The state of factoring algorithms and other cryptanalytic threats to RSA

Daniel J. Bernstein
University of Illinois at Chicago
Technische Universiteit Eindhoven

Nadia Heninger
Microsoft Research New England

Tanja Lange
Technische Universiteit Eindhoven

Textbook RSA

Public Key

$N = pq$ modulus

e encryption exponent

Private Key

p, q primes

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

Encrypt

$$c = m^e \pmod{N}$$

Decrypt

$$m = c^d \pmod{N}$$

Verify

$$m = s^e \pmod{N}$$

Sign

$$s = m^d \pmod{N}$$

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.
- Conflicting results about 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.

Practical concern #1: Textbook RSA is insecure

RSA encryption is homomorphic under multiplication. This lets an attacker do all sorts of fun things:

Attack: Malleability

Given a ciphertext $c = m^e \bmod N$, $ca^e \bmod N$ is an encryption of ma for any a chosen by attacker.

Attack: Chosen ciphertext attack

Given a ciphertext c, attacker asks for decryption of $ca^e \bmod N$ and divides by a to obtain m.

Attack: Signature forgery

Attacker convinces a signer to sign $z = xy^e \bmod N$ and computes a valid signature of x as $z^d/y \bmod N$.

So in practice we always use padding on messages.

Practical concern #2: Efficiency

Choose e to be small and low hamming weight.

Use Chinese Remainder Theorem to speed up computations with d:

$$d_p = d \bmod p-1 \qquad d_q = d \bmod q-1$$

Compute

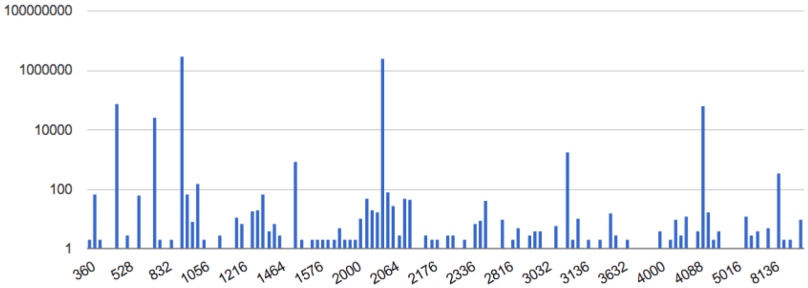
$$c_p = m^{d_p} \bmod p \qquad c_q = m^{d_q} \bmod q$$

$$c = \mathrm{crt}(c_p, c_q) \bmod N$$

Public-key cipher usage

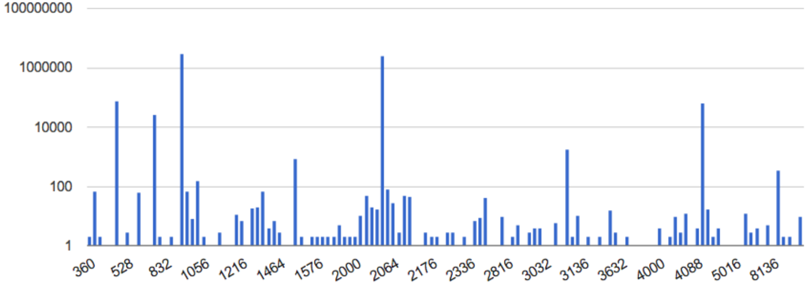
	RSA	DSA	ECDSA	ElGamal	GOST
TLS	5,756,445	6,241	8		225
SSH	3,821,651	3,729,010	153,109		7
PGP	676,590	2,119,245		2,126,098	

RSA key size distribution



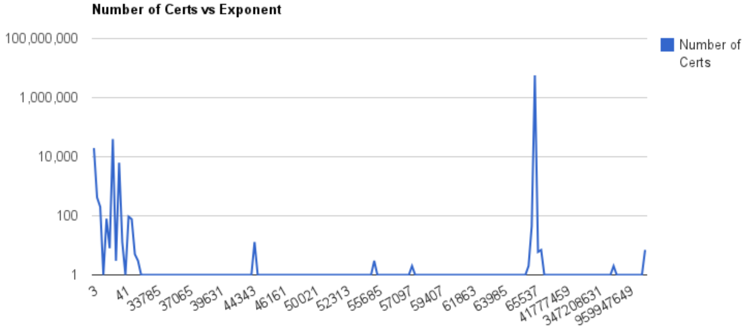
TLS, November 2011

RSA key size distribution



TLS, November 2011

RSA exponent distribution



TLS, November 2011

Implementation issues

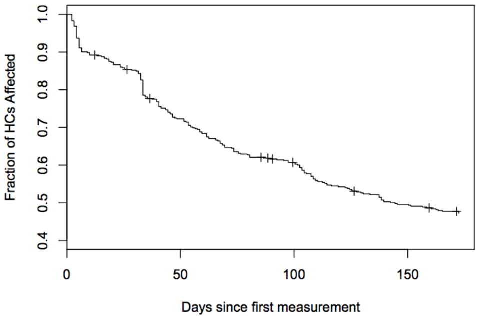
```
MD_Update(&m,buf,j);
```

The Debian OpenSSL entropy disaster

August, 2008: Discovered by Luciano Bello

Keys dependent only on pid and machine architecture:
294,912 keys per key size.

"When Private Keys are Public: Results from the 2008 Debian OpenSSL Vulnerability" [Yilek, Rescorla, Shacham, Enright, Savage 2009]



Searching for more entropy problems

Experiment

1. Acquire many public keys.
2. Look for obvious key-generation problems.

“Public keys” [Lenstra, Hughes, Augier, Bos, Kleinjung, Wachter Crypto 2012]

“Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices” [Heninger, Durumeric, Wustrow, Halderman Usenix Security 2012]

What could go wrong with RSA and entropy problems?

- Two hosts share N: $\rightarrow$ both know private key of the other.
- Two hosts share RSA moduli with a prime factor in common $\rightarrow$ outside observer can factor both keys by calculating the GCD of public moduli.

$$N_1 = pq_1 \qquad N_2 = pq_2$$

$$\gcd(N_1, N_2) = p$$

Time to factor 768-bit RSA modulus:
two years

Time to calculate GCD for 1024-bit RSA moduli:
15μs

Looking for problems: RSA common divisors

Speed-bump

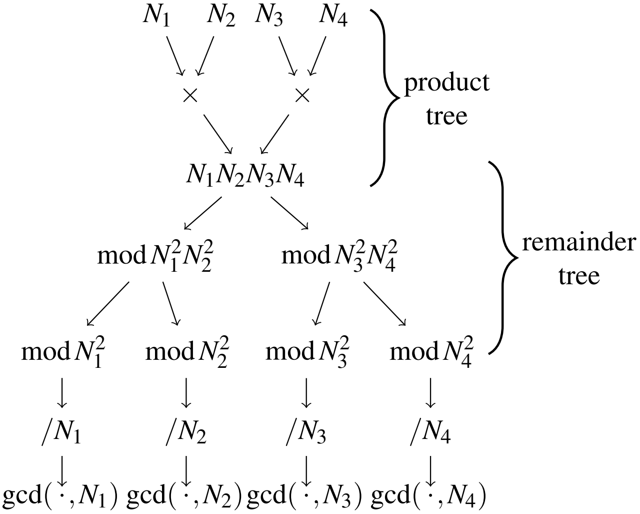
Computing pairwise $\gcd(N_i, N_j)$ for our dataset would take

$$15\mu s \times \binom{11 \times 10^6}{2} \text{pairs} \approx 30 \text{ years}$$

of computation time.

Efficient all-pairs GCDs

We implemented an efficient algorithm due to [Bernstein 2004].



Results

Repeated Keys

- $>$ 60% of TLS and SSH hosts have non-unique keys.
- $>$ 5% of TLS hosts and $>$ 10% of SSH hosts serve default or low-entropy keys
- 0.03% TLS hosts and 0.5% of SSH hosts serve Debian weak keys

Factored keys

- 0.5% of TLS hosts and 0.03% of SSH hosts keys factored

Flaw Found in an Online Encryption Method

By JOHN MARKOFF
Published: February 14, 2012

SAN FRANCISCO — A team of European and American mathematicians and cryptographers have discovered an unexpected weakness in the encryption system widely used worldwide for online shopping, banking, e-mail and other Internet services intended to remain private and secure.

The flaw — which involves a small but measurable number of cases — has to do with the way the system generates random numbers, which are used to make it practically impossible for an attacker to unscramble digital messages. While it can affect the transactions of individual Internet users, there is nothing an individual can do about it. The operators of large Web sites will need to make changes to ensure the security of their systems, the researchers said.

The potential danger of the flaw is that even though the number of users affected by the flaw may be small, confidence in the security of Web transactions is reduced, the authors said.

The system requires that a user first create and publish the product of two large prime numbers, in addition to another number, to generate a public "key." The original numbers are kept secret. To encrypt a message, a second person employs a formula that contains the public number. In

Readers' Comments

Readers shared their thoughts on this article.

Read All Comments (127) »

RECOMMEND
TWITTER
LINKEDIN
COMMENTS (127)
SIGN IN TO E-MAIL
PRINT
REPRINTS
SHARE

Log in to see what your friends are sharing on nytimes.com. Privacy Policy | What's This?

Log In With Facebook

What's Popular Now

Why I Am Leaving Goldman Sachs

The Benefits of Bilingualism

Get the TimesLimited E-Mail

Sign Up

Attributing vulnerabilities to implementations

Vast majority of compromised keys generated by headless or embedded network devices.

- Used information in certificate subjects, version strings, served over https or http, etc. to cluster hosts by implementation.
- Routers, firewalls, switches, server management cards, cable modems, VOIP devices, printers, projectors...

Vulnerabilities due mainly to generating keys on first boot with `/dev/urandom`, complicated interaction with application entropy pool behavior.

Disclosure and remediation

- Contacted 61 manufacturers of vulnerable products.
- After 9 months 13 of them have told us they fixed problem.
- 5 released security advisories.

This is just the tip of the iceberg

More examples of bad randomness!

This is just the tip of the iceberg

More examples of bad randomness!

- PGP database. [Lenstra et al.]
  2 factored RSA keys out of 700,000. Why?

This is just the tip of the iceberg

More examples of bad randomness!

- PGP database. [Lenstra et al.]
  2 factored RSA keys out of 700,000. Why?
- Smartcards. [2012 Chou (slides in Chinese)]

  Factored 103 Taiwan Citizen Digital Certificates
  (out of 2.26 million):
  smartcard certificates used for paying taxes etc.

  Names, email addresses, national IDs were public
  but **103 private keys** are now known.

This is just the tip of the iceberg

More examples of bad randomness!

- PGP database. [Lenstra et al.]
  2 factored RSA keys out of 700,000. Why?

- Smartcards. [2012 Chou (slides in Chinese)]

  Factored 103 Taiwan Citizen Digital Certificates
  (out of 2.26 million):
  smartcard certificates used for paying taxes etc.

  Names, email addresses, national IDs were public
  but **103 private keys** are now known.

  Smartcard manufacturer:
  "Giesecke & Devrient: Creating Confidence."

Evaluating RSA's risk

Factoring keys is bad, but DSA (and ECDSA) are **worse** if you're worried about entropy problems.

Bad entropy from a single signature can compromise private key.

- e.g. A perfectly good DSA key used on a 2008 Debian system $\rightarrow$ compromised.
- e.g. 1% of DSA SSH host keys compromised from signatures with bad randomness after two scans.

Would be easy to fix in standard. (Make nonce deterministic: hash of message, secret salt.)

Side-channel attacks

Timing attacks

- **Hardware** [Kocher 96] "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems."
- **Remote software** [Brumley Boneh 05] "Remote timing attacks are practical."

Cache timing

- **Inter-process software** [Percival 05] "Cache missing for fun and profit."
- **Cross-VM software** [Zhang Juels Reiter Ristenpart 12] "Cross-VM Side Channels and Their Use to Extract Private Keys"

Faults

- [Boneh, DeMillo, Lipton 96], [Lenstra 96]

Side-channel attacks

Side-channel structures relevant to RSA:

Exponentiation

- Square-and-multiply: different execution paths/instruction timing/power levels dependent on bits of private key.
- **Defense:** Exponent blinding, square and always multiply, never branch.

CRT coefficients

- Fault attacks can produce a value valid mod only one prime.
- **Defense:** Verify output.

Padding oracles

- Implementations differentiating between correct and incorrect decryption $\rightarrow$ chosen-ciphertext attacks.
- **Defense:** Don't distinguish failures.

Partial key recovery and related attacks

RSA particularly susceptible to partial key recovery attacks.

Theorem (Coppersmith/Howgrave-Graham)

We can find roots x of polynomials f of degree d mod divisors B of N, $B = N^{\beta}$, when $|x| \le N^{\beta^2/d}$.

(Note that RSA problem is to find roots of $x^e - c \bmod N$.)

Partial key recovery and related attacks

RSA particularly susceptible to partial key recovery attacks.

Theorem (Coppersmith/Howgrave-Graham)

We can find roots x of polynomials f of degree d mod divisors B of N, $B = N^\beta$, when $|x| \leq N^{\beta^2/d}$.

(Note that RSA problem is to find roots of $x^e - c \bmod N$.)

- Can factor given $1/2$ bits of p. [Coppersmith 96]
- Can factor given $1/4$ bits of d. [Boneh Durfee Frankel 98]
- Can factor given $1/2$ bits of d_p. [Blömer May 03]

Also implies constraints on key choice:

- Can factor if $d < N^{0.292}$ [Boneh Durfee 98]

Partial key recovery and related attacks

RSA particularly susceptible to partial key recovery attacks.

Theorem (Coppersmith/Howgrave-Graham)

We can find roots x of polynomials f of degree d mod divisors B of N, $B = N^{\beta}$, when $|x| \leq N^{\beta^2/d}$.

(Note that RSA problem is to find roots of $x^e - c \bmod N$.)

- Can factor given $1/2$ bits of p. [Coppersmith 96]
- Can factor given $1/4$ bits of d. [Boneh Durfee Frankel 98]
- Can factor given $1/2$ bits of d_p. [Blömer May 03]

Also implies constraints on key choice:

- Can factor if $d < N^{0.292}$ [Boneh Durfee 98]

Message security: Least significant bit of message as secure as entire message. [Alexi Chor Goldreich Schnorr 88]

Protocol issues.

Padding schemes: Simple cryptanalyses

Fixed-pattern padding

Define a padding scheme $(P|m)$.
Coppersmith's theorem: With $e = 3$, if $|m| < N^{1/3}$ then can efficiently compute m as solution to

$$c - (P \cdot 2^t + x)^3 \bmod N$$

[Brier Clavier Coron Naccache 01] Existential forgery of signatures with $|m| > N^{1/3}$ by finding solutions to relation

$$(P + m_1)(P + m_2) = (P + m_3)(P + m_4) \bmod N$$

using continued fractions.

The agony and ecstasy of PKCS#1v1.5 and OAEP

PKCS#1: ($0x00\,0x02$|padding string|$0x00$|message)

The agony and ecstasy of PKCS#1v1.5 and OAEP

PKCS#1: $(0x00\,0x02|\text{padding string}|0x00|\text{message})$

Cryptographers: PKCS#1 is not IND-CCA2 secure!

The agony and ecstasy of PKCS#1v1.5 and OAEP

PKCS#1: $(0x00\,0x02|\text{padding string}|0x00|\text{message})$

Cryptographers: PKCS#1 is not IND-CCA2 secure!

Practitioners: That is not relevant in practice.

The agony and ecstasy of PKCS#1v1.5 and OAEP

PKCS#1: $(0x00\,0x02|\text{padding string}|0x00|\text{message})$

Cryptographers: PKCS#1 is not IND-CCA2 secure!

Practitioners: That is not relevant in practice.

1994 **Bellare Rogaway:** Use OAEP, it's provably secure in random oracle model.

The agony and ecstasy of PKCS#1v1.5 and OAEP

PKCS#1: ($0x00\,0x02|$padding string$|0x00|$message)

Cryptographers: PKCS#1 is not IND-CCA2 secure!

Practitioners: That is not relevant in practice.

1994 **Bellare Rogaway:** Use OAEP, it's provably secure in random oracle model.

1996 **Bleichenbacher:** "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1"

The agony and ecstasy of PKCS#1v1.5 and OAEP

PKCS#1: ($0x00\,0x02$|padding string|$0x00$|message)

Cryptographers: PKCS#1 is not IND-CCA2 secure!

Practitioners: That is not relevant in practice.

1994 **Bellare Rogaway:** Use OAEP, it's provably secure in random oracle model.

1996 **Bleichenbacher:** "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1"

1998 **RFC 2437:** (1998) "`RSAES-OAEP is recommended for new applications; RSAES-PKCS1-v1_5 is included only for compatibility with existing applications, and is not recommended for new applications`"

The agony and ecstasy of PKCS#1v1.5 and OAEP

PKCS#1: ($0x00\,0x02$|padding string|$0x00$|message)

Cryptographers: PKCS#1 is not IND-CCA2 secure!

Practitioners: That is not relevant in practice.

1994 **Bellare Rogaway:** Use OAEP, it's provably secure in random oracle model.

1996 **Bleichenbacher:** "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1"

1998 **RFC 2437:** (1998) "`RSAES-OAEP is recommended for new applications; RSAES-PKCS1-v1_5 is included only for compatibility with existing applications, and is not recommended for new applications`"

The agony and ecstasy of PKCS#1v1.5 and OAEP

2001 **Shoup:** There's a hole in the OAEP security proof, but I fixed it. The proof uses Coppersmith's theorem.

The agony and ecstasy of PKCS#1v1.5 and OAEP

2001 **Shoup:** There's a hole in the OAEP security proof, but I fixed it. The proof uses Coppersmith's theorem.

2008 **RFC5246:** "`for maximal compatibility with earlier versions of TLS, this specification uses the RSAES-PKCS1-v1_5 scheme`"

The agony and ecstasy of PKCS#1v1.5 and OAEP

2001 **Shoup:** There's a hole in the OAEP security proof, but I fixed it. The proof uses Coppersmith's theorem.

2008 **RFC5246:** "`for maximal compatibility with earlier versions of TLS, this specification uses the RSAES-PKCS1-v1_5 scheme`"

2012 **Bardou Focardi Kawamoto Simionato Steel Tsay:** Bleichenbacher attack works against RSA SecureID tokens, Estonian ID cards.

Shoup's "Simple RSA"

$$C_0 = r^e \bmod N \qquad r \text{ random}$$

$$k_0 || k_1 = H(r) \qquad H \text{ hash function}$$

$$C_1 = \mathrm{enc}_{k_0}(m) \qquad \mathrm{enc} \text{ a symmetric cipher}$$

$$T = \mathrm{mac}_{k_1}(C_1)$$

Output (C_0, C_1, T).
Very short and efficient security proof.

Factoring, aka. breaking RSA if nothing else went wrong.

Preliminaries: Using Sage

The following 2 parts use some code snippets to give examples using the free open source mathematics software Sage. http://www.sagemath.org/.

Sage looks like Python

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

Preliminaries: Using Sage

The following 2 parts use some code snippets to give examples using the free open source mathematics software Sage. http://www.sagemath.org/.

Sage looks like Python, but there are a few differences:

^ is exponentiation, not xor

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

Preliminaries: Using Sage

The following 2 parts use some code snippets to give examples using the free open source mathematics software Sage. `http://www.sagemath.org/`.

Sage looks like 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

The following 2 parts use some code snippets to give examples using the free open source mathematics software Sage. `http://www.sagemath.org/`.

Sage looks like 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
```

That's it, just `factor(N)`

Preliminaries: Using Sage

The following 2 parts use some code snippets to give examples using the free open source mathematics software Sage. http://www.sagemath.org/.

Sage looks like 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)
```

Trial division

Factoring easy-to-factor numbers:

Trial division

Factoring easy-to-factor numbers:

```
sage: N=1701411834604692317316873037158841057535
```

Trial division

Factoring easy-to-factor numbers:

```
sage: N=1701411834604692317316873037158841057535
```

is obviously divisible by 5.

```
sage: N/5        # / is exact division
340282366920938463463374607431768211507
```

Searching for p by trial division takes time about $p/\log(p)$ (number of primes up to p) trial divisions.
Computers can test quickly for divisibility by a precomputed set of primes (using % or gcd with product). Can batch this computation for many moduli N using product and remainder trees.

Pollard rho

Do random walk modulo N, hope for collision modulo factor p.
E.g. using Floyd's cycle finding algorithm

```
N=698599699288686665490308069057420138223871
a=98357389475943875; c=10 # some random values
a1=(a^2+c) % N ; a2=(a1^2+c) % N
while gcd(N,a2-a1)==1:
    a1=(a1^2+c) %N
    a2=(((a2^2+c)%N)^2+c)%N
gcd(N,a2-a1)
```

Pollard rho

Do random walk modulo N, hope for collision modulo factor p.
E.g. using Floyd's cycle finding algorithm

```
N=698599699288686665490308069057420138223871
a=98357389475943875; c=10 # some random values
a1=(a^2+c) % N ; a2=(a1^2+c) % N
while gcd(N,a2-a1)==1:
    a1=(a1^2+c) %N
    a2=(((a2^2+c)%N)^2+c)%N
gcd(N,a2-a1) # output is 2053
```

Pollard's rho method runs till a prime p divides $a1 - a2$ and N.
By the birthday paradox expect collisions modulo p after $\sqrt{p}$ steps.

Pollard rho

Do random walk modulo N, hope for collision modulo factor p.
E.g. using Floyd's cycle finding algorithm

```
N=698599699288686665490308069057420138223871
a=98357389475943875; c=10 # some random values
a1=(a^2+c) % N ; a2=(a1^2+c) % N
while gcd(N,a2-a1)==1:
    a1=(a1^2+c) %N
    a2=(((a2^2+c)%N)^2+c)%N
gcd(N,a2-a1) # output is 2053
```

Pollard's rho method runs till a prime p divides $a1 - a2$ and N.
By the birthday paradox expect collisions modulo p after $\sqrt{p}$ steps.
Each step is more expensive than trial division, so don't use this to find 5 but to find 2053.

Pollard's $p - 1$ method

If $a^r \equiv 1 \bmod p$ then $p \mid \gcd(a^r - 1, N)$.

Pollard's $p-1$ method

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

Pollard's $p-1$ method

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

```
N=44426601460658291157725536008128017297890787
4637194279031281180366057
r=lcm(range(1,2^22)) # this takes a while ...
s=Integer(pow(2,r,N))
gcd(s-1,N)
```

Pollard's $p-1$ method

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

```
N=44426601460658291157725536008128017297890787
4637194279031281180366057
r=lcm(range(1,2^22)) # this takes a while ...
s=Integer(pow(2,r,N))
gcd(s-1,N) # output is 1267650600228229401496703217601
```

This method finds larger factors than the rho method (in the same time) but only works for special primes.

Pollard's $p-1$ method

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

```
N=44426601460658291157725536008128017297890787
4637194279031281180366057
r=lcm(range(1,2^22)) # this takes a while ...
s=Integer(pow(2,r,N))
gcd(s-1,N) # output is 1267650600228229401496703217601
```

This method finds larger factors than the rho method (in the same time) but only works for special primes. Here
$p-1 = 2^6 \cdot 3^2 \cdot 5^2 \cdot 17 \cdot 227 \cdot 491 \cdot 991 \cdot 36559 \cdot 308129 \cdot 4161791$
has only small factors (aka. $p-1$ is *smooth*).

Outdated recommendation: avoid such primes, use only "strong primes". ECM (next pages) finds all primes.

ECM – Math description

Pollard's $p-1$ method uses multiplicative group of integers modulo p; finds p if $\text{ord}(a)_p$ divides r for some but not all primes p.

ECM – Math description

Pollard's $p-1$ method uses multiplicative group of integers modulo p; finds p if $\text{ord}(a)_p$ divides r for some but not all primes p.

Lenstra's Elliptic Curve Method uses the group of points on an elliptic curve modulo p. Let P be a point on the curve. If the order of P (under computations modulo p) divides r for some but not all primes p, can find p using an appropriate gcd with rP and N.

ECM – Math description

Pollard's $p - 1$ method uses multiplicative group of integers modulo p; finds p if $\text{ord}(a)_p$ divides r for some but not all primes p.

Lenstra's Elliptic Curve Method uses the group of points on an elliptic curve modulo p. Let P be a point on the curve. If the order of P (under computations modulo p) divides r for some but not all primes p, can find p using an appropriate gcd with rP and N.

Computations work as in $p - 1$ method: the curve is given modulo N; all arithmetic is done modulo N.

ECM – Math description

Pollard's $p-1$ method uses multiplicative group of integers modulo p; finds p if $\text{ord}(a)_p$ divides r for some but not all primes p.

Lenstra's Elliptic Curve Method uses the group of points on an elliptic curve modulo p. Let P be a point on the curve. If the order of P (under computations modulo p) divides r for some but not all primes p, can find p using an appropriate gcd with rP and N.

Computations work as in $p-1$ method: the curve is given modulo N; all arithmetic is done modulo N.

Hasse's theorem: the order of an elliptic curve modulo p is in $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$.

ECM – Math description

Pollard's $p-1$ method uses multiplicative group of integers modulo p; finds p if $\mathrm{ord}(a)_p$ divides r for some but not all primes p.

Lenstra's Elliptic Curve Method uses the group of points on an elliptic curve modulo p. Let P be a point on the curve. If the order of P (under computations modulo p) divides r for some but not all primes p, can find p using an appropriate gcd with rP and N.

Computations work as in $p-1$ method: the curve is given modulo N; all arithmetic is done modulo N.

Hasse's theorem: 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.

ECM – Math description

Pollard's $p-1$ method uses multiplicative group of integers modulo p; finds p if $\mathrm{ord}(a)_p$ divides r for some but not all primes p.

Lenstra's Elliptic Curve Method uses the group of points on an elliptic curve modulo p. Let P be a point on the curve. If the order of P (under computations modulo p) divides r for some but not all primes p, can find p using an appropriate gcd with rP and N.

Computations work as in $p-1$ method: the curve is given modulo N; all arithmetic is done modulo N.

Hasse's theorem: 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.
Lenstra: Good distribution in the interval.

ECM – Math description

Pollard's $p-1$ method uses multiplicative group of integers modulo p; finds p if $\mathrm{ord}(a)_p$ divides r for some but not all primes p.

Lenstra's Elliptic Curve Method uses the group of points on an elliptic curve modulo p. Let P be a point on the curve. If the order of P (under computations modulo p) divides r for some but not all primes p, can find p using an appropriate gcd with rP and N.

Computations work as in $p-1$ method: the curve is given modulo N; all arithmetic is done modulo N.

Hasse's theorem: 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.
Lenstra: Good distribution in the interval.

ECM has the power to change the group; if E_1 does not work, go for $E_2, E_3, \ldots$ till a point has smooth order modulo a p.

EECM: Edwards ECM, Basic version

Use Elliptic curve in twisted Edwards form:
$E : ax^2 + y^2 = 1 + dx^2y^2$ with point $P = (x, y)$; $a, d \neq 0, a \neq d$.
Generate random curve by picking random nonzero a, x, y,
compute $d = (ax^2 + y^2 - 1)/x^2y^2$.

Multiplication in $p - 1$ method replaced by addition on E:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1y_1x_2y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1y_1x_2y_1}\right).$$

Neutral element in this group is $(0, 1)$.

EECM: Edwards ECM, Basic version

Use Elliptic curve in twisted Edwards form:
$E : ax^2 + y^2 = 1 + dx^2y^2$ with point $P = (x, y)$; $a, d \neq 0, a \neq d$.
Generate random curve by picking random nonzero a, x, y,
compute $d = (ax^2 + y^2 - 1)/x^2y^2$.

Multiplication in $p - 1$ method replaced by addition on E:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1y_1x_2y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1y_1x_2y_1}\right).$$

Neutral element in this group is $(0, 1)$.

Compute $rP = (\bar{x}, \bar{y})$ modulo N using double-and-add method; avoid divisions by using *projective coordinates*. For formulas see `http://hyperelliptic.org/EFD`.

EECM: Edwards ECM, Basic version

Use Elliptic curve in twisted Edwards form:
$E : ax^2 + y^2 = 1 + dx^2y^2$ with point $P = (x, y)$; $a, d \neq 0, a \neq d$.
Generate random curve by picking random nonzero a, x, y,
compute $d = (ax^2 + y^2 - 1)/x^2y^2$.

Multiplication in $p - 1$ method replaced by addition on E:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1y_1x_2y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1y_1x_2y_1}\right).$$

Neutral element in this group is $(0, 1)$.

Compute $rP = (\bar{x}, \bar{y})$ modulo N using double-and-add method; avoid divisions by using *projective coordinates*. For formulas see `http://hyperelliptic.org/EFD`.

Compute $\gcd(\bar{x}, N)$; this finds primes p for which the order of P modulo p divides r.

ECM: production version

- Use special curves with
  - small coefficients for faster computation, e.g. $(1/23, 1/7)$ is a point on $25x^2 + y^2 = 1 - 24167x^2y^2$;
  - with better chance of smooth orders; this curve has a guaranteed factor of 12.
- Split computation into 2 stages:
  - stage 1 as described before with somewhat smaller t in `r=lcm(range(1,t))`;
  - stage 2 checks $(q_i r)P$ for the next few primes $q_i > t$ (computed in a batched manner).
- See `http://eecm.cr.yp.to/` for explanations, good curves, code, references, etc.

ECM: production version

- Use special curves with
  - small coefficients for faster computation, e.g. $(1/23, 1/7)$ is a point on $25x^2 + y^2 = 1 - 24167x^2y^2$;
  - with better chance of smooth orders; this curve has a guaranteed factor of 12.
- Split computation into 2 stages:
  - stage 1 as described before with somewhat smaller t in `r=lcm(range(1,t))`;
  - stage 2 checks $(q_i r)P$ for the next few primes $q_i > t$ (computed in a batched manner).
- See `http://eecm.cr.yp.to/` for explanations, good curves, code, references, etc.
- Method runs very well on GPUs; distributed computing.
- ECM is still active research area.

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

ECM: production version

- Use special curves with
  - small coefficients for faster computation, e.g. $(1/23, 1/7)$ is a point on $25x^2 + y^2 = 1 - 24167x^2y^2$;
  - with better chance of smooth orders; this curve has a guaranteed factor of 12.
- Split computation into 2 stages:
  - stage 1 as described before with somewhat smaller t in `r=lcm(range(1,t))`;
  - stage 2 checks $(q_i r)P$ for the next few primes $q_i > t$ (computed in a batched manner).
- See `http://eecm.cr.yp.to/` for explanations, good curves, code, references, etc.
- Method runs very well on GPUs; distributed computing.
- ECM is still active research area.

ECM is very efficient at factoring random numbers (once small factors are removed). Favorite method to kill RSA-360.

Factoring bad choices of N

Problem if one takes 'same size' too literally:

$$N = 1000000000000000000000000000000000000000000000000000000\
0000000000000000000000000000000000000000000000000000000000\
0000000000000000000000000000000000000000000000000000000029\
9999999999999999999999999999999999999999999999999999999999\
9999999999999999999999999999999999999999999999999999999999\
99999999999999999999999999999999999999999999999999997921.$$

Factoring bad choices of N

Problem if one takes 'same size' too literally:

N = 10000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000029
9999999999999999999999999999999999999999999999999999999999
9999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999997921.

Yes, this looks like very close to a power of 10, actually close to 10^{340}. Square root $\sqrt{N}$ is almost an integer, almost 10^{170}.

Factoring bad choices of N

Problem if one takes 'same size' too literally:

N = 1000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000029
9999999999999999999999999999999999999999999999999999999999
9999999999999999999999999999999999999999999999999999999999
999999999999999999999999999999999999999999999999997921.

Yes, this looks like very close to a power of 10, actually close to 10^{340}. Square root $\sqrt{N}$ is almost an integer, almost 10^{170}.

Brute-force search `N % (10^170-i)` finds factor $p = 10^{170} - 33$ and then $q = N/p = 10^{170} + 63$.

Factoring bad choices of N

Problem if one takes 'same size' too literally:

$$\begin{aligned}N = \ & 1000000000000000000000000000000000000000000000000000000\\ & 0000000000000000000000000000000000000000000000000000000000\\ & 0000000000000000000000000000000000000000000000000000000029\\ & 9999999999999999999999999999999999999999999999999999999999\\ & 9999999999999999999999999999999999999999999999999999999999\\ & 99999999999999999999999999999999999999999999999999997921.\end{aligned}$$

Yes, this looks like very close to a power of 10, actually close to 10^{340}. Square root $\sqrt{N}$ is almost an integer, almost 10^{170}.

Brute-force search `N % (10^170-i)` finds factor $p = 10^{170} - 33$ and then $q = N/p = 10^{170} + 63$.

In real life would expect this with power of 2 instead of 10.

This problem happens not only for p and q too close to powers of 2 or 10. User starts search for p with some offset c as $p = \texttt{next\_prime}(2^{512} + c)$. Takes $q = \texttt{next\_prime}(p)$.

```
sage: N=11579208923731619542357098500872121122114628
262713908746538761285902758367353
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463463374607431817146356.999999999999
999999999999999999999999940'
```

This problem happens not only for p and q too close to powers of 2 or 10. User starts search for p with some offset c as $p = \texttt{next\_prime}(2^{512} + c)$. Takes $q = \texttt{next\_prime}(p)$.

```
sage: N=115792089237316195423570985008721211221144628
262713908746538761285902758367353
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463463374607431817146356.999999999999
99999999999999999999999940' # very close to an integer
```

This problem happens not only for p and q too close to powers of 2 or 10. User starts search for p with some offset c as $p = \texttt{next\_prime}(2^{512} + c)$. Takes $q = \texttt{next\_prime}(p)$.

```
sage: N=11579208923731619542357098500872121122114628
262713908746538761285902758367353
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463463374607431817146356.999999999999
99999999999999999999999940' # very close to an integer
sage: a=ceil(sqrt(N)); a^2-N
4096
```

This problem happens not only for p and q too close to powers of 2 or 10. User starts search for p with some offset c as $p = \texttt{next\_prime}(2^{512} + c)$. Takes $q = \texttt{next\_prime}(p)$.

```
sage: N=11579208923731619542357098500872121122114628
2627139087465387612859027583673535
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463463374607431817146356.999999999999
99999999999999999999999999940' # very close to an integer
sage: a=ceil(sqrt(N)); a^2-N
4096  # 4096=64^2; this is a square!
```

This problem happens not only for p and q too close to powers of 2 or 10. User starts search for p with some offset c as $p = \texttt{next\_prime}(2^{512} + c)$. Takes $q = \texttt{next\_prime}(p)$.

```
sage: N=11579208923731619542357098500872121122114628
26271390874653876128590275836735З
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463463374607431817146356.999999999999
999999999999999999999999940' # very close to an integer
sage: a=ceil(sqrt(N)); a^2-N
4096  # 4096=64^2; this is a square!
sage: N/(a-64)
340282366920938463463374607431817146293
```

This problem happens not only for p and q too close to powers of 2 or 10. User starts search for p with some offset c as $p = \texttt{next\_prime}(2^{512} + c)$. Takes $q = \texttt{next\_prime}(p)$.

```
sage: N=115792089237316195423570985008721211221144628
262713908746538761285902758367353
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463463374607431817146356.999999999999
99999999999999999999999940' # very close to an integer
sage: a=ceil(sqrt(N)); a^2-N
4096  # 4096=64^2; this is a square!
sage: N/(a-64)
340282366920938463463374607431817146293 # an integer!
sage: N/340282366920938463463374607431817146293
340282366920938463463374607431817146421
```

Fermat factorization

We wrote $N = a^2 - b^2 = (a+b)(a-b)$ and factored it using $N/(a-b)$.

```
sage: N=11579208923731619544867939228200664041319989
0130332179010243714077028592474181
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463500268096066682468352.99999994715
09747085563508368188422193'
```

Fermat factorization

We wrote $N = a^2 - b^2 = (a+b)(a-b)$ and factored it using $N/(a-b)$.

```
sage: N=115792089237316195448679392282006640413199890130332179010243714077028592474181
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463500268096066682468352.99999994715097470855635083681884221 93'
sage: a=ceil(sqrt(N)); i=0
sage: while not is_square((a+i)^2-N):
....:   i=i+1
```

Fermat factorization

We wrote $N = a^2 - b^2 = (a+b)(a-b)$ and factored it using $N/(a-b)$.

```
sage: N=1157920892373161954486793922820066404131998901303321790102437140770285924741811
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463500268096066682468352.99999994715
09747085563508368188422193'
sage: a=ceil(sqrt(N)); i=0
sage: while not is_square((a+i)^2-N):
....:   i=i+1  # gives i=2
```

Fermat factorization

We wrote $N = a^2 - b^2 = (a + b)(a - b)$ and factored it using $N/(a - b)$.

```
sage: N=115792089237316195448679392282006640413199890130332179010243714077028592474181
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463500268096066682468352.99999994715
0974708556350836818842219 3'
sage: a=ceil(sqrt(N)); i=0
sage: while not is_square((a+i)^2-N):
....:   i=i+1  # gives i=2
....:          # was q=next_prime(p+2^66+974892437589)
```

This always works

Fermat factorization

We wrote $N = a^2 - b^2 = (a+b)(a-b)$ and factored it using $N/(a-b)$.

```
sage: N=115792089237316195448679392282006640413199890130332179010243714077028592474181
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463500268096066682468352.99999994715097470855635083681884221 93'
sage: a=ceil(sqrt(N)); i=0
sage: while not is_square((a+i)^2-N):
....:   i=i+1  # gives i=2
....:          # was q=next_prime(p+2^66+974892437589)
```

This always works eventually: $N = ((q+p)/2)^2 - ((q-p)/2)^2$

Fermat factorization

We wrote $N = a^2 - b^2 = (a+b)(a-b)$ and factored it using $N/(a-b)$.

```
sage: N=115792089237316195448679392282006640413199890130332179010243714077028592474181
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463500268096066682468352.99999994715097470855635083681884221 93'
sage: a=ceil(sqrt(N)); i=0
sage: while not is_square((a+i)^2-N):
....:   i=i+1  # gives i=2
....:          # was q=next_prime(p+2^66+974892437589)
```

This always works eventually: $N = ((q+p)/2)^2 - ((q-p)/2)^2$ but searching for $(q+p)/2$ starting with $\lceil\sqrt{N}\rceil$ will usually run for about $\sqrt{N} \approx p$ steps.

An example of the quadratic sieve (QS)

Let's try Fermat to factor $N = 2759$. Recall idea:
if $a^2 - N$ is a square b^2 then $N = (a-b)(a+b)$.

$53^2 - 2759 = 50$. Not exactly a square: $50 = 2 \cdot 5^2$.

An example of the quadratic sieve (QS)

Let's try Fermat to factor $N = 2759$. Recall idea:
if $a^2 - N$ is a square b^2 then $N = (a - b)(a + b)$.

$53^2 - 2759 = 50$. Not exactly a square: $50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$. Ummm, doesn't look like a square.

An example of the quadratic sieve (QS)

Let's try Fermat to factor $N = 2759$. Recall idea:
if $a^2 - N$ is a square b^2 then $N = (a - b)(a + b)$.

$53^2 - 2759 = 50$. Not exactly a square: $50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$. Ummm, doesn't look like a square.
$55^2 - 2759 = 266$.

An example of the quadratic sieve (QS)

Let's try Fermat to factor $N = 2759$. Recall idea:
if $a^2 - N$ is a square b^2 then $N = (a - b)(a + b)$.

$53^2 - 2759 = 50$. Not exactly a square: $50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$. Ummm, doesn't look like a square.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.

An example of the quadratic sieve (QS)

Let's try Fermat to factor $N = 2759$. Recall idea:
if $a^2 - N$ is a square b^2 then $N = (a - b)(a + b)$.

$53^2 - 2759 = 50$. Not exactly a square: $50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$. Ummm, doesn't look like a square.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.
$57^2 - 2759 = 490$. Hey, 49 is a square ... $490 = 2 \cdot 5 \cdot 7^2$.

An example of the quadratic sieve (QS)

Let's try Fermat to factor $N = 2759$. Recall idea:
if $a^2 - N$ is a square b^2 then $N = (a-b)(a+b)$.

$53^2 - 2759 = 50$. Not exactly a square: $50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$. Ummm, doesn't look like a square.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.
$57^2 - 2759 = 490$. Hey, 49 is a square ... $490 = 2 \cdot 5 \cdot 7^2$.
$58^2 - 2759 = 605$. Not exactly a square: $605 = 5 \cdot 11^2$.

An example of the quadratic sieve (QS)

Let's try Fermat to factor $N = 2759$. Recall idea:
if $a^2 - N$ is a square b^2 then $N = (a-b)(a+b)$.

$53^2 - 2759 = 50$. Not exactly a square: $50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$. Ummm, doesn't look like a square.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.
$57^2 - 2759 = 490$. Hey, 49 is a square ... $490 = 2 \cdot 5 \cdot 7^2$.
$58^2 - 2759 = 605$. Not exactly a square: $605 = 5 \cdot 11^2$.

Fermat doesn't seem to be working very well for this number.

An example of the quadratic sieve (QS)

Let's try Fermat to factor $N = 2759$. Recall idea:
if $a^2 - N$ is a square b^2 then $N = (a-b)(a+b)$.

$53^2 - 2759 = 50$. Not exactly a square: $50 = 2 \cdot 5^2$.
$54^2 - 2759 = 157$. Ummm, doesn't look like a square.
$55^2 - 2759 = 266$.
$56^2 - 2759 = 377$.
$57^2 - 2759 = 490$. Hey, 49 is a square … $490 = 2 \cdot 5 \cdot 7^2$.
$58^2 - 2759 = 605$. Not exactly a square: $605 = 5 \cdot 11^2$.

Fermat doesn't seem to be working very well for this number.

But the *product* $50 \cdot 490 \cdot 605$ is a square: $2^2 \cdot 5^4 \cdot 7^2 \cdot 11^2$.
QS computes $\gcd\{2759, 53 \cdot 57 \cdot 58 - \sqrt{50 \cdot 490 \cdot 605}\} = 31$.

Exercise: Square product has 50% chance of factoring pq.

QS more systematically

Try larger N. Easy to generate many differences $a^2 - N$:

```
N = 314159265358979323
X = [a^2-N for a in range(sqrt(N)+1,sqrt(N)+500000)]
```

QS more systematically

Try larger N. Easy to generate many differences $a^2 - N$:

```
N = 314159265358979323
X = [a^2-N for a in range(sqrt(N)+1,sqrt(N)+500000)]
```

See which differences are easy to factor:

```
P = list(primes(2,1000))
F = easyfactorizations(P,X)
```

QS more systematically

Try larger N. Easy to generate many differences $a^2 - N$:

```
N = 314159265358979323
X = [a^2-N for a in range(sqrt(N)+1,sqrt(N)+500000)]
```

See which differences are easy to factor:

```
P = list(primes(2,1000))
F = easyfactorizations(P,X)
```

Use linear algebra mod 2 to find a square:

```
M = matrix(GF(2),len(F),len(P),lambda i,j:P[j] in F[i][0])
for K in M.left_kernel().basis():
  x = product([sqrt(f[2]+N) for f,k in zip(F,K) if k==1])
  y = sqrt(product([f[2] for f,k in zip(F,K) if k==1]))
  print [gcd(N,x - y),gcd(N,x + y)]
```

Strategies to implement easyfactorizations

Trial-dividing $a^2 - N$ using primes in $[1, y]$ costs $y^{1+o(1)}$.
Four major directions of improvements:

- Early aborts: e.g., throw $a^2 - N$ away if unfactored part is uncomfortably large after primes in $[1, y^{0.5}]$.
  1982 Pomerance: optimized early aborts reduce cost of trial division to $y^{0+o(1)}$ while reducing effectiveness by factor $y^{0.5+o(1)}$.

Strategies to implement `easyfactorizations`

Trial-dividing $a^2 - N$ using primes in $[1, y]$ costs $y^{1+o(1)}$.
Four major directions of improvements:

- Early aborts: e.g., throw $a^2 - N$ away if unfactored part is uncomfortably large after primes in $[1, y^{0.5}]$.
  1982 Pomerance: optimized early aborts reduce cost of trial division to $y^{0+o(1)}$ while reducing effectiveness by factor $y^{0.5+o(1)}$.
- Batch trial division: same as tree idea from before.

Strategies to implement `easyfactorizations`

Trial-dividing $a^2 - N$ using primes in $[1, y]$ costs $y^{1+o(1)}$.
Four major directions of improvements:

- Early aborts: e.g., throw $a^2 - N$ away if unfactored part is uncomfortably large after primes in $[1, y^{0.5}]$.
  1982 Pomerance: optimized early aborts reduce cost of trial division to $y^{0+o(1)}$ while reducing effectiveness by factor $y^{0.5+o(1)}$.
- Batch trial division: same as tree idea from before.
- "Sieving": like the Sieve of Eratosthenes. Example: use arithmetic progressions of a with 1009 dividing $a^2 - N$.

Strategies to implement easyfactorizations

Trial-dividing $a^2 - N$ using primes in $[1, y]$ costs $y^{1+o(1)}$.
Four major directions of improvements:

- Early aborts: e.g., throw $a^2 - N$ away if unfactored part is uncomfortably large after primes in $[1, y^{0.5}]$.
  1982 Pomerance: optimized early aborts reduce cost of trial division to $y^{0+o(1)}$ while reducing effectiveness by factor $y^{0.5+o(1)}$.
- Batch trial division: same as tree idea from before.
- "Sieving": like the Sieve of Eratosthenes. Example: use arithmetic progressions of a with 1009 dividing $a^2 - N$.
- rho, $p - 1$, $p + 1$, ECM. **Low memory, high parallelism.**

Sieving seemed very important 30 years ago. Today much less use: we care more about communication cost and lattice optimization.

Interlude: Smoothness

How many integers in $[1, y^2]$ factor into primes in $[1, y]$?

Interlude: Smoothness

How many integers in $[1, y^2]$ factor into primes in $[1, y]$?

Easy lower bound: at least $\approx 0.5y^2/(\log y)^2$.
(There are $\approx y/\log y$ primes in $[1, y]$.
Consider products of two such primes.)

Interlude: Smoothness

How many integers in $[1, y^2]$ factor into primes in $[1, y]$?

Easy lower bound: at least $\approx 0.5y^2/(\log y)^2$.
(There are $\approx y/\log y$ primes in $[1, y]$.
Consider products of two such primes.)

Somewhat careful analysis: constant times y^2.

Interlude: Smoothness

How many integers in $[1, y^2]$ factor into primes in $[1, y]$?

Easy lower bound: at least $\approx 0.5 y^2 / (\log y)^2$.
(There are $\approx y / \log y$ primes in $[1, y]$.
Consider products of two such primes.)

Somewhat careful analysis: constant times y^2.

More careful analysis: $\approx 0.306 y^2$.

Interlude: Smoothness

How many integers in $[1, y^2]$ factor into primes in $[1, y]$?

Easy lower bound: at least $\approx 0.5y^2/(\log y)^2$.
(There are $\approx y/\log y$ primes in $[1, y]$.
Consider products of two such primes.)

Somewhat careful analysis: constant times y^2.

More careful analysis: $\approx 0.306y^2$.

How many integers in $[1, y^u]$ factor into primes in $[1, y]$?
Somewhat careful analysis: $\approx u^{-u}y^u$.
More careful analysis: e.g., $\approx 0.277 \cdot 10^{-10}y^u$ for $u = 10$.

QS scalability

QS is slow for small N ... but scales very well to larger N.

Choose $y = N^{1/u}$.
If differences $a^2 - N$ were random integers mod N
then they would factor into primes in $[1, y]$ with probability $\approx u^{-u}$.
(Actually $a^2 - N$ is closer to $\sqrt{N}$; even more likely to factor.)

Factorization exponent vectors produce linear dependencies
once there are $\approx u^u y / \log y$ differences.

Choose u on scale of $\sqrt{\log N / \log\log N}$
to balance u^u with $N^{1/u}$. Subexponential cost!

History of the world, part 1

1931 Lehmer–Powers, 1975 Morrison–Brillhart, "CFRAC": find small squares mod N using $\sqrt{N}$ continued fraction.

1977 Schroeppel "linear sieve": find square products of $ab(ab - N)$ by sieving $ab - N$; use a, b in small range around $\sqrt{N}$. This uses $\exp(O(\sqrt{\log N \log\log N}))$ operations.

1982 Pomerance, QS: $a^2 - N$.

History of the world, part 1

1931 Lehmer–Powers, 1975 Morrison–Brillhart, "CFRAC": find small squares mod N using $\sqrt{N}$ continued fraction.

1977 Schroeppel "linear sieve": find square products of $ab(ab - N)$ by sieving $ab - N$; use a, b in small range around $\sqrt{N}$. This uses $\exp(O(\sqrt{\log N \log\log N}))$ operations.

1982 Pomerance, QS: $a^2 - N$.

Retroactively plug in ECM or batch trial division, and fast linear algebra: each method uses $\exp((1 + o(1))\sqrt{\log N \log\log N})$ operations.

History of the world, part 1

1931 Lehmer–Powers, 1975 Morrison–Brillhart, "CFRAC": find small squares mod N using $\sqrt{N}$ continued fraction.

1977 Schroeppel "linear sieve": find square products of $ab(ab - N)$ by sieving $ab - N$; use a, b in small range around $\sqrt{N}$. This uses $\exp(O(\sqrt{\log N \log\log N}))$ operations.

1982 Pomerance, QS: $a^2 - N$.

Retroactively plug in ECM or batch trial division, and fast linear algebra: each method uses $\exp((1 + o(1))\sqrt{\log N \log\log N})$ operations.

Applying ECM directly to N also uses $\exp((1 + o(1))\sqrt{\log N \log\log N})$ operations.

History of the world, part 2

1982 Schnorr, 1987 Seysen, 1988 A. Lenstra,
1992 H. Lenstra–Pomerance: another method
that *provably* uses $\exp((1 + o(1))\sqrt{\log N \log\log N})$ operations.

1988 Pomerance–Smith–Tuler:
"Over the last few years there has developed a remarkable six-way tie for the asymptotically fastest factoring algorithms. ... It might be tempting to conjecture that $L(N)$ is in fact the true complexity of factoring, but no one seems to have any idea how to obtain even heuristic lower bounds for factoring."

History of the world, part 2

1982 Schnorr, 1987 Seysen, 1988 A. Lenstra,
1992 H. Lenstra–Pomerance: another method
that *provably* uses $\exp((1 + o(1))\sqrt{\log N \log\log N})$ operations.

1988 Pomerance–Smith–Tuler:
"Over the last few years there has developed a remarkable six-way tie for the asymptotically fastest factoring algorithms. . . . It might be tempting to conjecture that $L(N)$ is in fact the true complexity of factoring, but no one seems to have any idea how to obtain even heuristic lower bounds for factoring."

1985 Odlyzko, commenting on the same conjecture:
"It is this author's guess that this is not the case, and that we are missing some insight that will let us break below the $L(p)$ barrier."

The number-field sieve (NFS)

1988 Pollard, independently 1989 Elkies,
generalized by 1990 Lenstra–Lenstra–Manasse–Pollard:
Use $(a + b\alpha)(a + bm)$ with $\alpha \equiv m \pmod{n}$.
$\exp((2.08\ldots + o(1))(\log N)^{1/3}(\log\log N)^{2/3})$.

1991 Adleman, 1993 Buhler–Lenstra–Pomerance:
$\exp((1.92\ldots + o(1))(\log N)^{1/3}(\log\log N)^{2/3})$.
Adleman estimated QS/NFS cutoff as $N \approx 2^{1100}$.

1993 Coppersmith:
$\exp((1.90\ldots + o(1))(\log N)^{1/3}(\log\log N)^{2/3})$.

1993 Coppersmith, batch NFS ("factorization factory"):
$\exp((1.63\ldots + o(1))(\log N)^{1/3}(\log\log N)^{2/3})$
after a precomputation independent of N.

So what does this mean for RSA-1024?

Complicated NFS analysis and optimization. Latest estimates:
Attacker breaks my 1024-bit key by scanning $\approx 2^{70}$ pairs (a, b).

Plan A: NSA is building a 2^{26}-watt computer center in Bluffdale.
Plan B: The Conficker botnet broke into $\approx 2^{23}$ machines.
Plan C: China has a supercomputer center in Tianjin.

So what does this mean for RSA-1024?

Complicated NFS analysis and optimization. Latest estimates:
Attacker breaks my 1024-bit key by scanning $\approx 2^{70}$ pairs (a, b).

Plan A: NSA is building a 2^{26}-watt computer center in Bluffdale.
Plan B: The Conficker botnet broke into $\approx 2^{23}$ machines.
Plan C: China has a supercomputer center in Tianjin.

2^{57} watts	Earth receives from the Sun
2^{56} watts	Earth's surface receives from the Sun
2^{44} watts	
2^{30} watts	
2^{26} watts	

So what does this mean for RSA-1024?

Complicated NFS analysis and optimization. Latest estimates:
Attacker breaks my 1024-bit key by scanning $\approx 2^{70}$ pairs (a, b).

Plan A: NSA is building a 2^{26}-watt computer center in Bluffdale.
Plan B: The Conficker botnet broke into $\approx 2^{23}$ machines.
Plan C: China has a supercomputer center in Tianjin.

2^{57} watts	Earth receives from the Sun
2^{56} watts	Earth's surface receives from the Sun
2^{44} watts	Current world power usage
2^{30} watts	
2^{26} watts	

So what does this mean for RSA-1024?

Complicated NFS analysis and optimization. Latest estimates:
Attacker breaks my 1024-bit key by scanning $\approx 2^{70}$ pairs (a, b).

Plan A: NSA is building a 2^{26}-watt computer center in Bluffdale.
Plan B: The Conficker botnet broke into $\approx 2^{23}$ machines.
Plan C: China has a supercomputer center in Tianjin.

2^{57} watts	Earth receives from the Sun
2^{56} watts	Earth's surface receives from the Sun
2^{44} watts	Current world power usage
2^{30} watts	Botnet running 2^{23} typical CPUs
2^{26} watts	

So what does this mean for RSA-1024?

Complicated NFS analysis and optimization. Latest estimates:
Attacker breaks my 1024-bit key by scanning $\approx 2^{70}$ pairs (a, b).

Plan A: NSA is building a 2^{26}-watt computer center in Bluffdale.
Plan B: The Conficker botnet broke into $\approx 2^{23}$ machines.
Plan C: China has a supercomputer center in Tianjin.

2^{57} watts	Earth receives from the Sun
2^{56} watts	Earth's surface receives from the Sun
2^{44} watts	Current world power usage
2^{30} watts	Botnet running 2^{23} typical CPUs
2^{26} watts	One dinky little computer center

So what does this mean for RSA-1024?

Complicated NFS analysis and optimization. Latest estimates:
Attacker breaks my 1024-bit key by scanning $\approx 2^{70}$ pairs (a, b).

Plan A: NSA is building a 2^{26}-watt computer center in Bluffdale.
Plan B: The Conficker botnet broke into $\approx 2^{23}$ machines.
Plan C: China has a supercomputer center in Tianjin.

2^{57} watts	Earth receives from the Sun
2^{56} watts	Earth's surface receives from the Sun
2^{44} watts	Current world power usage
2^{30} watts	Botnet running 2^{23} typical CPUs
2^{26} watts	One dinky little computer center

2^{26} watts of standard GPUs: 2^{84} floating-point mults/year.
Latest estimates: This is enough to break 1024-bit RSA.
... and special-purpose chips should be at least $10\times$ faster.
... and batch NFS should be even faster.

Quantum computers

Okay, you're using RSA-3072.

. . . and then the attacker builds a big quantum computer.
Imagine extreme case: qubit ops are about as cheap as bit ops.

Quantum computers

Okay, you're using RSA-3072.

... and then the attacker builds a big quantum computer.
Imagine extreme case: qubit ops are about as cheap as bit ops.

Major impact, part 1: 1996 Grover.
Speeds up searching s possible roots of f
from $\approx s$ iterations of f to $\approx\sqrt{s}$ iterations of f.
Example (2010 Bernstein): This speeds up ECM!

Quantum computers

Okay, you're using RSA-3072.

... and then the attacker builds a big quantum computer.
Imagine extreme case: qubit ops are about as cheap as bit ops.

Major impact, part 1: 1996 Grover.
Speeds up searching s possible roots of f
from $\approx s$ iterations of f to $\approx\sqrt{s}$ iterations of f.
Example (2010 Bernstein): This speeds up ECM!

Major impact, part 2: 1994 Shor.
Factors N using *one exponentiation modulo N*.

Post-quantum RSA

Conventional wisdom:
Shor's algorithm supersedes all previous factorization methods.
In fact, it breaks RSA as quickly as RSA decrypts,
so we have no hope of security from scaling RSA key sizes.

Post-quantum RSA

Conventional wisdom:
Shor's algorithm supersedes all previous factorization methods.
In fact, it breaks RSA as quickly as RSA decrypts,
so we have no hope of security from scaling RSA key sizes.

This isn't true!
Use "multi-prime RSA."

Post-quantum RSA

Conventional wisdom:
Shor's algorithm supersedes all previous factorization methods.
In fact, it breaks RSA as quickly as RSA decrypts,
so we have no hope of security from scaling RSA key sizes.

This isn't true!
Use "multi-prime RSA." Oops, 1997/1998 Tandem patent.

Post-quantum RSA

Conventional wisdom:
Shor's algorithm supersedes all previous factorization methods.
In fact, it breaks RSA as quickly as RSA decrypts,
so we have no hope of security from scaling RSA key sizes.

This isn't true!
Use "multi-prime RSA." Oops, 1997/1998 Tandem patent.
Fortunately, already in 1983 RSA patent: "the present invention may use a modulus n which is a product of three or more primes."

Post-quantum RSA

Conventional wisdom:
Shor's algorithm supersedes all previous factorization methods.
In fact, it breaks RSA as quickly as RSA decrypts,
so we have no hope of security from scaling RSA key sizes.

This isn't true!
Use "multi-prime RSA." Oops, 1997/1998 Tandem patent.
Fortunately, already in 1983 RSA patent: "the present invention may use a modulus n which is a product of three or more primes."

Concrete analysis suggests that RSA with 2^{31} 4096-bit primes provides $>2^{100}$ security vs. all known quantum attacks.
Key fits on a hard drive; encryption+decryption take only a week.