Factoring and RSA

Nadia Heninger

University of Pennsylvania

September 18, 2017

*Some slides joint with Dan Bernstein and Tanja Lange

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

R.L. Rivest, A. Shamir, and L. Adleman*

Textbook RSA

[Rivest Shamir Adleman 1977]

Public Key

$N = pq$ modulus

e encryption exponent

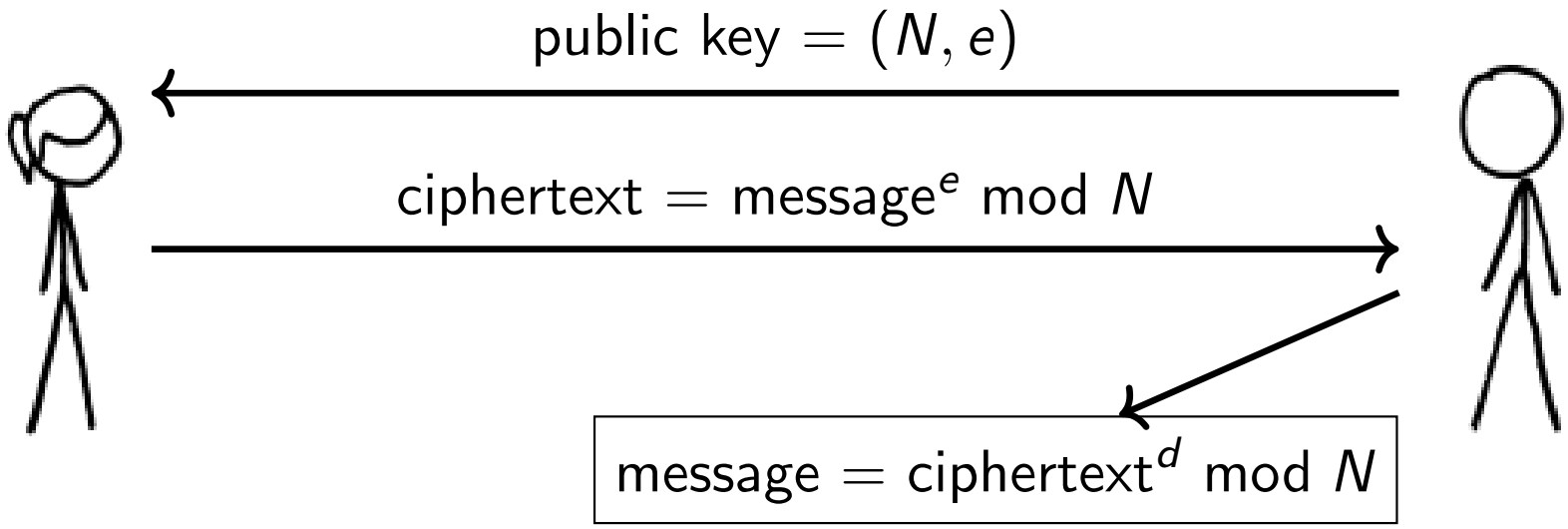
Private Key

p, q primes

d decryption exponent

$(d = e^{-1} \bmod (p-1)(q-1))$

Encryption



Textbook RSA

[Rivest Shamir Adleman 1977]

Public Key

$N = pq$ modulus

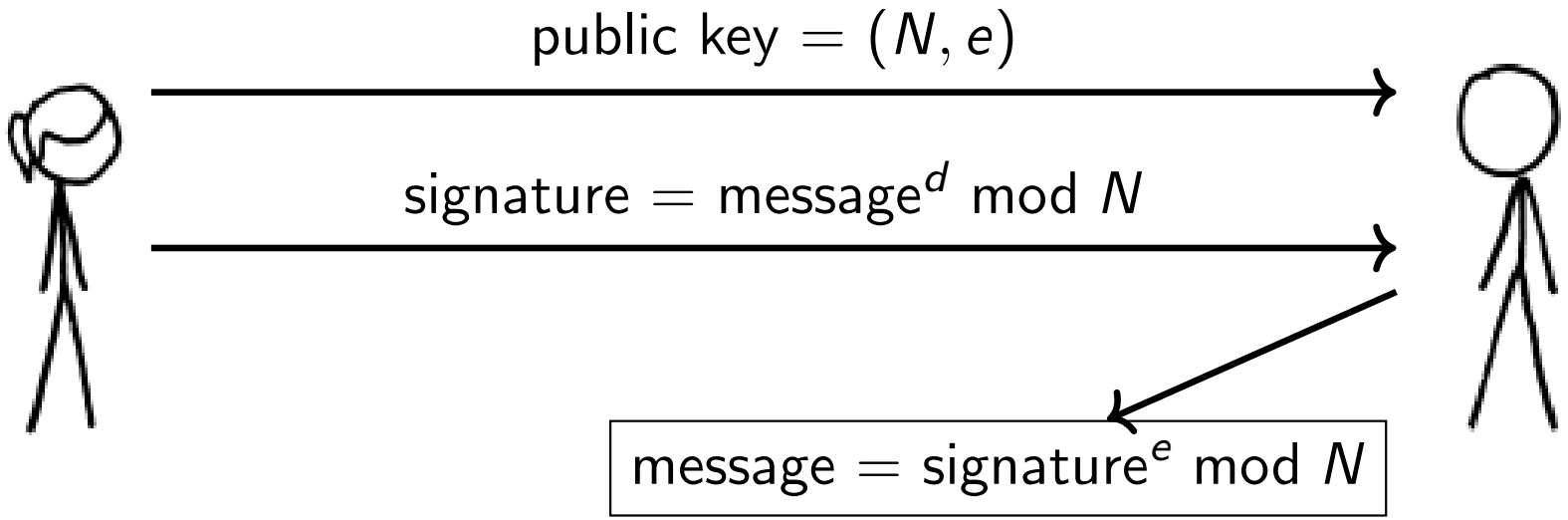
e encryption exponent

Private Key

p, q primes

d decryption exponent
$(d = e^{-1} \bmod (p-1)(q-1))$

Signing



Computational problems

Factoring

Problem: Given N, compute its prime factors.

- Computationally equivalent to computing private key d.
- Factoring is in NP and coNP $\rightarrow$ not NP-complete (unless P=NP or similar).

Computational problems

eth roots mod N

Problem: Given N, e, and c, compute x such that $x^e \equiv c \bmod N$.

- Equivalent to decrypting an RSA-encrypted ciphertext.
- Equivalent to selective forgery of RSA signatures.
- Unknown whether it reduces to factoring:
  - "Breaking RSA may not be equivalent to factoring" [Boneh Venkatesan 1998]
    "an algebraic reduction from factoring to breaking low-exponent RSA can be converted into an efficient factoring algorithm"
  - "Breaking RSA generically is equivalent to factoring" [Aggarwal Maurer 2009]
    "a generic ring algorithm for breaking RSA in $\mathbb{Z}_N$ can be converted into an algorithm for factoring"
- "RSA assumption": This problem is hard.

A garden of attacks on textbook RSA

Unpadded RSA encryption is homomorphic under multiplication. Let's have some fun!

Attack: Malleability

Given a ciphertext $c = \text{Enc}(m) = m^e \bmod N$, attacker can forge ciphertext $\text{Enc}(ma) = ca^e \bmod N$ for any a.

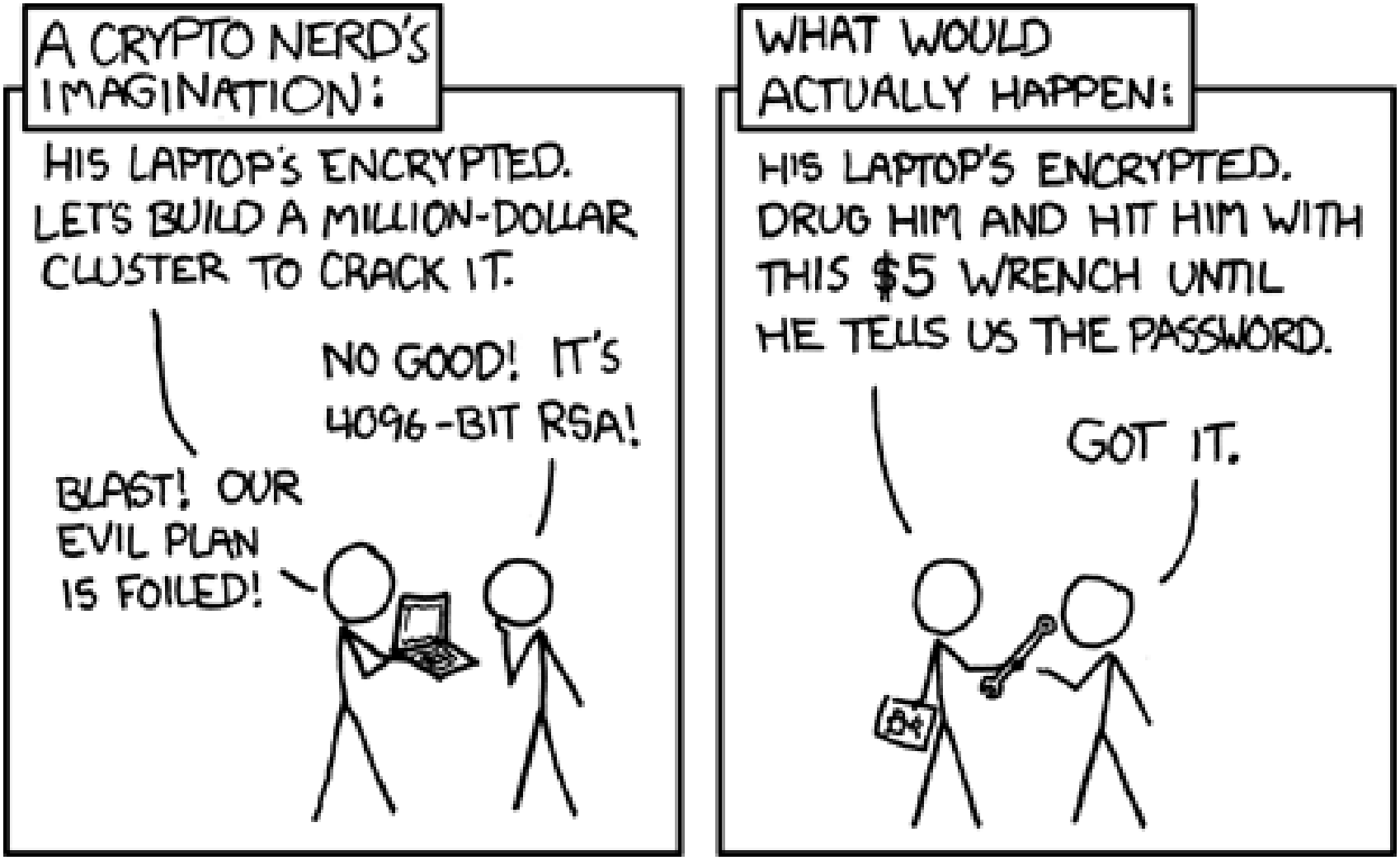
Attack: Chosen ciphertext attack

Given a ciphertext $c = \text{Enc}(m)$ for unknown m, attacker asks for $\text{Dec}(ca^e \bmod N) = d$ and computes $m = da^{-1} \bmod N$.

Attack: Signature forgery

Attacker wants $\text{Sign}(x)$. Attacker computes $z = xy^e \bmod N$ for some y and asks signer for $s = \text{Sign}(z) = z^d \bmod N$. Attacker computes $\text{Sign}(z) = sy^{-1} \bmod N$.

So in practice **always use padding on messages**.



http://xkcd.com/538/

Preliminaries: Using Sage

Working code examples will be given in Sage.

> Sage is free open source mathematics software.
> Download from `http://www.sagemath.org/`.

Sage is based on Python

```
sage: 2*3
6
```

Preliminaries: Using Sage

Working code examples will be given in Sage.

> Sage is free open source mathematics software.
> Download from `http://www.sagemath.org/`.

Sage is based on Python, but there are a few differences:

^ is exponentiation, not xor

```
sage: 2^3
8
```

Preliminaries: Using Sage

Working code examples will be given in Sage.

> Sage is free open source mathematics software.
> Download from `http://www.sagemath.org/`.

Sage is based on Python, but there are a few differences:

^ is exponentiation, not xor

```
sage: 2^3
8
```

It has lots of useful libraries:

```
sage: factor(15)
3 * 5
```

Preliminaries: Using Sage

Working code examples will be given in Sage.

> Sage is free open source mathematics software.
> Download from `http://www.sagemath.org/`.

Sage is based on Python, but there are a few differences:

^ is exponentiation, not xor

```
sage: 2^3
8
```

It has lots of useful libraries:

```
sage: factor(15)
3 * 5
```

```
sage: factor(x^2-1)
(x - 1) * (x + 1)
```

Practicing Sage and Textbook RSA

Key generation:

```
sage: p = random_prime(2^512); q = random_prime(2^512)
sage: N = p*q
sage: e = 65537
sage: d = inverse_mod(e,(p-1)*(q-1))
```

Encryption:

```
sage: m = Integer('helloworld',base=35)
sage: c = pow(m,65537,N)
```

Decryption:

```
sage: Integer(pow(c,d,N)).str(base=35)
'helloworld'
```

So how hard *is* factoring?

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
CPU times: user 1.63 ms, sys: 37 s, total: 1.67 ms
Wall time: 1.66 ms
1235716393 * 4051767059
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
CPU times: user 1.63 ms, sys: 37 s, total: 1.67 ms
Wall time: 1.66 ms
1235716393 * 4051767059

sage: time factor(random_prime(2^64)*random_prime(2^64))
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
CPU times: user 1.63 ms, sys: 37 s, total: 1.67 ms
Wall time: 1.66 ms
1235716393 * 4051767059

sage: time factor(random_prime(2^64)*random_prime(2^64))
CPU times: user 92.5 ms, sys: 16.3 ms, total: 109 ms
Wall time: 163 ms
12072631544896004447 * 13285534720168965833
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
CPU times: user 1.63 ms, sys: 37 s, total: 1.67 ms
Wall time: 1.66 ms
1235716393 * 4051767059

sage: time factor(random_prime(2^64)*random_prime(2^64))
CPU times: user 92.5 ms, sys: 16.3 ms, total: 109 ms
Wall time: 163 ms
12072631544896004447 * 13285534720168965833

sage: time factor(random_prime(2^96)*random_prime(2^96))
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
CPU times: user 1.63 ms, sys: 37 s, total: 1.67 ms
Wall time: 1.66 ms
1235716393 * 4051767059

sage: time factor(random_prime(2^64)*random_prime(2^64))
CPU times: user 92.5 ms, sys: 16.3 ms, total: 109 ms
Wall time: 163 ms
12072631544896004447 * 13285534720168965833

sage: time factor(random_prime(2^96)*random_prime(2^96))
CPU times: user 6.03 s, sys: 145 ms, total: 6.18 s
Wall time: 6.35 s
39863518068977786560464995143 * 40008408160629540866839699141
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
CPU times: user 1.63 ms, sys: 37 s, total: 1.67 ms
Wall time: 1.66 ms
1235716393 * 4051767059

sage: time factor(random_prime(2^64)*random_prime(2^64))
CPU times: user 92.5 ms, sys: 16.3 ms, total: 109 ms
Wall time: 163 ms
12072631544896004447 * 13285534720168965833

sage: time factor(random_prime(2^96)*random_prime(2^96))
CPU times: user 6.03 s, sys: 145 ms, total: 6.18 s
Wall time: 6.35 s
39863518068977786560464995143 * 40008408160629540866839699141

sage: time factor(random_prime(2^128)*random_prime(2^128))
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
CPU times: user 1.63 ms, sys: 37 s, total: 1.67 ms
Wall time: 1.66 ms
1235716393 * 4051767059

sage: time factor(random_prime(2^64)*random_prime(2^64))
CPU times: user 92.5 ms, sys: 16.3 ms, total: 109 ms
Wall time: 163 ms
12072631544896004447 * 13285534720168965833

sage: time factor(random_prime(2^96)*random_prime(2^96))
CPU times: user 6.03 s, sys: 145 ms, total: 6.18 s
Wall time: 6.35 s
39863518068977786560464995143 * 40008408160629540866839699141

sage: time factor(random_prime(2^128)*random_prime(2^128))
CPU times: user 7min 56s, sys: 5.38 s, total: 8min 2s
Wall time: 8min 12s
710441398673820995839650640848265404 41 * 950912147141503934646
```

Factoring in practice

Two families of factoring algorithms:

1. Algorithms whose running time depends on the size of the factor to be found.
   - Good for factoring small numbers, and finding small factors of big numbers.
2. Algorithms whose running time depends on the size of the number to be factored.
   - Good for factoring big numbers with big factors.

Trial Division

Good for finding very small factors

Takes $p/\log p$ trial divisions to find a prime factor p.

Pollard rho

Good for finding slightly larger prime factors

Intuition

- Try to take a random walk among elements $\bmod N$.
- If p divides N, there will be a cycle of length p.
- Expect a collision after searching about $\sqrt{p}$ random elements.

Pollard rho

Good for finding slightly larger prime factors

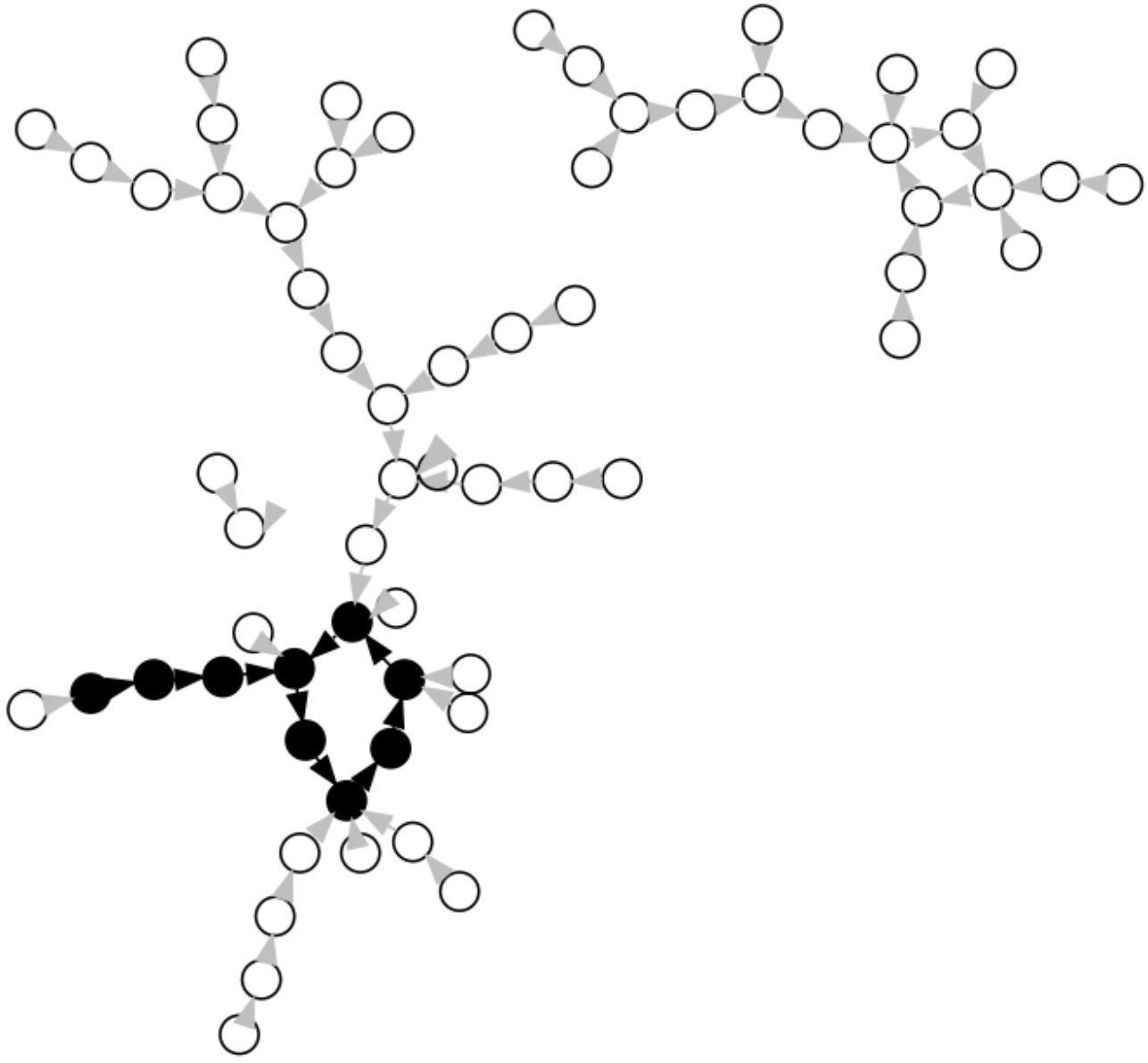
Intuition

- Try to take a random walk among elements $\bmod N$.
- If p divides N, there will be a cycle of length p.
- Expect a collision after searching about $\sqrt{p}$ random elements.

Details

- "Random" function: $f(x) = x^2 + c \bmod N$ for random c.
- For random starting point a, compute $a, f(a), f(f(a)), \ldots$
- Naive implementation uses $\sqrt{p}$ memory, $O(1)$ lookup time.
- To reduce memory:
  - Floyd cycle-finding algorithm: Store two pointers, and move one twice as fast as the other until they coincide.
  - Method of distinguished points: Store points satisfying easily tested property like k leading zeros.

Why is it called the rho algorithm?



Pollard rho in Sage

```
def rho(n):
    a = 9835738947594387 5; c=10 # some random values
    f = lambda x: (x^2+c)%n

    a1 = f(a) ; a2 = f(a1)
    while gcd(n, a2-a1)==1:
        a1 = f(a1); a2 = f(f(a2))
    return gcd(n, a2-a1)

sage: N = 6985996992886866654903080690574201382238 71
sage: rho(N)
2053
```

Reminders: Orders and groups

Theorem (Fermat's Little Theorem)

$a^{p-1} \equiv 1 \bmod p$ *for any* $0 < a < p$.

Let $\mathrm{ord}(a)_p$ be the order of a mod p. (Smallest positive integer such that $a^{\mathrm{ord}(a)_p} \equiv 1 \bmod p$.)

Theorem (Lagrange)

$\mathrm{ord}(a)_p$ *divides* $p - 1$.

Pollard's $p-1$ method

Good for finding special small factors

Intuition

- If $a^r \equiv 1 \bmod p$ then $\mathrm{ord}(a)_p \mid r$ and $p \mid \gcd(a^r - 1, N)$.
- Don't know p, pick very smooth number r, hoping for $\mathrm{ord}(a)_p$ to divide it.

Definition: An integer is *B-smooth* if all its prime factors are $\leq B$.

Pollard's $p-1$ method

Good for finding special small factors

Intuition

- If $a^r \equiv 1 \bmod p$ then $\mathrm{ord}(a)_p \mid r$ and $p \mid \gcd(a^r - 1, N)$.
- Don't know p, pick very smooth number r, hoping for $\mathrm{ord}(a)_p$ to divide it.

Definition: An integer is *B-smooth* if all its prime factors are $\leq B$.

```
N=44426601460658291157725536008128017297890787
463719427903128118036605 7
r=lcm(range(1,2^22)) # this takes a while ...
s=Integer(pow(2,r,N))
sage: gcd(s-1,N)
126765060022822940149670321760 1
```

Pollard $p-1$ method

- This method finds larger factors than the rho method (in the same time)

  ...but only works for special primes.

In the previous example,

$$p-1 = 2^6 \cdot 3^2 \cdot 5^2 \cdot 17 \cdot 227 \cdot 491 \cdot 991 \cdot 36559 \cdot 308129 \cdot 4161791$$

has only small factors (aka. $p-1$ is *smooth*).

- Many crypto standards require using only "*safe primes*" a.k.a primes where $p-1 = 2q$, so $p-1$ is really non-smooth.
- This recommendation is outdated for RSA. The elliptic curve method (next slide) works even for "safe" primes.

Lenstra's Elliptic Curve Method

Good for finding medium-sized factors

Intuition

- Pollard's $p-1$ method works in the multiplicative group of integers modulo p.
- The elliptic curve method is exactly the $p-1$ method, but over the group of points on an elliptic curve modulo p:
  - Multiplication of group elements becomes addition of points on the curve.
  - All arithmetic is still done modulo N.

Lenstra's Elliptic Curve Method

Good for finding medium-sized factors

Intuition

- Pollard's $p-1$ method works in the multiplicative group of integers modulo p.
- The elliptic curve method is exactly the $p-1$ method, but over the group of points on an elliptic curve modulo p:
  - Multiplication of group elements becomes addition of points on the curve.
  - All arithmetic is still done modulo N.

Theorem (Hasse)

The order of an elliptic curve modulo p is in $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$.

There are lots of smooth numbers in this interval.

If one elliptic curve doesn't work, try until you find a smooth order.

Elliptic Curves in Sage

```
def curve(d):
    frac_n = type(d)
    class P(object):
        def __init__(self,x,y):
            self.x,self.y = frac_n(x),frac_n(y)

        def __add__(a,b):
            return P((a.x*b.y + b.x*a.y)/(1 + d*a.x*a.y*b.x*b.y),
                     (a.y*b.y - a.x*b.x)/(1 - d*a.x*b.x*a.y*b.y))

        def __mul__(self, m):
            return double_and_add(self,m,P(0,1))

...
```

Elliptic Curve Factorization

```
def ecm(n,y,t):
    # Choose a curve and a point on the curve.
    frac_n = Q(n)
    P = curve(frac_n(1,3))
    p = P(2,3)

    q = p * lcm(xrange(1,y))
    return gcd(q.x.t,n)
```

- Method runs very well on GPUs.
- Still an active research area.

ECM is very efficient at factoring random numbers, once small factors are removed.

Heuristic running time $L_p(1/2, \sqrt{2}) = O(e^{\sqrt{2}\sqrt{\ln p \ln \ln p}})$.

Quadratic Sieve Intuition: Fermat factorization

Main insight: If we can find two squares a^2 and b^2 such that

$$a^2 \equiv b^2 \bmod N$$

Then

$$a^2 - b^2 = (a + b)(a - b) \equiv 0 \bmod N$$

and we might hope that one of $a + b$ or $a - b$ shares a nontrivial common factor with N.

Quadratic Sieve Intuition: Fermat factorization

Main insight: If we can find two squares a^2 and b^2 such that

$$a^2 \equiv b^2 \bmod N$$

Then

$$a^2 - b^2 = (a+b)(a-b) \equiv 0 \bmod N$$

and we might hope that one of $a+b$ or $a-b$ shares a nontrivial common factor with N.

First try:

1. Start at $c = \lceil \sqrt{N} \rceil$
2. Check $c^2 - N, (c+1)^2 - N, \ldots$ until we find a square.

This is Fermat factorization, which could take up to p steps.

Quadratic Sieve

General-purpose factoring

Intuition

We might not find a square outright, but we can construct a square as a product of numbers we look through.

Quadratic Sieve

General-purpose factoring

Intuition

We might not find a square outright, but we can construct a square as a product of numbers we look through.

1. **Sieving** Try to factor each of $c^2 - N, (c+1)^2 - N, \ldots$
2. Only save a $d_i = c_i^2 - N$ if all of its prime factors are less than some bound B. (If it is *B-smooth*.)
3. Store each d_i by its exponent vector $d_i = 2^{e_2} 3^{e_3} \ldots B^{e_B}$.
4. If $\prod_i d_i$ is a square, then its exponent vector contains only even entries.

Quadratic Sieve

General-purpose factoring

Intuition

We might not find a square outright, but we can construct a square as a product of numbers we look through.

1. **Sieving** Try to factor each of $c^2 - N, (c+1)^2 - N, \ldots$
2. Only save a $d_i = c_i^2 - N$ if all of its prime factors are less than some bound B. (If it is *B-smooth*.)
3. Store each d_i by its exponent vector $d_i = 2^{e_2} 3^{e_3} \ldots B^{e_B}$.
4. If $\prod_i d_i$ is a square, then its exponent vector contains only even entries.
5. **Linear Algebra** Once enough factorizations have been collected, can use linear algebra to find a linear dependence mod2.

Quadratic Sieve

General-purpose factoring

Intuition

We might not find a square outright, but we can construct a square as a product of numbers we look through.

1. **Sieving** Try to factor each of $c^2 - N, (c+1)^2 - N, \ldots$
2. Only save a $d_i = c_i^2 - N$ if all of its prime factors are less than some bound B. (If it is *B-smooth*.)
3. Store each d_i by its exponent vector $d_i = 2^{e_2} 3^{e_3} \ldots B^{e_B}$.
4. If $\prod_i d_i$ is a square, then its exponent vector contains only even entries.
5. **Linear Algebra** Once enough factorizations have been collected, can use linear algebra to find a linear dependence mod2.
6. **Square roots** Take square roots and hope for a nontrivial factorization. Math exercise: Square product has 50% chance of factoring pq.

An example of the quadratic sieve

Let's try to factor $N = 2759$.

An example of the quadratic sieve

Let's try to factor $N = 2759$.

Sieving values $(\lceil \sqrt{N} + i \rceil)^2 \bmod N$:

An example of the quadratic sieve

Let's try to factor $N = 2759$.

Sieving values $(\lceil \sqrt{N} + i \rceil)^2 \bmod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$.

An example of the quadratic sieve

Let's try to factor $N = 2759$.

Sieving values $(\lceil \sqrt{N} + i \rceil)^2 \bmod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$.

An example of the quadratic sieve

Let's try to factor $N = 2759$.

Sieving values $(\lceil \sqrt{N} + i \rceil)^2 \bmod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$.
$55^2 - 2759 = 266$.

An example of the quadratic sieve

Let's try to factor $N = 2759$.

Sieving values $(\lceil \sqrt{N} + i \rceil)^2 \bmod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.

An example of the quadratic sieve

Let's try to factor $N = 2759$.

Sieving values $(\lceil \sqrt{N} + i \rceil)^2 \bmod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.
$57^2 - 2759 = 490 = 2 \cdot 5 \cdot 7^2$.

An example of the quadratic sieve

Let's try to factor $N = 2759$.

Sieving values $(\lceil \sqrt{N} + i \rceil)^2 \bmod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.
$57^2 - 2759 = 490 = 2 \cdot 5 \cdot 7^2$.
$58^2 - 2759 = 605 = 5 \cdot 11^2$.

An example of the quadratic sieve

Let's try to factor $N = 2759$.

Sieving values $(\lceil \sqrt{N} + i \rceil)^2 \bmod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.
$57^2 - 2759 = 490 = 2 \cdot 5 \cdot 7^2$.
$58^2 - 2759 = 605 = 5 \cdot 11^2$.

Linear Algebra: The *product* $50 \cdot 490 \cdot 605$ is a square:
$2^2 \cdot 5^4 \cdot 7^2 \cdot 11^2$.

An example of the quadratic sieve

Let's try to factor $N = 2759$.

Sieving values $(\lceil \sqrt{N} + i \rceil)^2 \bmod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.
$57^2 - 2759 = 490 = 2 \cdot 5 \cdot 7^2$.
$58^2 - 2759 = 605 = 5 \cdot 11^2$.

Linear Algebra: The *product* $50 \cdot 490 \cdot 605$ is a square:
$2^2 \cdot 5^4 \cdot 7^2 \cdot 11^2$.

Recall idea: If $a^2 - N$ is a square b^2 then $N = (a - b)(a + b)$.

QS computes $\gcd\{2759, 53 \cdot 57 \cdot 58 - \sqrt{50 \cdot 490 \cdot 605}\} = 31$.

Quadratic Sieve running time

- How do we choose B?
- How many numbers do we have to try to factor?
- Depends on (heuristic) probability that a randomly chosen number is B-smooth.

Running time: $L_N(1/2, 1) = e^{(1+o(1))\sqrt{\ln N \ln \ln N}}$.

Number field sieve

Best running time for general purpose factoring

Insight

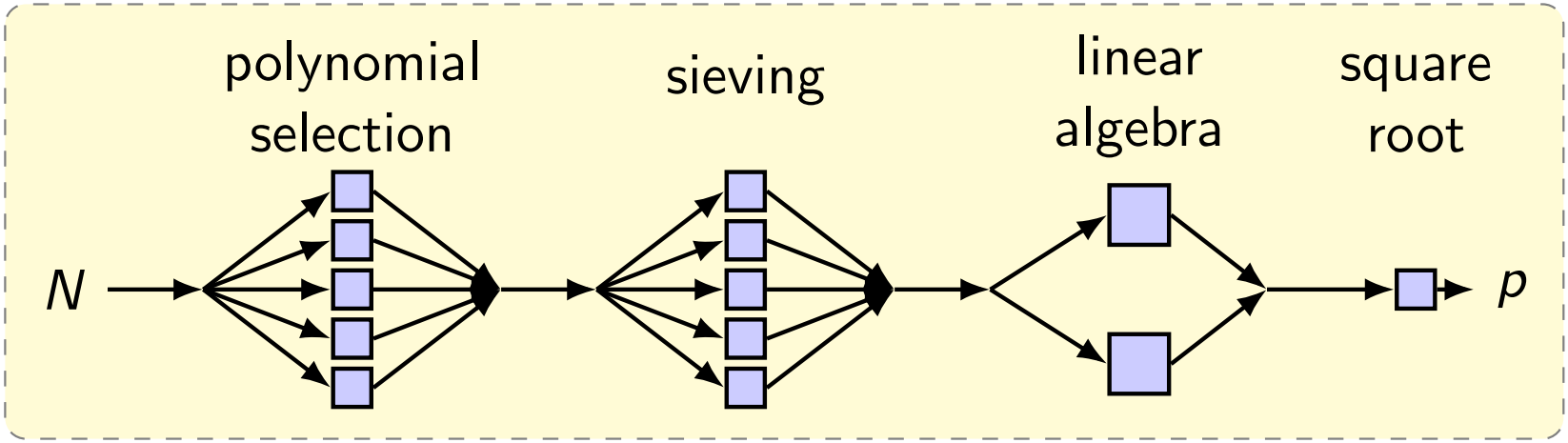
- Replace relationship $a^2 = b^2 \bmod N$ with a homomorphism between ring of integers $\mathcal{O}_K$ in a specially chosen number field and $\mathbb{Z}_N$.

$$\varphi : \mathcal{O}_K \mapsto \mathbb{Z}_N$$

Algorithm

1. **Polynomial selection** Find a good choice of number field K.
2. **Relation finding** Factor elements over $\mathcal{O}_K$ and over $\mathbb{Z}$.
3. **Linear algebra** Find a square in $\mathcal{O}_K$ and a square in $\mathbb{Z}$
4. **Square roots** Take square roots, map into $\mathbb{Z}$, and hope we find a factor.

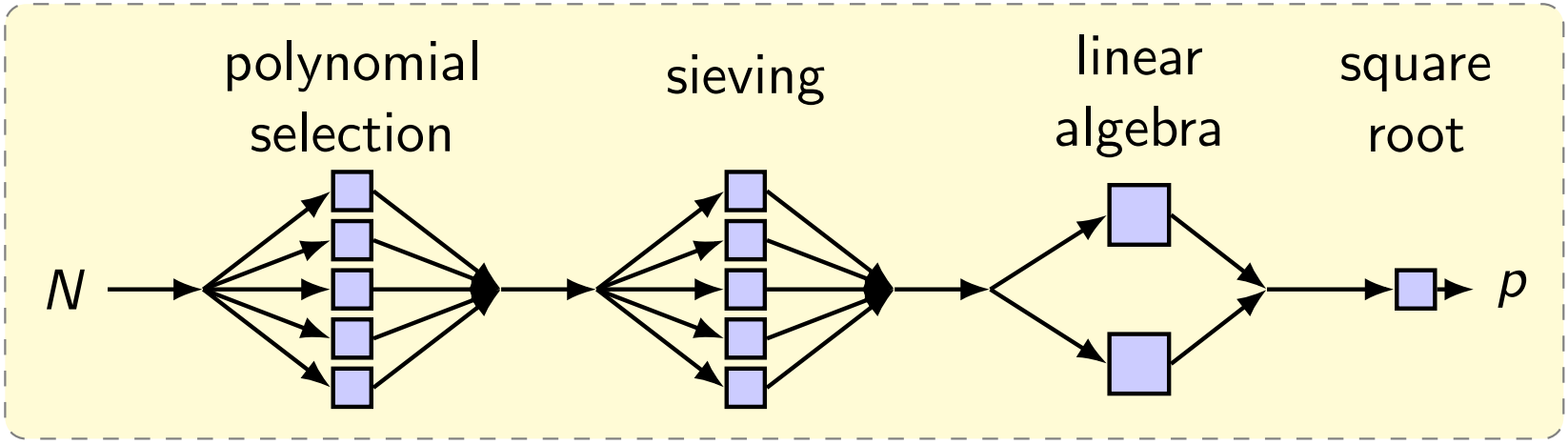
How long does factoring take with the number field sieve?



Answer 1

$$L_N(1/3, \sqrt[3]{64/9}) = e^{(1.923+o(1))(\ln N)^{1/3}(\ln \ln N)^{2/3}}$$

How long does factoring take with the number field sieve?



Answer 2

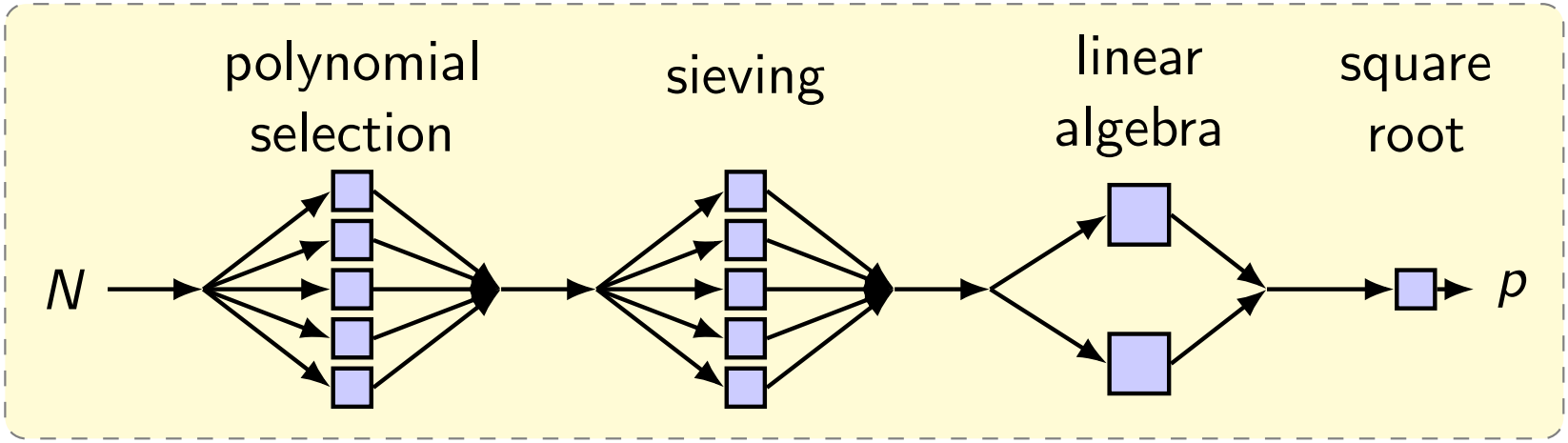
512-bit RSA: $<$ 1 core-year

768-bit RSA: $<$ 1,000 core-years

1024-bit RSA: $\approx$ 1,000,000 core-years

2048-bit RSA: Minimum recommended key size today.

How long does factoring take with the number field sieve?



Answer 3

512-bit RSA: 7 months — large academic effort [CBLLMMtRZ 1999]

768-bit RSA: 2.5 years — large academic effort [KAFLTBGKMOtRTZ 2009]

512-bit RSA: 2.5 months — single machine [Moody 2009]

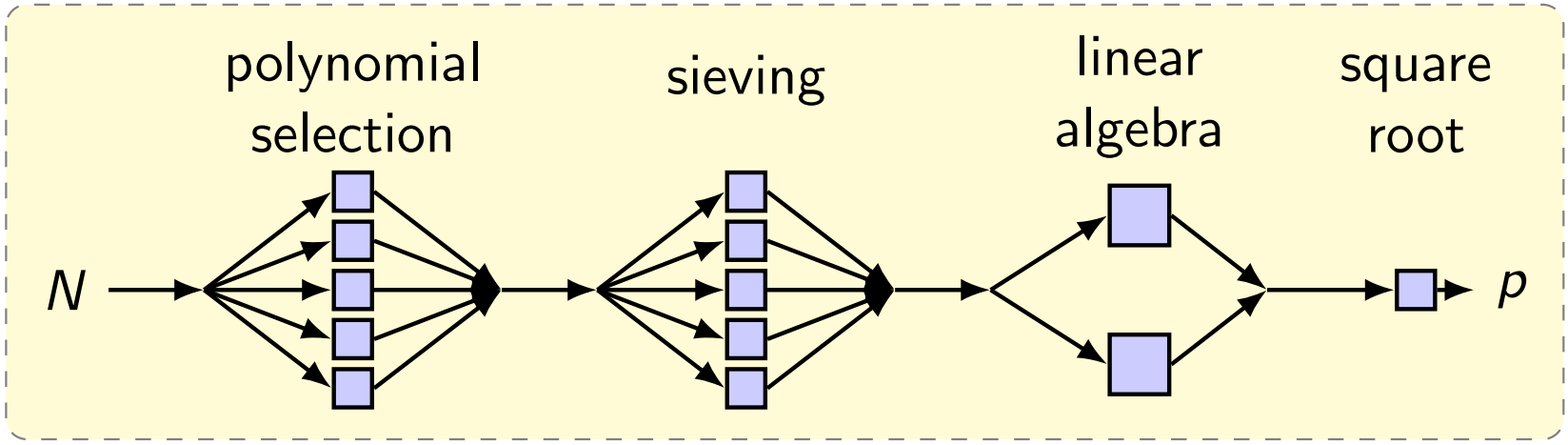
512-bit RSA: 72 hours — single Amazon EC2 machine [Harris 2012]

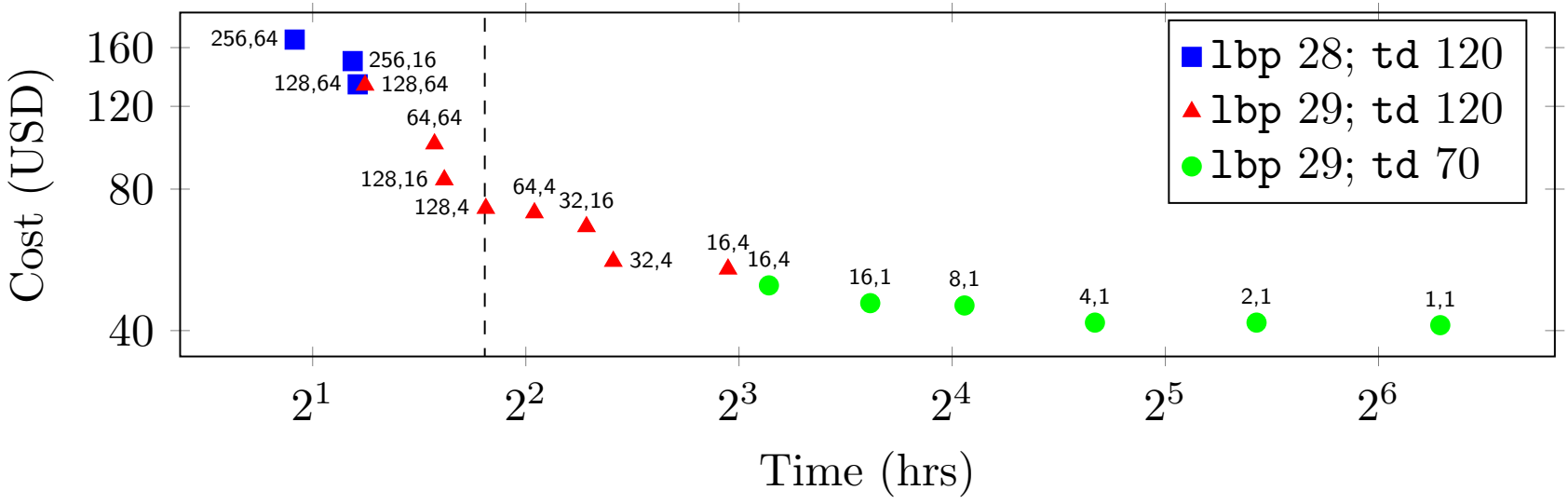
512-bit RSA: 7 hours — Amazon EC2 cluster [Heninger 2015]

512-bit RSA: $<$ 4 hours — Amazon EC2 cluster [VCLFBH 2016]

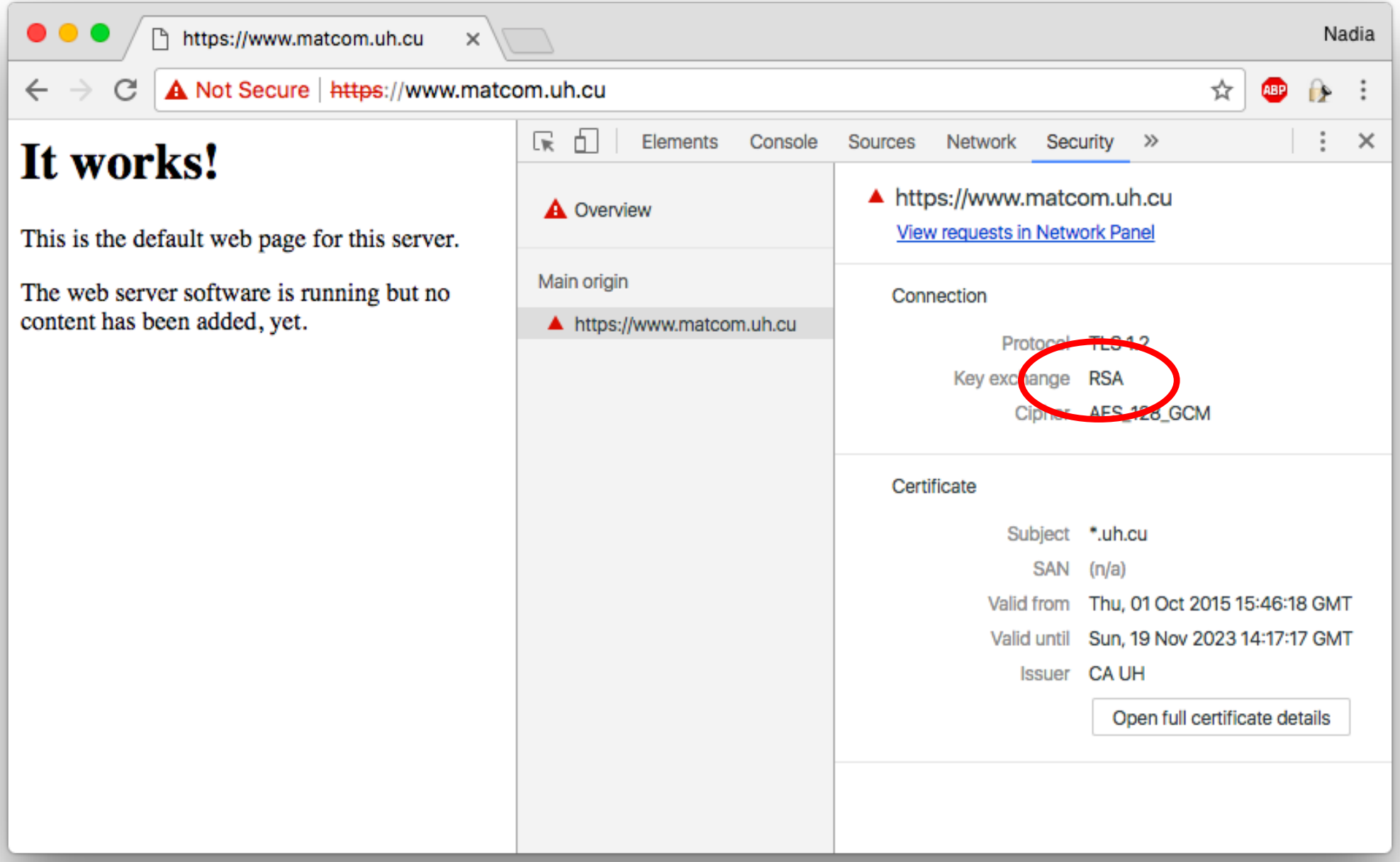
Factoring 512-bit RSA using cloud computing in 2015

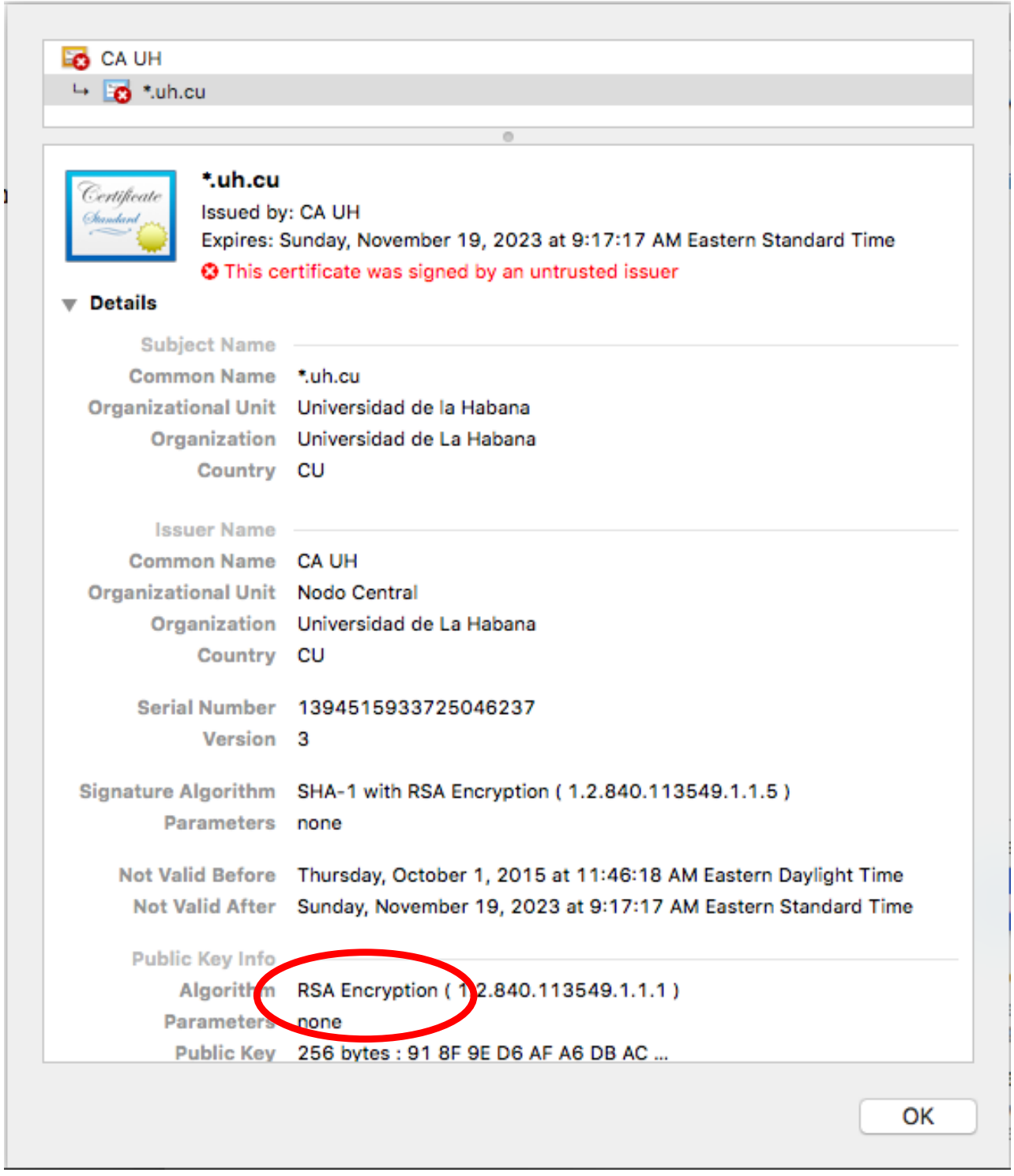
[Valenta Cohney Liao Fried Bodduluri Heninger 2016]





Factoring algorithms in the context of network protocols


https://www.matcom.uh.cu
Nadia
Not Secure | https://www.matcom.uh.cu
It works!
This is the default web page for this server.
The web server software is running but no content has been added, yet.
Elements
Console
Sources
Network
Security
Overview
Main origin
https://www.matcom.uh.cu
https://www.matcom.uh.cu
View requests in Network Panel
Connection
Protocol TLS 1.2
Key exchange RSA
Cipher AES_128_GCM
Certificate
Subject *.uh.cu
SAN (n/a)
Valid from Thu, 01 Oct 2015 15:46:18 GMT
Valid until Sun, 19 Nov 2023 14:17:17 GMT
Issuer CA UH
Open full certificate details


CA UH
*.uh.cu
*.uh.cu
Issued by: CA UH
Expires: Sunday, November 19, 2023 at 9:17:17 AM Eastern Standard Time
This certificate was signed by an untrusted issuer
Details
Subject Name
Common Name *.uh.cu
Organizational Unit Universidad de la Habana
Organization Universidad de La Habana
Country CU
Issuer Name
Common Name CA UH
Organizational Unit Nodo Central
Organization Universidad de La Habana
Country CU
Serial Number 1394515933725046237
Version 3
Signature Algorithm SHA-1 with RSA Encryption (1.2.840.113549.1.1.5)
Parameters none
Not Valid Before Thursday, October 1, 2015 at 11:46:18 AM Eastern Daylight Time
Not Valid After Sunday, November 19, 2023 at 9:17:17 AM Eastern Standard Time
Public Key Info
Algorithm RSA Encryption (1.2.840.113549.1.1.1)
Parameters none
Public Key 256 bytes : 91 8F 9E D6 AF A6 DB AC ...
OK

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]

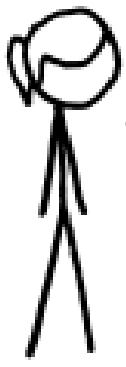
certificate = **RSA pubkey** k_{2048} + CA signatures

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]



certificate = **RSA pubkey** k_{2048} + CA signatures

client key exchange: $\mathbf{RSAenc_{k_{2048}}(pms)}$

client finished: $\text{Auth}_{k_{mc}}(\text{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]

certificate = **RSA pubkey** k_{2048} + CA signatures

client key exchange: $\mathbf{RSAenc_{k_{2048}}(pms)}$

client finished: $\text{Auth}_{k_{m_c}}(\text{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

server finished: $\text{Auth}_{k_{m_s}}(\text{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]

certificate = **RSA pubkey** k_{2048} + CA signatures

client key exchange: $\mathbf{RSAenc_{k_{2048}}(pms)}$

client finished: $\mathrm{Auth}_{k_{m_c}}(\mathrm{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

server finished: $\mathrm{Auth}_{k_{m_s}}(\mathrm{dialog})$

$\mathrm{Enc}_{k_e}(\mathrm{request})$

Does anyone use 512-bit RSA?

International Traffic in Arms Regulations

April 1, 1992 version

Category XIII--Auxiliary Military Equipment ...

(b) Information Security Systems and equipment, cryptographic devices, software, and components specifically designed or modified therefore, including:

(1) Cryptographic (including key management) systems, equipment, assemblies, modules, integrated circuits, components or software with the capability of maintaining secrecy or confidentiality of information or information systems, except cryptographic equipment and software as follows:

(i) Restricted to decryption functions specifically designed to allow the execution of copy protected software, provided the decryption functions are not user-accessible.

(ii) Specially designed, developed or modified for use in machines for banking or money transactions, and restricted to use only in such transactions. Machines for banking or money transactions include automatic teller machines, self-service statement printers, point of sale terminals or equipment for the encryption of interbanking transactions.

...

Question: How do you selectively weaken a protocol based on RSA?

Question: How do you selectively weaken a protocol based on RSA?

Export answer: Optionally use a small RSA key.

Commerce Control List: Category 5 - Info. Security

(From 2015)

a.1.a. A symmetric algorithm employing a key length in excess of 56-bits; or

a.1.b. An asymmetric algorithm where the security of the algorithm is based on any of the following:

a.1.b.1. Factorization of integers in excess of 512 bits (e.g., RSA);

a.1.b.2. Computation of discrete logarithms in a multiplicative group of a finite field of size greater than 512 bits (e.g., Diffie-Hellman over Z/pZ); or

a.1.b.3. Discrete logarithms in a group other than mentioned in 5A002.a.1.b.2 in excess of 112 bits (e.g., Diffie-Hellman over an elliptic curve);

a.2. Designed or modified to perform cryptanalytic functions;

TLS RSA Export Key Exchange

client hello: client random

[. . . RSA_EXPORT . . .]

TLS RSA Export Key Exchange

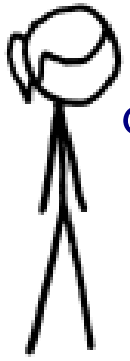
client hello: client random

[. . . RSA_EXPORT . . .]

server hello: server random, [RSA_EXPORT]

certificate = **RSA pubkey k_{2048}** + CA signatures

server key exchange: **RSA pubkey k_{512}**



TLS RSA Export Key Exchange

client hello: client random

[. . . RSA_EXPORT . . .]

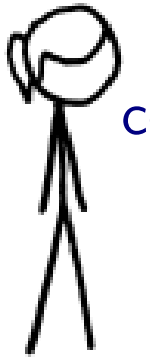
server hello: server random, [RSA_EXPORT]

certificate = **RSA pubkey k_{2048}** + CA signatures

server key exchange: **RSA pubkey k_{512}**

client key exchange: **RSAenc$_{k_{512}}$(pms)**

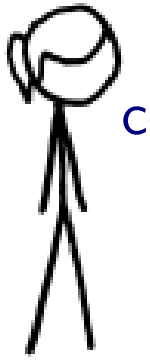
client finished: Auth$_{k_{mc}}$(dialog)



KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Export Key Exchange



client hello: client random

[. . . RSA_EXPORT . . .]

server hello: server random, [RSA_EXPORT]

certificate = **RSA pubkey k_{2048}** + CA signatures

server key exchange: **RSA pubkey k_{512}**

client key exchange: **RSAenc$_{k_{512}}$(pms)**

client finished: Auth$_{k_{mc}}$(dialog)

server finished: Auth$_{k_{ms}}$(dialog)

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

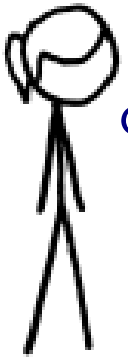
KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Export Key Exchange

client hello: client random

[. . . RSA_EXPORT . . .]

server hello: server random, [RSA_EXPORT]



certificate = **RSA pubkey k_{2048}** + CA signatures

server key exchange: **RSA pubkey k_{512}**

client key exchange: **RSAenc$_{k_{512}}$(pms)**

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

client finished: Auth$_{k_{m_c}}$(dialog)

server finished: Auth$_{k_{m_s}}$(dialog)

Enc$_{k_e}$(request)

RSA export cipher suites in TLS

In March 2015, export cipher suites supported by 36.7% of the 14 million sites serving browser-trusted certificates!

`TLS_RSA_EXPORT_WITH_RC4_40_MD5`
`TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5`
`TLS_RSA_EXPORT_WITH_DES40_CBC_SHA`

Totally insecure, but no modern client would negotiate export ciphers. ... right?

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

[. . . RSA . . .]

FREAK: MITM downgrade attack to export RSA

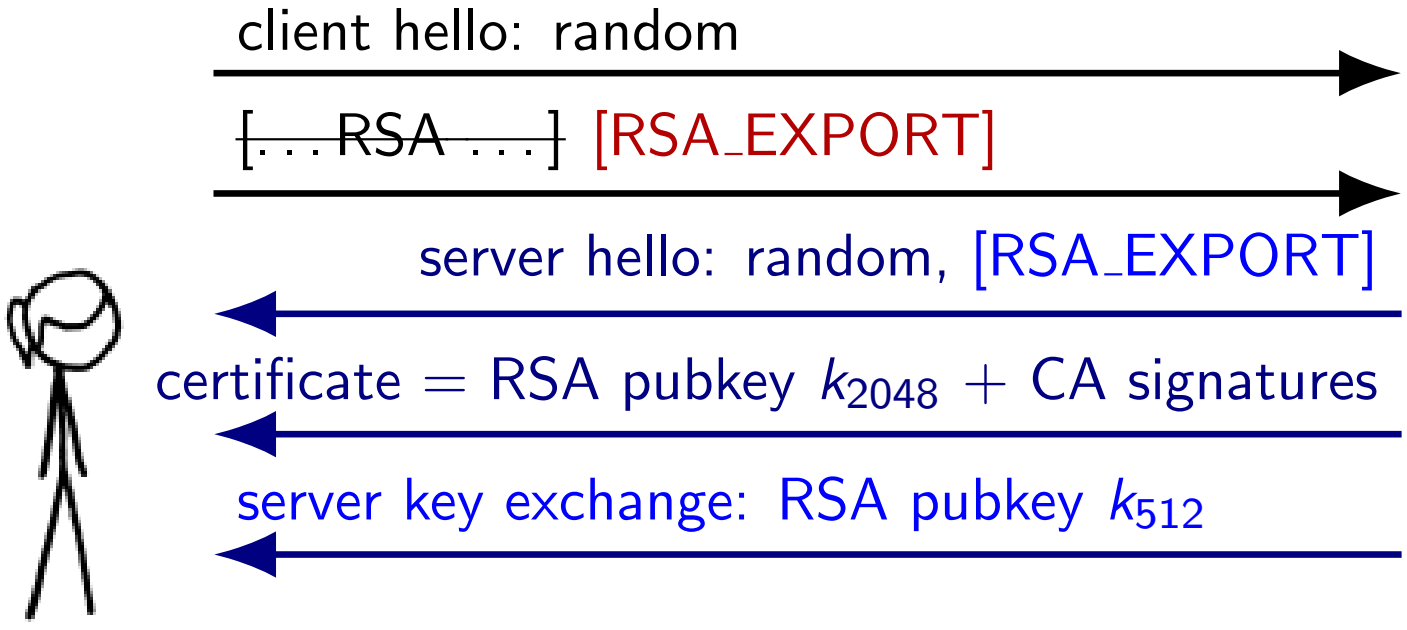
Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

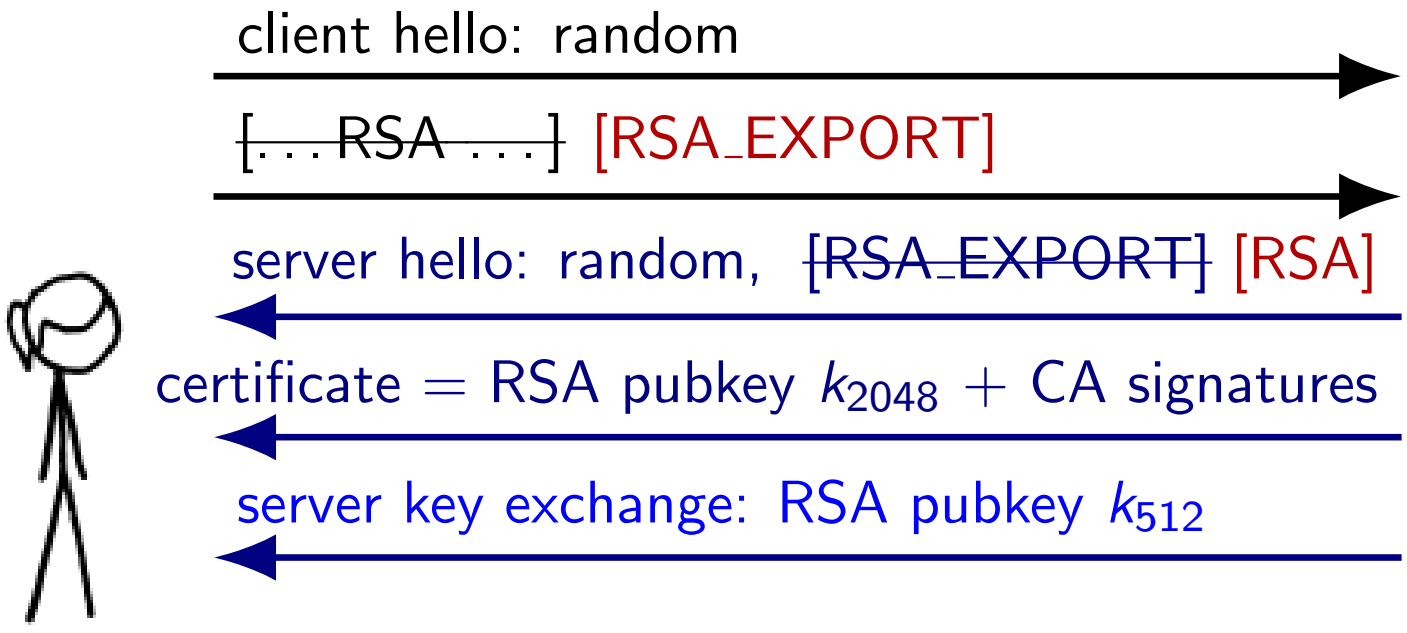
FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]



FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]



FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[...RSA...]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[...RSA...]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{m_c}}(\mathrm{dialog})$

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{mc}}$(dialog)

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random →

~~[. . . RSA . . .]~~ [RSA_EXPORT] →

server hello: random, ~~[RSA_EXPORT]~~ [RSA] ←

certificate = RSA pubkey k_{2048} + CA signatures ←

server key exchange: RSA pubkey k_{512} ←

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$ →

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog) →

server finished: $\mathrm{Auth}_{k_{ms}}$(modified dialog) ←

KDF(pms, randoms) → k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) → k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[...RSA...]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{ms}}$(modified dialog)

Enc_{k_e}(request)

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK vulnerability in practice

- Implementation flaw affected OpenSSL, Microsoft SChannel, IBM JSSE, Safari, Android, Chrome, BlackBerry, Opera, IE

FREAK vulnerability in practice

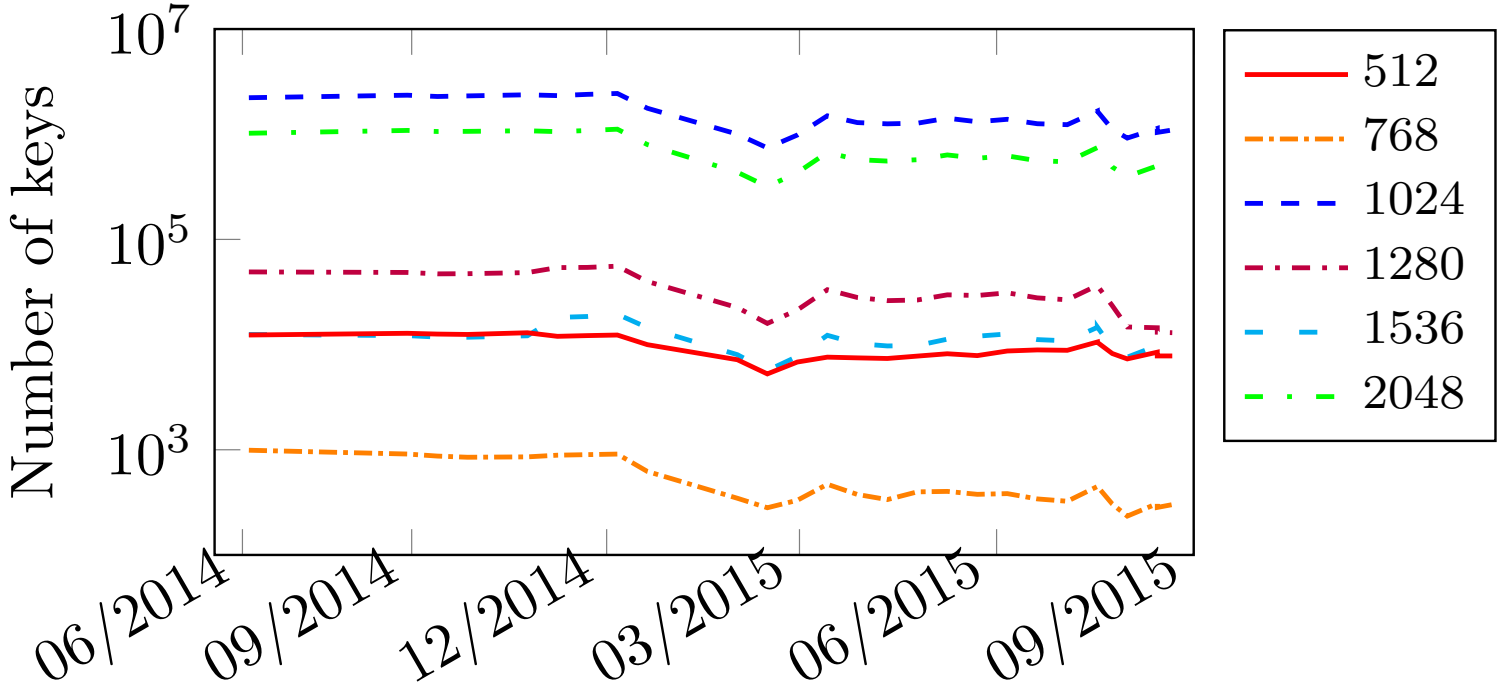
- Implementation flaw affected OpenSSL, Microsoft SChannel, IBM JSSE, Safari, Android, Chrome, BlackBerry, Opera, IE
- Attack outline:
  1. MITM attacker downgrades connection to export, learns server's ephemeral 512-bit RSA export key.
  2. Attacker factors 512-bit modulus to obtain server private key.
  3. Attacker uses private key to forge client/server authentication for successful downgrade.

FREAK vulnerability in practice

- Implementation flaw affected OpenSSL, Microsoft SChannel, IBM JSSE, Safari, Android, Chrome, BlackBerry, Opera, IE
- Attack outline:
  1. MITM attacker downgrades connection to export, learns server's ephemeral 512-bit RSA export key.
  2. Attacker factors 512-bit modulus to obtain server private key.
  3. Attacker uses private key to forge client/server authentication for successful downgrade.
- Attacker challenge: Need to know 512-bit private key before connection times out
- Implementation shortcut: "Ephemeral" 512-bit RSA server keys generated only on application start; last for hours, days, weeks, months.

DNSSEC: Domain Name System Security Extensions

[Rapid7 + SURFnet datasets + custom scans]



RFC 6781 [2012]

"it is estimated that most zones can safely use 1024-bit keys for at least the next ten years."

DKIM: Domain-Keys Identified Mail

[Rapid7 + SURFNET + custom scans]

Public Keys

512 bits	103 (0.9%)
384 bits	20 (0.2%)
128 bits	1 (0.0%)
Parse error	591 (5.1%)
Total	11,637

DKIM: Domain-Keys Identified Mail

[Rapid7 + SURFNET + custom scans]

Public Keys

512 bits	103 (0.9%)
384 bits	20 (0.2%)
128 bits	1 (0.0%)
Parse error	591 (5.1%)
Total	11,637

128-bit key

```
[REDACTED] bdb6389e41d8df6141acdda91a7c23c1
```

DKIM: Domain-Keys Identified Mail

[Rapid7 + SURFNET + custom scans]

Public Keys

512 bits	103 (0.9%)
384 bits	20 (0.2%)
128 bits	1 (0.0%)
Parse error	591 (5.1%)
Total	11,637

128-bit key

```
[REDACTED] bdb6389e41d8df6141acdda91a7c23c1
```

```
sage: time factor(Integer("bdb6389e41d8df6141acdda91a7c23c1",16))
CPU times: user 68.3 ms, sys: 17.3 ms, total: 85.6 ms
Wall time: 132 ms
14060786408729026139 * 17934291173672884499
```

Summary of RSA best practices

- Use elliptic curve cryptography.

If that's not an option:

- Choose RSA modulus N at least 2048 bits.
- Use a good random number generator to generate primes.
- Use a secure, randomized padding scheme.