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

\[ N = pq \text{ modulus} \]
\[ e \text{ encryption exponent} \]

Private Key

\[ p, q \text{ primes} \]
\[ d \text{ decryption exponent} \]
\[ (d = e^{-1} \mod (p-1)(q-1)) \]

Encryption

public key = \((N, e)\)

ciphertext = message^e \mod N

message = ciphertext^d \mod N
Textbook RSA
[Rivest Shamir Adleman 1977]

Public Key

\[ N = pq \text{ modulus} \]
\[ e \text{ encryption exponent} \]

Private Key

\[ p, q \text{ primes} \]
\[ d \text{ decryption exponent} \]
\[ d = e^{-1} \mod (p - 1)(q - 1) \]

Signing

\[ \text{public key} = (N, e) \]
\[ \text{signature} = \text{message}^d \mod N \]
\[ \text{message} = \text{signature}^e \mod N \]
Computational problems

Factoring

**Problem:** Given \( N \), compute its prime factors.

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

**eth roots mod \( N \)**

**Problem:** Given \( N, e, \) and \( c, \) compute \( x \) such that \( x^e \equiv c \mod N. \)

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

**Attack: Chosen ciphertext attack**
Given a ciphertext \( c = \text{Enc}(m) \) for unknown \( m \), attacker asks for \( \text{Dec}(ca^e \mod N) = d \) and computes \( m = da^{-1} \mod N \).

**Attack: Signature forgery**
Attacker wants \( \text{Sign}(x) \). Attacker computes \( z = xy^e \mod N \) for some \( y \) and asks signer for \( s = \text{Sign}(z) = z^d \mod N \). Attacker computes \( \text{Sign}(z) = sy^{-1} \mod N \).

So in practice **always use padding on messages**.
A Crypto Nerd's Imagination:

His laptop's encrypted. Let's build a million-dollar cluster to crack it.

No good! It's 4096-bit RSA!

Blast! Our evil plan is foiled!

What Would Actually Happen:

His laptop's encrypted. Drug him and hit him with this $5 wrench until he tells us the password.

GOT IT.
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 based on Python, but there are a few differences:

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

^ is exponentiation, not xor
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:

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

^ is exponentiation, not xor

It has lots of useful libraries:

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

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

sage: 2^3
8

^ is exponentiation, not xor

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

**Decryption:**

```sage
sage: Integer(pow(c, d, N)).str(base=35)
'helloworld'
```
So how hard *is* factoring?
So how hard *is* factoring?

```python
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
```

```python
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
```

```python
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
```

```python
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
71044139867382099583965064084826540441 * 95091214714150393464646264945135836937
```
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))
```

```
    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
71044139867382099583965064084826540441 * 95091214714150393464646264945135836937
```
So how hard *is* factoring?

```python
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
```

```python
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
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
71044139867382099583965064084826540441 * 95091214714150393464646
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 mod $N$.
- 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 mod $N$.
▷ 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: $f(x) = x^2 + c \mod N$ 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

```python
def rho(n):
    a = 98357389475943875; 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 = 698599699288686665490308069057420138223871
sage: rho(N)
2053
```
Reminders: Orders and groups

Theorem (Fermat’s Little Theorem)
\[a^{p-1} \equiv 1 \pmod{p} \text{ for any } 0 < a < p.\]

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

Theorem (Lagrange)
\(\text{ord}(a)_p\) divides \(p - 1\).
Pollard’s $p - 1$ method

Good for finding special small factors

Intuition

- If $a^r \equiv 1 \mod p$ then $\text{ord}(a)_p \mid r$ and $p \mid \gcd(a^r - 1, N)$.

- Don’t know $p$, pick very smooth number $r$, hoping for $\text{ord}(a)_p$ to divide it.

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

```
N=44426601460658291157725536008128017297890787
4637194279031281180366057
r=lcm(range(1,2^22)) # this takes a while ...
s=Integer(pow(2,r,N))
sage: gcd(s-1,N)
1267650600228229401496703217601
```
Pollard’s \( p - 1 \) method

Good for finding special small factors

**Intuition**

- If \( a^r \equiv 1 \mod p \) then \( \text{ord}(a)_p \mid r \) and \( p \mid \gcd(a^r - 1, N) \).

- Don’t know \( p \), pick very smooth number \( r \), hoping for \( \text{ord}(a)_p \) to divide it.

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

\[
N=44426601460658291157725536008128017297890787
4637194279031281180366057
\]

\[
r=\text{lcm}(\text{range}(1,2^{22})) \quad \# \text{this takes a while} \quad ...
\]

\[
s=\text{Integer}(\text{pow}(2,r,N))
\]

\[
sage: \gcd(s-1,N)
1267650600228229401496703217601\]
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 $N$. 

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.
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.
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__(self, 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 $L_p(1/2, \sqrt{2}) = O(e^{\sqrt{2}\sqrt{\ln p}\ln\ln p})$. 
Main insight: If we can find two squares $a^2$ and $b^2$ such that

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

Then

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

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

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

Then

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

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 $c^2 - N$, $(c + 1)^2 - N$, ...
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 $\prod d_i$ 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 $N = 2759$. 

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

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

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

An example of the quadratic sieve

Let’s try to factor $N = 2759$.

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

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

**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** $\gcd\{2759, 53 \cdot 57 \cdot 58 - \sqrt{50 \cdot 490 \cdot 605}\} = 31$. 
An example of the quadratic sieve

Let’s try to factor $N = 2759$.

**Sieving** values $\left(\lceil \sqrt{N} + i \rceil \right)^2 \mod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$. 
An example of the quadratic sieve

Let’s try to factor $N = 2759$.

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

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

$54^2 - 2759 = 157$. 

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

QS computes $\gcd\{2759, 53 \cdot 57 \cdot 58 \}$ = 31.
An example of the quadratic sieve

Let’s try to factor $N = 2759$.

**Sieving** values $\left(\lceil \sqrt{N} + i \rceil \right)^2 \mod N$:

$53^2 - 2759 = 50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$.
$55^2 - 2759 = 266$. 
An example of the quadratic sieve

Let’s try to factor $N = 2759$.

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

$$53^2 - 2759 = 50 = 2 \cdot 5^2.$$  
$$54^2 - 2759 = 157.$$  
$$55^2 - 2759 = 266.$$  
$$56^2 - 2759 = 377.$$
An example of the quadratic sieve

Let’s try to factor $N = 2759$.

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

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

$54^2 - 2759 = 157$.

$55^2 - 2759 = 266$.

$56^2 - 2759 = 377$.

$57^2 - 2759 = 490 = 2 \cdot 5 \cdot 7^2$. 

Recall idea: If $a^2 - N$ is a square $b^2$ then $N = (a - b)(a + b)$.
An example of the quadratic sieve

Let’s try to factor $N = 2759$.

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

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

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

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

An example of the quadratic sieve

Let’s try to factor $N = 2759$.

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

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

**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 $(\lceil \sqrt{N} + i \rceil)^2 \mod N$:

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

**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 \( a^2 = b^2 \mod N \) 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

\[ L_N\left(\frac{1}{3}, \sqrt[3]{\frac{64}{9}}\right) = e^{(1.923 + o(1)) (\ln N)^{1/3} (\ln \ln N)^{2/3}} \]
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: ≈ 1,000,000 core-years
2048-bit RSA: Minimum recommended key size today.
How long does factoring take with the number field sieve?

![Diagram of the number field sieve process]

**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
It works!

This is the default web page for this server.

The web server software is running but no content has been added, yet.
*.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**: *.uh.cu
- **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)
- **Parameters**: none
- **Public Key**: 256 bytes: 91 8F 9E D6 AF A6 DB AC ...
TLS RSA Key Exchange

client hello: client random

[...RSA ...]
TLS RSA Key Exchange

client hello: client random

[...RSA ...]

server hello: server random, [RSA]

certificate = \text{RSA pubkey } k_{2048} + \text{CA signatures}

client key exchange: RSAenc\(k_{2048}(\text{pms})\)

KDF\(\text{pms}, \text{randoms} \rightarrow k_m c_k, k_m s_k, k_e\)

KDF\(\text{pms}, \text{randoms} \rightarrow k_m c_k, k_m s_k, k_e\)

client finished: Auth\(k_m c_k(\text{dialog})\)

server finished: Auth\(k_m s_k(\text{dialog})\)

Enc\(k_e(\text{request})\)
TLS RSA Key Exchange

client hello: client random

[...RSA ...]

server hello: server random, [RSA]

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

client key exchange: $\text{RSAenc}_{k_{2048}}(pms)$

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

KDF($pms$, randoms) $\rightarrow$ $km_c, km_s, ke$

KDF($pms$, randoms) $\rightarrow$ $km_c, km_s, ke$
TLS RSA Key Exchange

client hello: client random

[...RSA ...]

server hello: server random, [RSA]

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

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

KDF(pms, randoms) → $k_m$, $k_s$, $k_e$

client finished: Auth_{km_c}(dialog)

server finished: Auth_{km_s}(dialog)

KDF(pms, randoms) → $k_m$, $k_s$, $k_e$
TLS RSA Key Exchange

client hello: client random

[...RSA ...]

server hello: server random, [RSA]

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

client key exchange: RSA_{\text{enc}}_{k_{2048}}(pms)

client finished: Auth_{k_m} (dialog)

KDF(pms, randoms) $\rightarrow$ $k_m, k_s, k_e$

server finished: Auth_{k_s} (dialog)

Enc_{k_e}(request)

KDF(pms, randoms) $\rightarrow$ $k_m, k_s, k_e$
Does anyone use 512-bit RSA?
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.
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 $\mathbb{Z}/p\mathbb{Z}$); 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 $k_{512}$

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

KDF($pms$, randoms) $\rightarrow k_{mc}, k_{ms}, k_{ec}$

KDF($pms$, randoms) $\rightarrow k_{mc}, k_{ms}, k_{ec}$

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

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

Enc $k_{ec}(request)$
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: $\text{RSAEnc}_{k_{512}}(pms)$

KDF($pms$, randoms) $\rightarrow$ $k_m, k_s, k_e$

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

KDF($pms$, randoms) $\rightarrow$ $k_m, k_s, k_e$
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) → k_{mc}, k_{ms}, k_e

client finished: Auth_{k_{mc}}(dialog)

KDF(pms, randoms) → k_{mc}, k_{ms}, k_e

server finished: Auth_{k_{ms}}(dialog)
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: $\text{RSAenc}_{k_{512}}(\text{pms})$

KDF(\text{pms}, \text{randoms}) \rightarrow k_{mc}, k_{ms}, k_e

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

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

\text{Enc}_{k_e}(\text{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]
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]

\[
\text{client hello: random} \\
[\ldots \text{RSA} \ldots] \quad [\text{RSA\_EXPORT}] \\
\text{server hello: random, } [\text{RSA\_EXPORT}] \quad [\text{RSA}] \\
\text{certificate } = \text{ RSA pubkey } k_{2048} + \text{ CA signatures} \\
\text{server key exchange: RSA pubkey } k_{512} \\
\]
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: RSA$_{enc}k_{512}(pms)$

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. [BDFKPSZZZ 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: RSAenc$_{k_{512}}$(pms)

KDF(pms, randoms) $\rightarrow$ $k_{mc}, k_{ms}, k_e$

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

KDF(pms, randoms) $\rightarrow$ $k_{mc}, k_{ms}, k_e$
```
FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZZ 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: RSAenc$_{k_{512}}$(pms)

KDF(pms, randoms) → $k_{mc}, k_{ms}, k_{e}$

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

KDF(pms, randoms) → $k_{mc}, k_{ms}, k_{e}$
FREAK: MITM downgrade attack to export RSA

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

\[
\begin{align*}
\text{client hello: random} & \quad \rightarrow \\
[\ldots\text{RSA}\ldots] \quad [\text{RSA\_EXPORT}] & \\
\text{server hello: random, [RSA\_EXPORT] [RSA]} & \\
\text{certificate = RSA pubkey } k_{2048} + \text{CA signatures} & \\
\text{server key exchange: RSA pubkey } k_{512} & \\
\text{client key exchange: RSAEnc}_{k_{512}}(pms) & \\
\text{client finished: } \text{Auth}_{k_m} (\text{modified dialog}) & \\
\text{server finished: } \text{Auth}_{k_m} (\text{dialog}) & \\
\end{align*}
\]

KDF(pms, randoms) \rightarrow k_m, k_m, k_e
FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accepted unexpected server key exchange messages. [BDFKPSZZZ 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: RSAenc$_{k_{512}}(pms)$

KDF($pms$, randoms) $\rightarrow$ $k_m, k_s, k_e$

client finished: Auth$_{k_m}$ (modified dialog)

server finished: Auth$_{k_s}$ (modified dialog)
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: RSAenc$_{k_{512}}$(pms)

KDF(pms, randoms) $\rightarrow$ $k_{mc}, k_{ms}, k_e$

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

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

Enc$_{k_e}$(request)
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.
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.
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

<table>
<thead>
<tr>
<th>Key Length</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 bits</td>
<td>103</td>
<td>0.9%</td>
</tr>
<tr>
<td>384 bits</td>
<td>20</td>
<td>0.2%</td>
</tr>
<tr>
<td>128 bits</td>
<td>1</td>
<td>0.0%</td>
</tr>
<tr>
<td>Parse error</td>
<td>591</td>
<td>5.1%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>11,637</strong></td>
<td><strong>11,637</strong></td>
</tr>
</tbody>
</table>

```python
db = 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
```
DKIM: Domain-Keys Identified Mail
[Rapid7 + SURFNET + custom scans]

Public Keys

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 bits</td>
<td>103</td>
<td>0.9%</td>
</tr>
<tr>
<td>384 bits</td>
<td>20</td>
<td>0.2%</td>
</tr>
<tr>
<td>128 bits</td>
<td>1</td>
<td>0.0%</td>
</tr>
<tr>
<td>Parse error</td>
<td>591</td>
<td>5.1%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>11,637</strong></td>
<td></td>
</tr>
</tbody>
</table>

128-bit key

[REDACTED] bdb6389e41d8df6141acdda91a7c23c1
DKIM: Domain-Keys Identified Mail
[Rapid7 + SURFNET + custom scans]

Public Keys

<table>
<thead>
<tr>
<th>Bit Size</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 bits</td>
<td>103</td>
<td>0.9%</td>
</tr>
<tr>
<td>384 bits</td>
<td>20</td>
<td>0.2%</td>
</tr>
<tr>
<td>128 bits</td>
<td>1</td>
<td>0.0%</td>
</tr>
<tr>
<td>Parse error</td>
<td>591</td>
<td>5.1%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>11,637</strong></td>
<td></td>
</tr>
</tbody>
</table>

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.