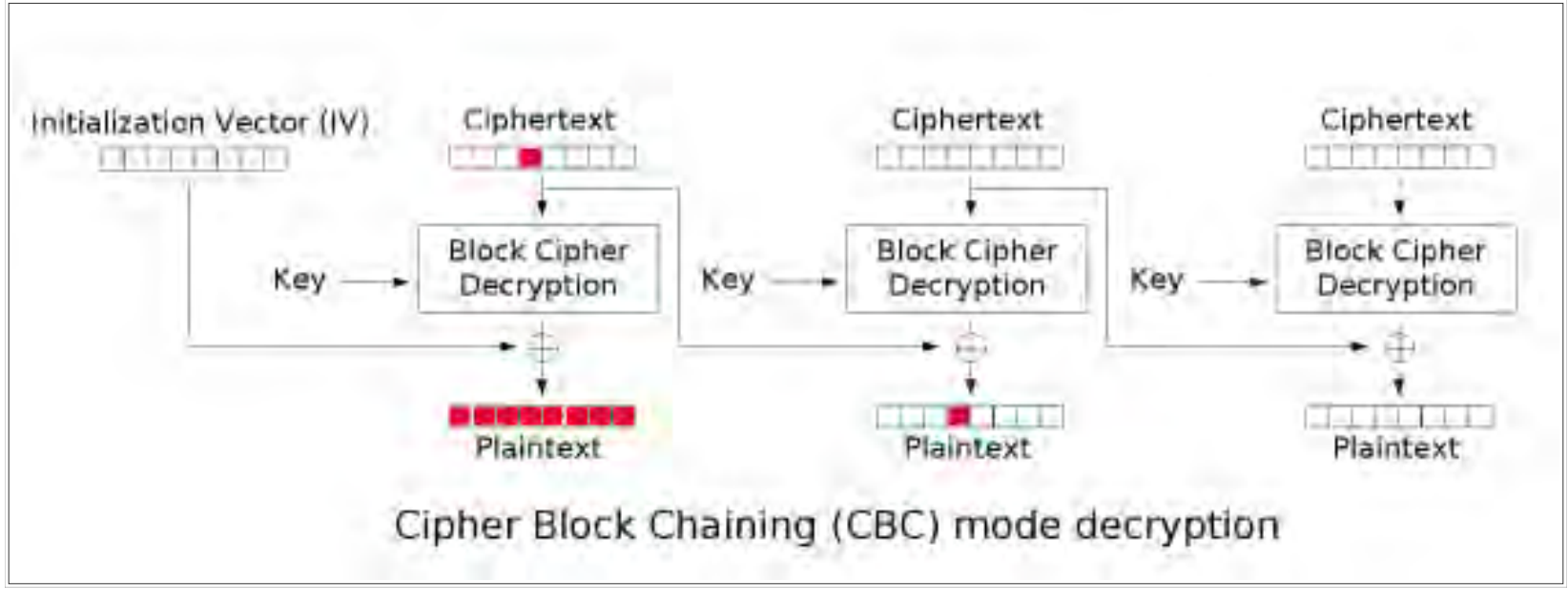
# PADDING ORACLE ATTACK

# 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

```
ctext_blocks[-2][-1] ^= guess^0x01
```

# 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
- Apache mod_session_crypto CVE-2016-0736

# (EC)DSA

## HOW DOES IT WORK?

# KEY GENERATION

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

# 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

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

# REPEATED NONCE ATTACK

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

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

$$x = (r(s_2 - s_1))^{-1}(s_1 H(m_2) - s_2 H(m_1)) \mod q$$

# EXAMPLES OF THE VULNERABILITY

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

# RSA

# 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 $d$

# 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

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

# 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 $m'$
- $m = r^{-1} m' \mod n$

# 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

# EXAMPLES OF VULNERABILITY

- CVE-2007-6721 Bouncy Castle Java API
- CWE-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 $p$ and $q$
- $m \mod n \mapsto (m \mod p, m \mod q)$ so $s = m^d \mod n \mapsto (m^{d_p} \mod p, m^{d_q} \mod q)$
- $d_p = d \mod p$ and likewise $d_q$
- 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 $q$
- $p = \gcd(m - s^e \mod n, n)$

# 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 $\gcd(n_i, z_i/n_i)$