FactHacks: RSA factorization in the real world

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

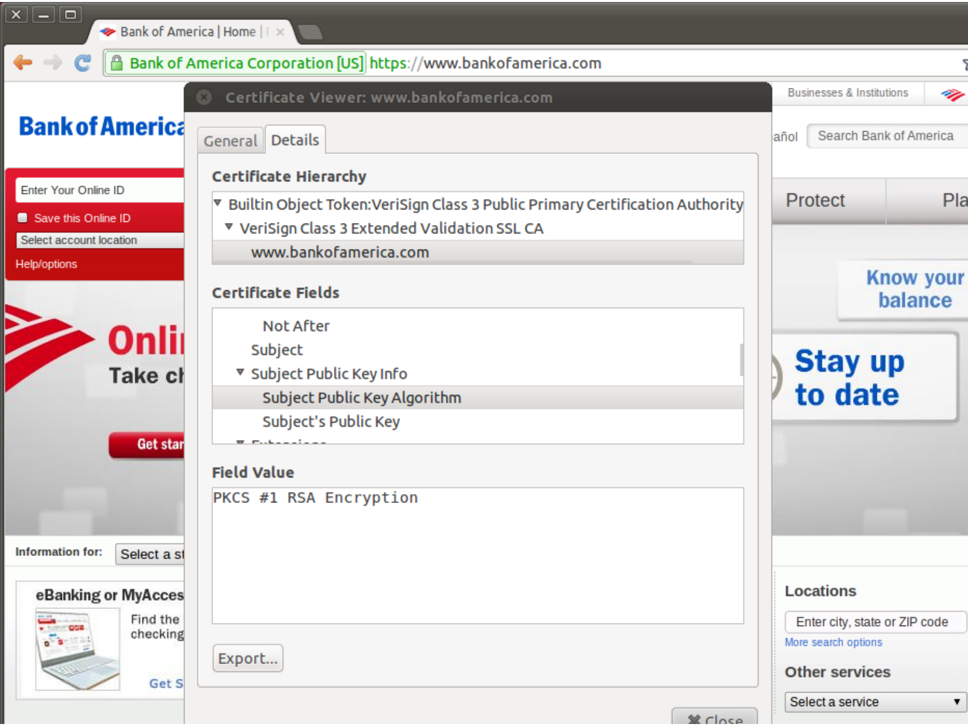
Nadia Heninger
Microsoft Research New England

Tanja Lange
Technische Universiteit Eindhoven

http://facthacks.cr.yp.to

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

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



Bank of America | Home |

Bank of America Corporation [US] https://www.bankofamerica.com

Certificate Viewer: www.bankofamerica.com

General | Details

Certificate Hierarchy

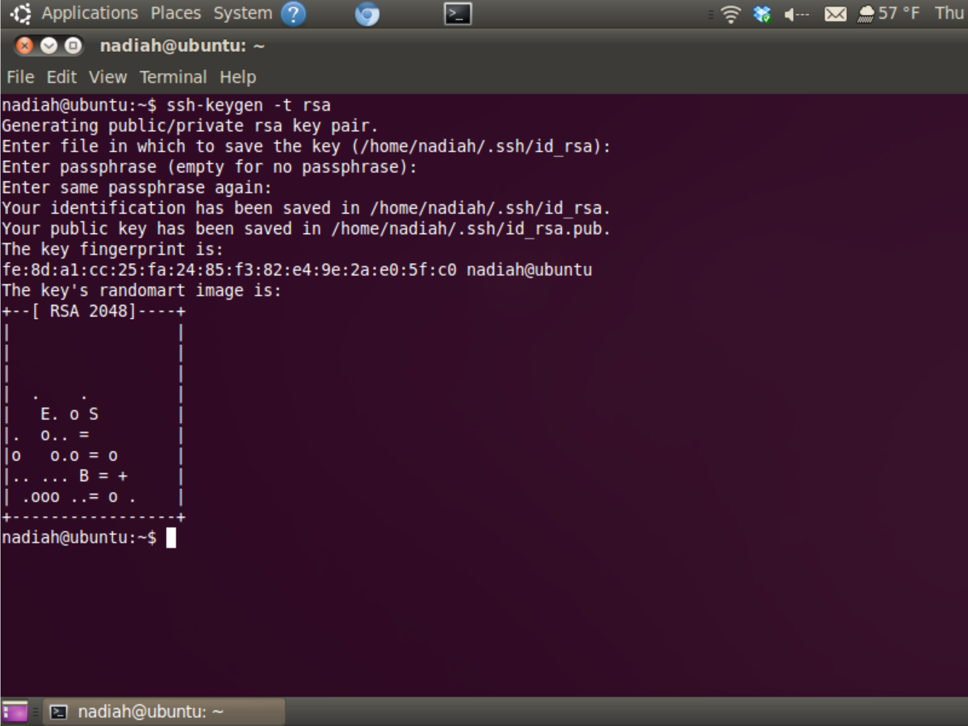
- Builtin Object Token:VeriSign Class 3 Public Primary Certification Authority
  - VeriSign Class 3 Extended Validation SSL CA
    - www.bankofamerica.com

Certificate Fields

- Not After
- Subject
- Subject Public Key Info
  - Subject Public Key Algorithm
  - Subject's Public Key

Field Value

```
PKCS #1 RSA Encryption
```



```
nadiah@ubuntu:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/nadiah/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/nadiah/.ssh/id_rsa.
Your public key has been saved in /home/nadiah/.ssh/id_rsa.pub.
The key fingerprint is:
fe:8d:a1:cc:25:fa:24:85:f3:82:e4:9e:2a:e0:5f:c0 nadiah@ubuntu
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|                 |
|                 |
|  .    .         |
|   E. o S        |
|.  o.. =         |
|o   o.o = o      |
|.. ... B = +     |
| .ooo ..= o .    |
+-----------------+
nadiah@ubuntu:~$ 
```

Preliminaries: Using Sage

We wanted to give you working code examples. We're going to use 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

We wanted to give you working code examples. We're going to use Sage.

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

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

^ is exponentiation, not xor

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

Preliminaries: Using Sage

We wanted to give you working code examples. We're going to use Sage.

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

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

^ is exponentiation, not xor

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

It has lots of useful libraries:

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

Preliminaries: Using Sage

We wanted to give you working code examples. We're going to use Sage.

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

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

^ is exponentiation, not xor

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

It has lots of useful libraries:

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

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

RSA Review

```
p = random_prime(2^512)
q = random_prime(2^512)
```

RSA Review

```
p = random_prime(2^512)
q = random_prime(2^512)
```

Public Key

```
N = p*q
e = 3
```

$\leftarrow$ or 65537 or 35...

RSA Review

Public Key

```
p = random_prime(2^512)
q = random_prime(2^512)
```

```
N = p*q
e = 3
```

or 65537 or 35...

Private Key

```
d = inverse_mod(e,(p-1)*(q-1))
```

RSA Review

Public Key

```
p = random_prime(2^512)
q = random_prime(2^512)
```

```
N = p*q
e = 3
```

or 65537 or 35...

Private Key

```
d = inverse_mod(e,(p-1)*(q-1))
```

```
message^e % n
```

Decryption

```
message = pow(ciphertext,d,n)
```

Encryption

```
ciphertext = pow(message,e,n)
```

RSA Review

Public Key

```
p = random_prime(2^512)
q = random_prime(2^512)
```

```
N = p*q
e = 3
```

or 65537 or 35...

Private Key

```
d = inverse_mod(e,(p-1)*(q-1))
```

```
message^e % n
```

Decryption

```
message = pow(ciphertext,d,n)
```

Encryption

```
ciphertext = pow(message,e,n)
```

Warning: You *must* use message padding.

RSA and factoring

Private Key

```
d = inverse_mod(e,(p-1)*(q-1))
```

Public Key

```
N = p*q
e = 3
```

- **Fact:** If we can factor N, can compute private key from public key.
- Factoring might not be the only way to break RSA: might be some way to compute message from ciphertext that doesn't reveal d or factorization of N. We don't know.
- **Fact:** Factoring not known to be NP-hard. It probably isn't.

So how hard *is* factoring?

So how hard *is* factoring?

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

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
170795249 * 1091258383
Time: CPU 0.01 s, Wall: 0.01 s
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
170795249 * 1091258383
Time: CPU 0.01 s, Wall: 0.01 s
sage: time factor(random_prime(2^64)*random_prime(2^64))
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
170795249 * 1091258383
Time: CPU 0.01 s, Wall: 0.01 s
sage: time factor(random_prime(2^64)*random_prime(2^64))
4711473922727062493 * 14104094416937800129
Time: CPU 0.13 s, Wall: 0.15 s
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
170795249 * 1091258383
Time: CPU 0.01 s, Wall: 0.01 s
sage: time factor(random_prime(2^64)*random_prime(2^64))
4711473922727062493 * 14104094416937800129
Time: CPU 0.13 s, Wall: 0.15 s
sage: time factor(random_prime(2^96)*random_prime(2^96))
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
170795249 * 1091258383
Time: CPU 0.01 s, Wall: 0.01 s
sage: time factor(random_prime(2^64)*random_prime(2^64))
4711473922727062493 * 14104094416937800129
Time: CPU 0.13 s, Wall: 0.15 s
sage: time factor(random_prime(2^96)*random_prime(2^96))
4602215373378843555620133613 * 33422266165499970562761950677
Time: CPU 4.64 s, Wall: 4.76 s
```

So how hard *is* factoring?

```
sage: time factor(random_prime(2^32)*random_prime(2^32))
170795249 * 1091258383
Time: CPU 0.01 s, Wall: 0.01 s
sage: time factor(random_prime(2^64)*random_prime(2^64))
4711473922727062493 * 14104094416937800129
Time: CPU 0.13 s, Wall: 0.15 s
sage: time factor(random_prime(2^96)*random_prime(2^96))
460221537337884355620133613 * 334222661654999705627619506
Time: CPU 4.64 s, Wall: 4.76 s
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))
170795249 * 1091258383
Time: CPU 0.01 s, Wall: 0.01 s
sage: time factor(random_prime(2^64)*random_prime(2^64))
4711473922727062493 * 14104094416937800129
Time: CPU 0.13 s, Wall: 0.15 s
sage: time factor(random_prime(2^96)*random_prime(2^96))
4602215373378843555620133613 * 3342226616549997056276195067
Time: CPU 4.64 s, Wall: 4.76 s
sage: time factor(random_prime(2^128)*random_prime(2^128))
249431539076558964376759054734817465081 * 29785758332242892
Time: CPU 506.95 s, Wall: 507.41 s
```

Danger: Bad random-number generators

Can the attacker get lucky and *guess* your p?

"No. There are $>2^{502}$ primes between 2^{511} and 2^{512}.
Each guess has chance $<2^{-501}$ of matching *your* p or q."

Danger: Bad random-number generators

Can the attacker get lucky and *guess* your p?

"No. There are $>2^{502}$ primes between 2^{511} and 2^{512}.
Each guess has chance $<2^{-501}$ of matching *your* p or q."

What if your system's random-number generator is busted?
What if it's generating only 2^{40} different choices for p?

The hard attack

Download a target user's public key $N = pq$.

Buy a bunch of devices.
Try different software configurations.
Generate billions of *private* keys.
Check whether any of these primes divide the target N.

Does anyone screw up random-number generation so badly?

The hard attack

Download a target user's public key $N = pq$.

Buy a bunch of devices.
Try different software configurations.
Generate billions of *private* keys.
Check whether any of these primes divide the target N.

Does anyone screw up random-number generation so badly?
Yes!

The hard attack

Download a target user's public key $N = pq$.

Buy a bunch of devices.
Try different software configurations.
Generate billions of *private* keys.
Check whether any of these primes divide the target N.

Does anyone screw up random-number generation so badly? Yes!

1995 Goldberg–Wagner: During any particular second,
the Netscape browser generates only $\approx 2^{47}$ possible keys.

The hard attack

Download a target user's public key $N = pq$.

Buy a bunch of devices.
Try different software configurations.
Generate billions of *private* keys.
Check whether any of these primes divide the target N.

Does anyone screw up random-number generation so badly?
Yes!

1995 Goldberg–Wagner: During any particular second,
the Netscape browser generates only $\approx 2^{47}$ possible keys.

2008 Bello: Since 2006, Debian and Ubuntu
are generating $< 2^{20}$ possible keys for SSH, OpenVPN, etc.

The easy attack

Download *two* target public keys N_1, N_2.
Hope that they share a prime p: i.e., $N_1 = pq_1$, $N_2 = pq_2$.
Not a surprise if $N_1 = N_2$, but what if $N_1 \neq N_2$?

The easy attack

Download *two* target public keys N_1, N_2.
Hope that they share a prime p: i.e., $N_1 = pq_1$, $N_2 = pq_2$.
Not a surprise if $N_1 = N_2$, but what if $N_1 \neq N_2$?

Euclid's algorithm prints the shared prime p given N_1, N_2.

```
def greatestcommondivisor(n1,n2):
  # Euclid's algorithm
  while n1 != 0: n1,n2 = n2%n1,n1
  return abs(n2)
```

Built into Sage as gcd.

A small example of Euclid's algorithm

```
sage: n1,n2 = 4187,5989
sage: n1,n2 = n2%n1,n1; print n1
1802
sage: n1,n2 = n2%n1,n1; print n1
583
sage: n1,n2 = n2%n1,n1; print n1
53
sage: n1,n2 = n2%n1,n1; print n1
0
sage: 4187/53  # / is exact division, as in Python 3
79
sage: 5989/53  # use // if you want rounding division
113
```

So $\gcd\{4187, 5989\} = 53$ and $4187 = 53 \cdot 79$ and $5989 = 53 \cdot 113$.

Euclid's algorithm is very fast

```
sage: p = random_prime(2^512)
sage: q1 = random_prime(2^512)
sage: q2 = random_prime(2^512)
sage: time g = gcd(p*q1,p*q2)
Time: CPU 0.00 s, Wall: 0.00 s
sage: g == p
True
```

Finding shared factors of many inputs

Download millions of public keys $N_1, N_2, N_3, N_4, \ldots$.
There are **millions of millions** of pairs to try:
(N_1, N_2); (N_1, N_3); (N_2, N_3); (N_1, N_4); (N_2, N_4); etc.

Finding shared factors of many inputs

Download millions of public keys $N_1, N_2, N_3, N_4, \ldots$.
There are **millions of millions** of pairs to try:
(N_1, N_2); (N_1, N_3); (N_2, N_3); (N_1, N_4); (N_2, N_4); etc.

That's feasible; but **batch gcd** finds the shared primes much faster.

Our real goal is to compute
$\gcd\{N_1, N_2 N_3 N_4 \cdots\}$ (this gcd is > 1 if N_1 shares a prime);
$\gcd\{N_2, N_1 N_3 N_4 \cdots\}$ (this gcd is > 1 if N_2 shares a prime);
$\gcd\{N_3, N_1 N_2 N_4 \cdots\}$ (this gcd is > 1 if N_3 shares a prime);
etc.

Batch gcd, part 1: product tree

First step: Multiply all the keys! Compute $R = N_1 N_2 N_3 \cdots$.

```
def producttree(X):
  result = [X]
  while len(X) > 1:
    X = [prod(X[i*2:(i+1)*2])
         for i in range((len(X)+1)/2)]
    result.append(X)
  return result

# for example:
print producttree([10,20,30,40])
# output is [[10, 20, 30, 40], [200, 1200], [240000]]
```

Batch gcd, part 2: remainder tree

Reduce $R = N_1 N_2 N_3 \cdots$ modulo N_1^2 and N_2^2 and N_3^2 and so on.

Obtain $\gcd\{N_1, N_2 N_3 \cdots\}$ as $\gcd\{N_1, (R \bmod N_1^2)/N_1\}$;
obtain $\gcd\{N_2, N_1 N_3 \cdots\}$ as $\gcd\{N_2, (R \bmod N_2^2)/N_2\}$;
etc.

```
def batchgcd(X):
  prods = producttree(X)
  R = prods.pop()
  while prods:
    X = prods.pop()
    R = [R[floor(i/2)] % X[i]**2 for i in range(len(X))]
  return [gcd(r/n,n) for r,n in zip(R,X)]
```

Batch gcd is very fast

```
sage: # two-year-old laptop, clocked down to 800MHz
sage: def myrand():
....:      return Integer(randrange(2^1024))
....:
sage: time g = batchgcd([myrand() for i in range(100)])
Time: CPU 0.05 s, Wall: 0.05 s
sage: time g = batchgcd([myrand() for i in range(1000)])
Time: CPU 1.08 s, Wall: 1.08 s
sage: time g = batchgcd([myrand() for i in range(10000)])
Time: CPU 23.21 s, Wall: 23.29 s
sage:
```

Are random-number generators really this bad?

Are random-number generators really this bad?

2012 Heninger–Durumeric–Wustrow–Halderman,
best-paper award at USENIX Security Symposium:

Factored tens of thousands of public keys on the Internet
... typically keys for your home router, not for your bank.
Why? **Many deployed devices are generating guessable p's.**
Most common problem: horrifyingly bad interactions between
OpenSSL key generation, `/dev/urandom` seeding, entropy sources.

http://factorable.net

Are random-number generators really this bad?

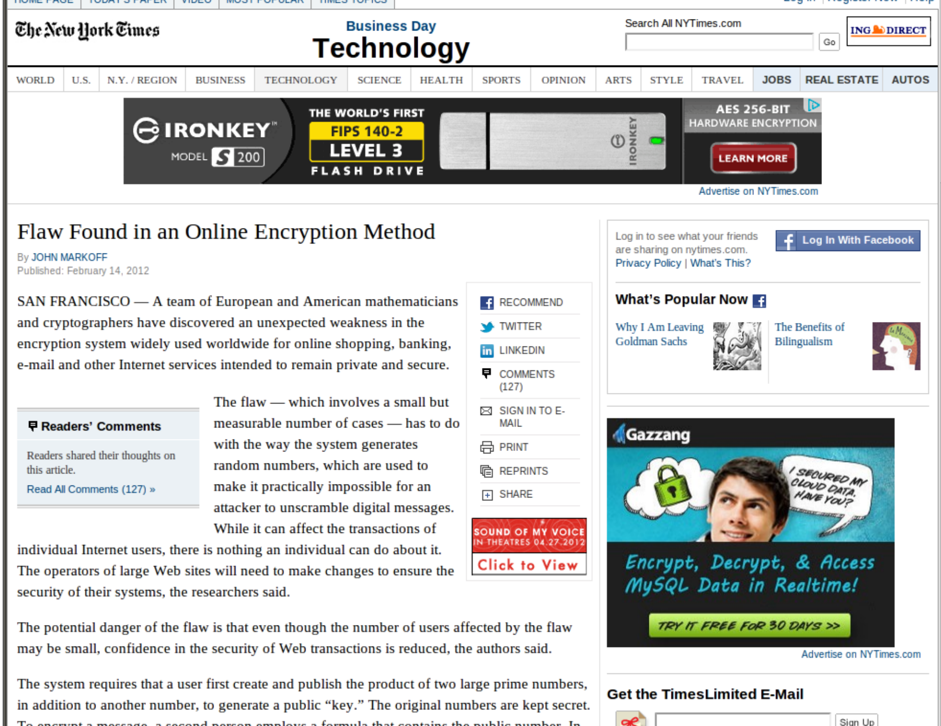
2012 Heninger–Durumeric–Wustrow–Halderman,
best-paper award at USENIX Security Symposium:

Factored tens of thousands of public keys on the Internet
. . . typically keys for your home router, not for your bank.
Why? **Many deployed devices are generating guessable p's.**
Most common problem: horrifyingly bad interactions between
OpenSSL key generation, `/dev/urandom` seeding, entropy sources.

http://factorable.net

2012 Lenstra–Hughes–Augier–Bos–Kleinjung–Wachter,
independent "Ron was wrong, Whit is right" paper, Crypto:

Factored tens of thousands of public keys on the Internet.
Dunno why, but OMG! Insecure e-commerce! Call the NYTimes!



HOME PAGE | TODAY'S PAPER | VIDEO | MOST POPULAR | TIMES TOPICS | Log In | Register Now | Help

The New York Times

Business Day
Technology

Search All NYTimes.com
Go

ING DIRECT

WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION | ARTS | STYLE | TRAVEL | JOBS | REAL ESTATE | AUTOS

IRONKEY MODEL S 200 THE WORLD'S FIRST FIPS 140-2 LEVEL 3 FLASH DRIVE AES 256-BIT HARDWARE ENCRYPTION LEARN MORE

Advertise on NYTimes.com

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

SOUND OF MY VOICE IN THEATRES 04.27.2012 Click to View

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

Gazzang
I SECURED MY CLOUD DATA. HAVE YOU?
Encrypt, Decrypt, & Access MySQL Data in Realtime!
TRY IT FREE FOR 30 DAYS >>

Advertise on NYTimes.com

Get the TimesLimited E-Mail
Sign Up

This is just the tip of the iceberg

Look for more examples of bad randomness!

This is just the tip of the iceberg

Look for more examples of bad randomness!

e.g., followup work by 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

Look for more examples of bad randomness!

e.g., followup work by 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."

Danger: Your prime is too small

$N = 1701411834604692317316873037158841057535$

Danger: Your prime is too small

$N = 170141183460469231731687303715884105753$5

is obviously divisible by 5.

Computers can test quickly for divisibility by a precomputed set of primes (using % or gcd with product).

Takes time about $p/\log(p)$ to find p.

Danger: Your prime is too small

$N = 1701411834604692317316873037158841057535$
is obviously divisible by 5.
Computers can test quickly for divisibility by a precomputed set of primes (using % or gcd with product).
Takes time about $p/\log(p)$ to find p.

Pollard rho

```
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)
```

Danger: Your prime is too small

$N = 1701411834604692317316873037158841057535$
is obviously divisible by 5.
Computers can test quickly for divisibility by a precomputed set of primes (using % or gcd with product).
Takes time about $p/\log(p)$ to find p.

Pollard rho

```
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 p or q divides `a1`− `a2`;
typically about $\sqrt{p}$ steps, for p the smaller of the primes.

Pollard's $p - 1$ method

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

Pollard's $p - 1$ method

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

```
N=44426601460658291157725536008128017297890787
4637194279031281180366057
y=lcm(range(1,2^22)) #this takes a while ...
s=Integer(pow(2,y,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 is *smooth*).

Math ahead:

Pollard's $p-1$ method

```
N=44426601460658291157725536008128017297890787
4637194279031281180366057
y=lcm(range(1,2^22)) #this takes a while ...
s=Integer(pow(2,y,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 is *smooth*).

Math ahead: Avoiding such p helps against the $p-1$ method – but does not help against ECM (the *elliptic curve method*), which works if the number of points on a curve modulo p is smooth.

"Strong primes" are obsolete: fail to defend against ECM.

Danger: Generating p and q from a single search

Lesson from before: Avoid small prime disaster, choose p and q of same size.

Problem if one takes 'same size' too literally:

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

Danger: Generating p and q from a single search

Lesson from before: Avoid small prime disaster, choose p and q of same size.

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}.

Danger: Generating p and q from a single search

Lesson from before: Avoid small prime disaster, choose p and q of same size.

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 $q = N/p = 10^{170} + 63$.

Danger: Generating p and q from a single search

Lesson from before: Avoid small prime disaster, choose p and q of same size.

Problem if one takes 'same size' too literally:

$$
\begin{aligned}
N = {} & 1000000000000000000000000000000000000000000000000000000\\
& 0000000000000000000000000000000000000000000000000000000000\\
& 0000000000000000000000000000000000000000000000000000000029\\
& 9999999999999999999999999999999999999999999999999999999999\\
& 9999999999999999999999999999999999999999999999999999999999\\
& 999999999999999999999999999999999999999999999999997921.
\end{aligned}
$$

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=115792089237316195423570985008721211221144628
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
999999999999999999999999940' # 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=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
```

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!
```

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=1157920892373161954235709850087212112211446282
62713908746538761285902758367353
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
```

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  # 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=115792089237316195448679392282006640413199890
130332179010243714077028592474181
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)
'340282366920938463500268096066682468352.99999994715097470855635083681884221930'
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=115792089237316195448679392282006640413199890130332179010243714077028592474181
sage: sqrt(N).numerical_approx(256).str(no_sci=2)
'340282366920938463500268096066682468352.99999994715097470855635083681884221937'
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)
'340282366920938463500268096066682468352.99999994715097470855635083681884221930'
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)
'340282366920938463500268096066682468352.9999999471509747085563508368188422193'
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.

Danger: Your keys are too small

Okay: Generate random p between 2^{511} and 2^{512}.
Independently generate random q between 2^{511} and 2^{512}.

Your public key $N = pq$ is between 2^{1022} and 2^{1024}.

Danger: Your keys are too small

Okay: Generate random p between 2^{511} and 2^{512}.
Independently generate random q between 2^{511} and 2^{512}.

Your public key $N = pq$ is between 2^{1022} and 2^{1024}.

Conventional wisdom: *Any* 1024-bit key can be factored in

- $\approx 2^{120}$ operations by CFRAC (continued-fraction method); or
- $\approx 2^{110}$ operations by LS (linear sieve); or
- $\approx 2^{100}$ operations by QS (quadratic sieve); or
- $\approx 2^{80}$ operations by NFS (number-field sieve).

Feasible today for botnets and for large organizations.
Will become feasible for more attackers as chips become cheaper.

An example of the quadratic sieve

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

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

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

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

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

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

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

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

Math 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)]
```

Many details and speedups

Core strategies to implement `easyfactorizations`:

- Batch trial division: same as the tree idea from before.
- "Sieving": like the Sieve of Eratosthenes.
- rho, $p - 1$, ECM: **very small memory requirements**.
- "Early aborts": optimized combination of everything.

"Sieving needs tons of memory" $\rightarrow$ "True, but ECM doesn't."

Many details and speedups

Core strategies to implement `easyfactorizations`:

- Batch trial division: same as the tree idea from before.
- "Sieving": like the Sieve of Eratosthenes.
- rho, $p-1$, ECM: **very small memory requirements**.
- "Early aborts": optimized combination of everything.

"Sieving needs tons of memory" $\rightarrow$ "True, but ECM doesn't."

Many important improvements outside `easyfactorizations`:

- Fast linear algebra.
- Multiple lattices ("MPQS"): smaller differences.
- NFS: much smaller differences.
- Batch NFS: **factor many keys at once**.

"The attack is feasible but not worthwhile" $\rightarrow$ "Batch NFS."

So what does it mean?

Complicated NFS analysis and optimization. Latest estimates: Scanning $\approx 2^{70}$ differences will factor any 1024-bit key.

How expensive is this?

So what does it mean?

Complicated NFS analysis and optimization. Latest estimates:
Scanning $\approx 2^{70}$ differences will factor any 1024-bit key.

How expensive is this? **It's free!**

The Conficker/Downadup botnet broke into $\approx 2^{23}$ machines.
There are $\approx 2^{25}$ seconds in a year.
Scanning $\approx 2^{70}$ differences in a year means
scanning $\approx 2^{22}$ differences/second/machine.

For comparison, the successful RSA-768 factorization
scanned $>2^{24}$ differences/second/machine.

So what does it mean?

Complicated NFS analysis and optimization. Latest estimates: Scanning $\approx 2^{70}$ differences will factor any 1024-bit key.

How expensive is this? **It's free!**

The Conficker/Downadup botnet broke into $\approx 2^{23}$ machines.
There are $\approx 2^{25}$ seconds in a year.
Scanning $\approx 2^{70}$ differences in a year means
scanning $\approx 2^{22}$ differences/second/machine.

For comparison, the successful RSA-768 factorization scanned $>2^{24}$ differences/second/machine.

"Linear algebra needs a supercomputer" $\rightarrow$
"No, can distribute linear algebra across many machines."

"Linear algebra needs tons of memory" $\rightarrow$
"No, can trade linear-algebra size for number of differences."

On the importance of not being seen

A year of botnet computation would be noticed, maybe stopped.

On the importance of not being seen

A year of botnet computation would be noticed, maybe stopped.

Alternate plan: Use a private computer cluster.
e.g. NSA is building a 2^{26}-watt computer center in Bluffdale.
e.g. China has a supercomputer center in Tianjin.

On the importance of not being seen

A year of botnet computation would be noticed, maybe stopped.

Alternate plan: Use a private computer cluster.
e.g. NSA is building a 2^{26}-watt computer center in Bluffdale.
e.g. 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	

On the importance of not being seen

A year of botnet computation would be noticed, maybe stopped.

Alternate plan: Use a private computer cluster.
e.g. NSA is building a 2^{26}-watt computer center in Bluffdale.
e.g. 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	

On the importance of not being seen

A year of botnet computation would be noticed, maybe stopped.

Alternate plan: Use a private computer cluster.
e.g. NSA is building a 2^{26}-watt computer center in Bluffdale.
e.g. 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	

On the importance of not being seen

A year of botnet computation would be noticed, maybe stopped.

Alternate plan: Use a private computer cluster.
e.g. NSA is building a 2^{26}-watt computer center in Bluffdale.
e.g. 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

On the importance of not being seen

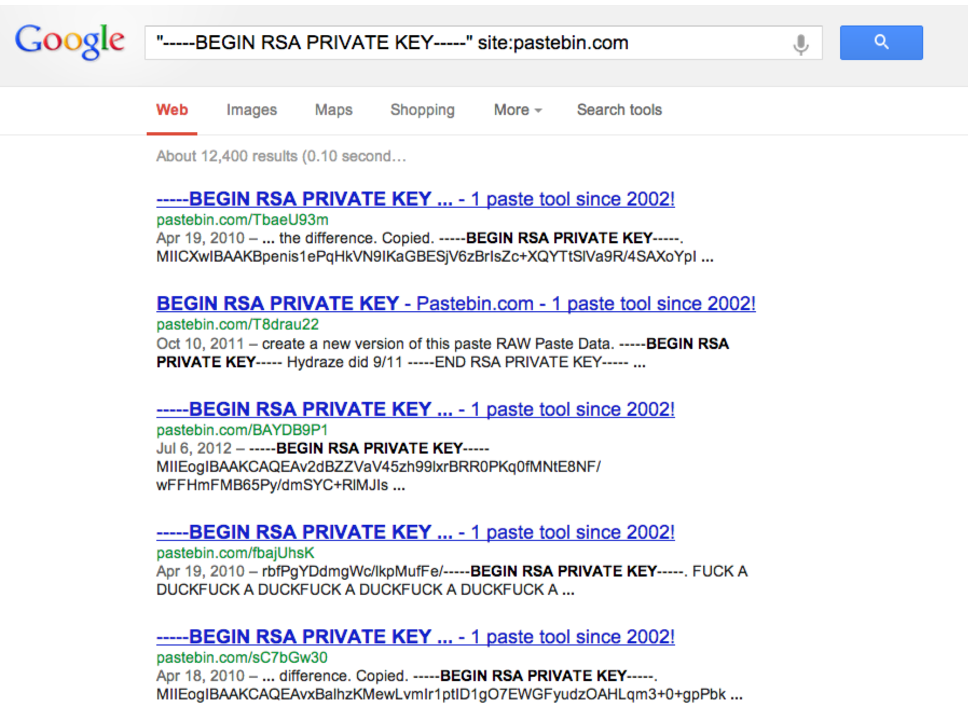
A year of botnet computation would be noticed, maybe stopped.

Alternate plan: Use a private computer cluster.
e.g. NSA is building a 2^{26}-watt computer center in Bluffdale.
e.g. 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.
Current estimates: This is enough to break 1024-bit RSA.
. . . and special-purpose chips should be at least $10\times$ faster.

Factoring keys with Google.



"-----BEGIN RSA PRIVATE KEY-----" site:pastebin.com

About 12,400 results (0.10 second...

-----**BEGIN RSA PRIVATE KEY** ... - 1 paste tool since 2002!
pastebin.com/TbaeU93m
Apr 19, 2010 – ... the difference. Copied. -----**BEGIN RSA PRIVATE KEY**-----.
MIICXwIBAAKBpenis1ePqHkVN9IKaGBESjV6zBrIsZc+XQYTtSIVa9R/4SAXoYpI ...

BEGIN RSA PRIVATE KEY - Pastebin.com - 1 paste tool since 2002!
pastebin.com/T8drau22
Oct 10, 2011 – create a new version of this paste RAW Paste Data. -----**BEGIN RSA PRIVATE KEY**----- Hydraze did 9/11 -----END RSA PRIVATE KEY----- ...

-----**BEGIN RSA PRIVATE KEY** ... - 1 paste tool since 2002!
pastebin.com/BAYDB9P1
Jul 6, 2012 – -----**BEGIN RSA PRIVATE KEY**-----
MIIEogIBAAKCAQEAv2dBZZVaV45zh99lxrBRR0PKq0fMNtE8NF/
wFFHmFMB65Py/dmSYC+RlMJls ...

-----**BEGIN RSA PRIVATE KEY** ... - 1 paste tool since 2002!
pastebin.com/fbajUhsK
Apr 19, 2010 – rbfPgYDdmgWc/lkpMufFe/-----**BEGIN RSA PRIVATE KEY**-----. FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A ...

-----**BEGIN RSA PRIVATE KEY** ... - 1 paste tool since 2002!
pastebin.com/sC7bGw30
Apr 18, 2010 – ... difference. Copied. -----**BEGIN RSA PRIVATE KEY**-----.
MIIEogIBAAKCAQEAvxBalhzKMewLvmIr1ptID1gO7EWGFyudzOAHLqm3+0+gpPbk ...

```
-----BEGIN RSA PRIVATE KEY-----
MIICXwIBAAKBpenis1ePqHkVN9IKaGBESjV6zBrIsZc+XQYTtSlVa9R/4SAXoYpI
upNrIjkCLd6DLDqfTO429xLDmYO4Ojzox7xiNcSMlBn8+TqTjf3TqAJmIOpgQVhJ
vW9is3OteT7l2ynAyMYvGqwROliCToMc/lOltlhPIFixw2AKUdOM5W76dwIDAQAB
AoGBAKDl8vuA9zUn2lTDddujAzBRp8ZEoJTxw7BVdLpZtgLWLuqPcXroyTkvBJC/
rbfPgYDdmgWc/lkpMufFe/-----BEGIN RSA PRIVATE KEY-----
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
...
FUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK A DUCKFUCK
Psg1RMTRceI/z3d/3BiuDjiUiRICFqOXDscCQQDFea/ocg8VVLvH/6pn7oNTQfbx
tkqCSSne3XgjAM+eA6TXbIo49d+3gsM3U1mGHR9ZBMy0O68ijhIqM7/7nJtBAkEA
jmkwiP2Fy0tQ9heq4rx90ZfmixcWf/H6JldRy7kJ/qG6uDnPvH55mTRuGPpas044
7sJphlPEY8ofkwJj7K/ZKQJBAIc75HQi/Br1lRC4qPmF2vwYgwpyF9RbZWO56Eo7
ipgts4FLFajgogOD+JxkkT1CXtEv7MqM6ihSxGVBD6UHN7I=
-----END RSA PRIVATE KEY-----
```

Unfucking the duck

```
-----BEGIN RSA PRIVATE KEY-----
MIICXwIBAAKBpenis1ePqHkVN9IKaGBESjV6zBrIsZc+XQYTtSlVa9R/4SAXoYpI
upNrIjkCLd6DLDqfTO429xLDmYO4Ojzox7xiNcSMlBn8+TqTjf3TqAJmIOpgQVhJ
vW9is3OteT7l2ynAyMYvGqwR0liCToMc/lOltlhPIFixw2AKUd0M5W76dwIDAQAB
AoGBAKDl8vuA9zUn2lTDdduj AzBRp8ZEoJTxw7BVdLpZtgLWLuqPcXroyTkvBJC/
rbfPgYDdmgWc/lkpMufFe/

Psg1RMTRceI/z3d/3BiuDjiUiRICFqOXDscCQQDFea/ocg8VVLvH/6pn7oNTQfbx
tkqCSSne3XgjAM+eA6TXbIo49d+3gsM3U1mGHR9ZBMy0O68ijhIqM7/7nJtBAkEA
jmkwiP2Fy0tQ9heq4rx90ZfmixcWf/H6JldRy7kJ/qG6uDnPvH55mTRuGPpas044
7sJphlPEY8ofkwJj7K/ZKQJBAIc75HQi/Br1lRC4qPmF2vwYgwpyF9RbZWO56Eo7
ipgts4FLFajgogOD+JxkkT1CXtEv7MqM6ihSxGVBD6UHN7I=
-----END RSA PRIVATE KEY-----
```

Unfucking the duck

```
-----BEGIN RSA PRIVATE KEY-----
MIICXwIBAAKBpenis1ePqHkVN9IKaGBESjV6zBrIsZc+XQYTtSlVa9R/4SAXoYpI
upNrIjkCLd6DLDqfTO429xLDmYO4OjzoX7xiNcSMlBn8+TqTjf3TqAJmIOpgQVhJ
vW9is3OteT7l2ynAyMYvGqwR0liCToMc/lOltlhPIFixw2AKUdOM5W76dwIDAQAB
AoGBAKDl8vuA9zUn2lTDddujAzBRp8ZEoJTxw7BVdLpZtgLWLuqPcXroyTkvBJC/
rbfPgYDdmgWc/lkpMufFe/ <-----          oh noes!             -->

Psg1RMTRceI/z3d/3BiuDjiUiRICFqOXDscCQQDFea/ocg8VVLvH/6pn7oNTQfbx
tkqCSSne3XgjAM+eA6TXbIo49d+3gsM3U1mGHR9ZBMy0O68ijhIqM7/7nJtBAkEA
jmkwiP2Fy0tQ9heq4rx90ZfmixcWf/H6JldRy7kJ/qG6uDnPvH55mTRuGPpas044
7sJphlPEY8ofkwJj7K/ZKQJBAIc75HQi/Br1lRC4qPmF2vwYgwpyF9RbZWO56Eo7
ipgts4FLFajgogOD+JxkkT1CXtEv7MqM6ihSxGVBD6UHN7I=
-----END RSA PRIVATE KEY-----
```

PKCS #1: RSA Cryptography Standard

```
RSAPublicKey ::= SEQUENCE {
    modulus           INTEGER,  -- n
    publicExponent    INTEGER   -- e
}

RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER,  -- n
    publicExponent    INTEGER,  -- e
    privateExponent   INTEGER,  -- d
    prime1            INTEGER,  -- p
    prime2            INTEGER,  -- q
    exponent1         INTEGER,  -- d mod (p-1)
    exponent2         INTEGER,  -- d mod (q-1)
    coefficient       INTEGER,  -- (inverse of q) mod p
    otherPrimeInfos   OtherPrimeInfos OPTIONAL
}
```

Unfucking the duck

N

```
-----BEGIN RSA PRIVATE KEY-----
MIICXwIBAAKBpenis1ePqHkVN9IKaGBESjV6zBrIsZc+XQYTtSlVa9R/4SAXoYpI
upNrIjkCLd6DLDqfTO429xLDmYO4OjzOx7xiNcSMlBn8+TqTjf3TqAJmIOpgQVhJ
vW9is3OteT7l2ynAyMYvGqwROliCToMc/lOltlhPIFixw2AKUdOM5W76dwIDAQAB
AoGBAKDl8vuA9zUn2lTDddujAzBRp8ZEoJTxw7BVdLpZtgLWLuqPcXroyTkvBJC/
rbfPgYDdmgWc/lkpMufFe/
```

e

d

$d \bmod p-1$

q

```
Psg1RMTRceI/z3d/3BiuDjiUiRICFqOXDscCQQDFea/ocg8VVLvH/6pn7oNTQfbx
tkqCSSne3XgjAM+eA6TXbIo49d+3gsM3U1mGHR9ZBMyOO68ijhIqM7/7nJtBAkEA
jmkwiP2Fy0tQ9heq4rx9OZfmixcWf/H6JldRy7kJ/qG6uDnPvH55mTRuGPpasO44
7sJphlPEY8ofkwJj7K/ZKQJBAIc75HQi/Br1lRC4qPmF2vwYgwpyF9RbZWO56Eo7
ipgts4FLFajgogOD+JxkkT1CXtEv7MqM6ihSxGVBD6UHN7I=
-----END RSA PRIVATE KEY-----
```

$q^{-1} \bmod p$

$d \bmod q-1$

Easy-to-compute relations between private key fields

```
q = gcd(int(pow(2,e*dp-1,n))-1,n)

p = n/q

d = inverse_mod(e,(p-1)*(q-1))

...
```

Incomplete portions of a single piece of the key?

Possible with Coppersmith/Howgrave-Graham techniques; see example on web.

Huzzah!

```
-----BEGIN RSA PRIVATE KEY-----
MIICXwIBAAKBgQDET1ePqHkVN9IKaGBESjV6zBrIsZc+XQYTtSlVa9R/4SAXoYpI
upNrIjkCLd6DLDqfTO429xLDmYO4OjzOx7xiNcSMlBn8+TqTjf3TqAJmIOpgQVhJ
vW9is3OteT7l2ynAyMYvGqwR0liCToMc/lOltlhPIFixw2AKUdOM5W76dwIDAQAB
AoGBAKDl8vuA9zUn2lTDddujAzBRp8ZEoJTxw7BVdLpZtgLWLuqPcXroyTkvBJC/
rbfPgYDdmgWc/lkpMufFe/TC+KgIDlWo5OPm/cwcChaM9nEINbFF1dqoA5gVxv6g
yUWQNKVKerToh/L3OpbiApArfB2aiimXUDH0eiGev6i6h0ShAkEA/MCm4KwarMP9
gPy2V/9qlJ1mEgZXMjHG4nWBfgPQE+9Lq1+e6kMePpuFgAC5ZJC8an4PCOLU5QIV
XBUW2uLGOQJBAMbVClSWms3llVT5IjKFNLdz0ShSu0Fh5UzRpMkxtEGYsO5VKnb4
Psg1RMTRceI/z3d/3BiuDjiUiRICFqOXDscCQQDFea/ocg8VVLvH/6pn7oNTQfbx
tkqCSSne3XgjAM+eA6TXbIo49d+3gsM3U1mGHR9ZBMy0O68ijhIqM7/7nJtBAkEA
jmkwiP2Fy0tQ9heq4rx9OZfmixcWf/H6JldRy7kJ/qG6uDnPvH55mTRuGPpas044
7sJphlPEY8ofkwJj7K/ZKQJBAIc75HQi/Br1lRC4qPmF2vwYgwpyF9RbZWO56Eo7
ipgts4FLFajgogOD+JxkkT1CXtEv7MqM6ihSxGVBD6UHN7I=
-----END RSA PRIVATE KEY-----
```

Lessons

Lessons

- Stop using 1024-bit RSA, if you haven't already.

Lessons

- Stop using 1024-bit RSA, if you haven't already.
- Make sure your primes are big enough.

Lessons

- Stop using 1024-bit RSA, if you haven't already.
- Make sure your primes are big enough.
- Make sure your primes are random.

Lessons

- Stop using 1024-bit RSA, if you haven't already.
- Make sure your primes are big enough.
- Make sure your primes are random.
- "FUCK A DUCK" is not good crypto.

Lessons

- Stop using 1024-bit RSA, if you haven't already.
- Make sure your primes are big enough.
- Make sure your primes are random.
- "FUCK A DUCK" is not good crypto.
- Pastebin is not a secure cloud store.

Lessons

- Stop using 1024-bit RSA, if you haven't already.
- Make sure your primes are big enough.
- Make sure your primes are random.
- "FUCK A DUCK" is not good crypto.
- Pastebin is not a secure cloud store.
- Probably shouldn't put your private key in a "secure" cloud store anyway.

Lessons

- Stop using 1024-bit RSA, if you haven't already.
- Make sure your primes are big enough.
- Make sure your primes are random.
- "FUCK A DUCK" is not good crypto.
- Pastebin is not a secure cloud store.
- Probably shouldn't put your private key in a "secure" cloud store anyway.
- Probably shouldn't fuck a duck.

Instructions, explanations, examples, and code snippets available online at:

http://facthacks.cr.yp.to