Exploiting Poor Randomness: Lattice-Based Techniques

Nadia Heninger

University of Pennsylvania

October 16, 2014

Taiwan Citizen Digital Certificate

Analysis in (Bernstein, Chang, Cheng, Chou, Heninger, Lange, van Someren Asiacrypt 2013)

Many countries adopting national PKI.

Taiwan's smart card IDs allow citizens to

- file income taxes,
- update car registrations,
- transact with government agencies,
- interact with companies (e.g. Chunghwa Telecom) online.

Taiwan Citizen Digital Certificate

- Smart cards are issued by the government.
- FIPS-140 and Common Criteria Level 4+ certified.
- RSA keys are generated on card.
- Certificates stored on national LDAP directory. This is publicly accessible to enable citizen-to-citizen and citizen-to-commerce interactions.

Certificate of Chen-Mou Cheng

```
Data: Version: 3 (0x2)
Serial Number: d7:15:33:8e:79:a7:02:11:7d:4f:25:b5:47:e8:ad:38
Signature Algorithm: sha1WithRSAEncryption
Issuer: C=TW, O=XXX
Validity
     Not Before: Feb 24 03:20:49 2012 GMT
     Not After : Feb 24 03:20:49 2017 GMT
Subject: C=TW, CN=YYY serialNumber=0000000112831644
Subject Public Key Info:
     Public Key Algorithm: rsaEncryption
Public-Key: (2048 bit) Modulus:
     00:bf:e7:7c:28:1d:c8:78:a7:13:1f:cd:2b:f7:63:
     2c:89:0a:74:ab:62:c9:1d:7c:62:eb:e8:fc:51:89:
     b3:45:0e:a4:fa:b6:06:de:b3:24:c0:da:43:44:16:
     e5:21:cd:20:f0:58:34:2a:12:f9:89:62:75:e0:55:
     8c:6f:2b:0f:44:c2:06:6c:4c:93:cc:6f:98:e4:4e:
     3a:79:d9:91:87:45:cd:85:8c:33:7f:51:83:39:a6:
     9a:60:98:e5:4a:85:c1:d1:27:bb:1e:b2:b4:e3:86:
     a3:21:cc:4c:36:08:96:90:cb:f4:7e:01:12:16:25:
     90:f2:4d:e4:11:7d:13:17:44:cb:3e:49:4a:f8:a9:
     a0:72:fc:4a:58:0b:66:a0:27:e0:84:eb:3e:f3:5d:
     5f:b4:86:1e:d2:42:a3:0e:96:7c:75:43:6a:34:3d:
     6b:96:4d:ca:f0:de:f2:bf:5c:ac:f6:41:f5:e5:bc:
     fc:95:ee:b1:f9:c1:a8:6c:82:3a:dd:60:ba:24:a1:
     eb:32:54:f7:20:51:e7:c0:95:c2:ed:56:c8:03:31:
     96:c1:b6:6f:b7:4e:c4:18:8f:50:6a:86:1b:a5:99:
     d9:3f:ad:41:00:d4:2b:e4:e7:39:08:55:7a:ff:08:
     30:9e:df:9d:65:e5:0d:13:5c:8d:a6:f8:82:0c:61:
     c8:6b
Exponent: 65537 (0x10001)

        .
        .
        .
```

A nice student project...

April 2012: Downloaded all certificates from LDAP server:

- 2,300,000 1024-bit RSA public keys
- 740,000 2048-bit RSA public keys (issued since 2010)

A nice student project...

April 2012: Downloaded all certificates from LDAP server:

- 2,300,000 1024-bit RSA public keys
- 740,000 2048-bit RSA public keys (issued since 2010)

HITCON 2012 (July 20–21):
Prof. Li-Ping Chou presents “Cryptanalysis in real life”
(based on work with Yun-An Chang and Chen-Mou Cheng)

- Factored 103 RSA-1024 public keys with GCD algorithm

A nice student project...

April 2012: Downloaded all certificates from LDAP server:

- 2,300,000 1024-bit RSA public keys
- 740,000 2048-bit RSA public keys (issued since 2010)

HITCON 2012 (July 20–21):
Prof. Li-Ping Chou presents “Cryptanalysis in real life”
(based on work with Yun-An Chang and Chen-Mou Cheng)

- Factored 103 RSA-1024 public keys with GCD algorithm
- Wrote report that some keys are factored, informed MOI.

A nice student project...

April 2012: Downloaded all certificates from LDAP server:

- 2,300,000 1024-bit RSA public keys
- 740,000 2048-bit RSA public keys (issued since 2010)

HITCON 2012 (July 20–21):
Prof. Li-Ping Chou presents “Cryptanalysis in real life”
(based on work with Yun-An Chang and Chen-Mou Cheng)

- Factored 103 RSA-1024 public keys with GCD algorithm
- Wrote report that some keys are factored, informed MOI.
- MOI promised to replace cards of affected users.

Investigating the factors...

Most common factor appears 46 times

```
c000000000000000000000000000000
000000000000000000000000000000
000000000000000000000000000000
0000000000000000000000000002f9
```

Investigating the factors...

Most common factor appears 46 times

```
c000000000000000000000000000000
000000000000000000000000000000
000000000000000000000000000000
0000000000000000000000000002f9
```

which is the next prime after $2^{511} + 2^{510}$.

Investigating the factors...

Most common factor appears 46 times

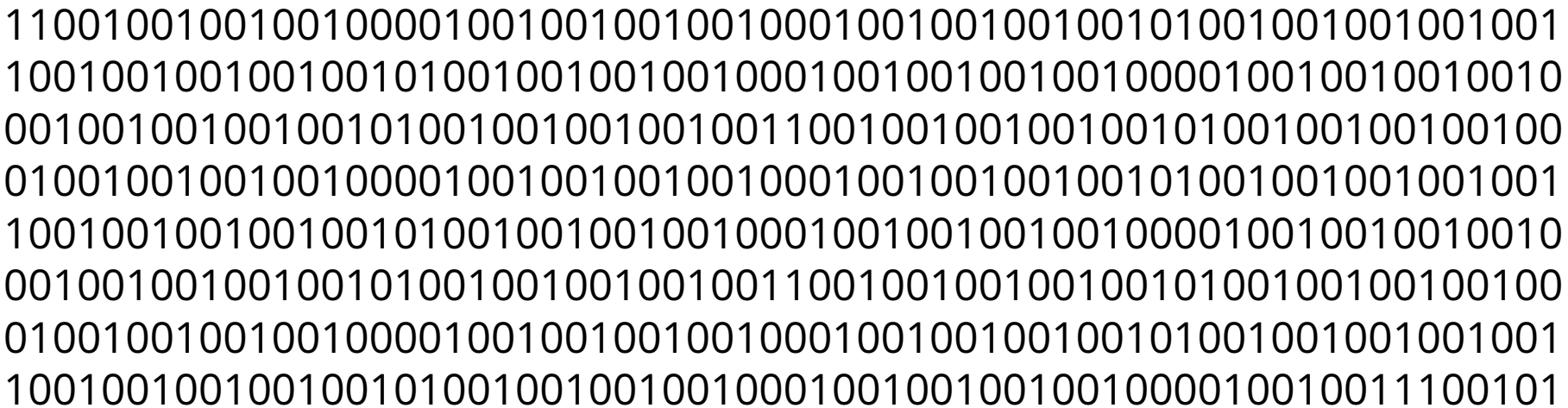
```
c000000000000000000000000000000
000000000000000000000000000000
000000000000000000000000000000
0000000000000000000000000002f9
```

which is the next prime after $2^{511} + 2^{510}$.
The next most common factor, repeated 7 times, is

```
c92424922492924992492449242492
249292499249492449242492249292 49
924949244924249224992492494924
49242492249292499249244924 24e5
```

Several other factors exhibit such a pattern.

How is this pattern generated?



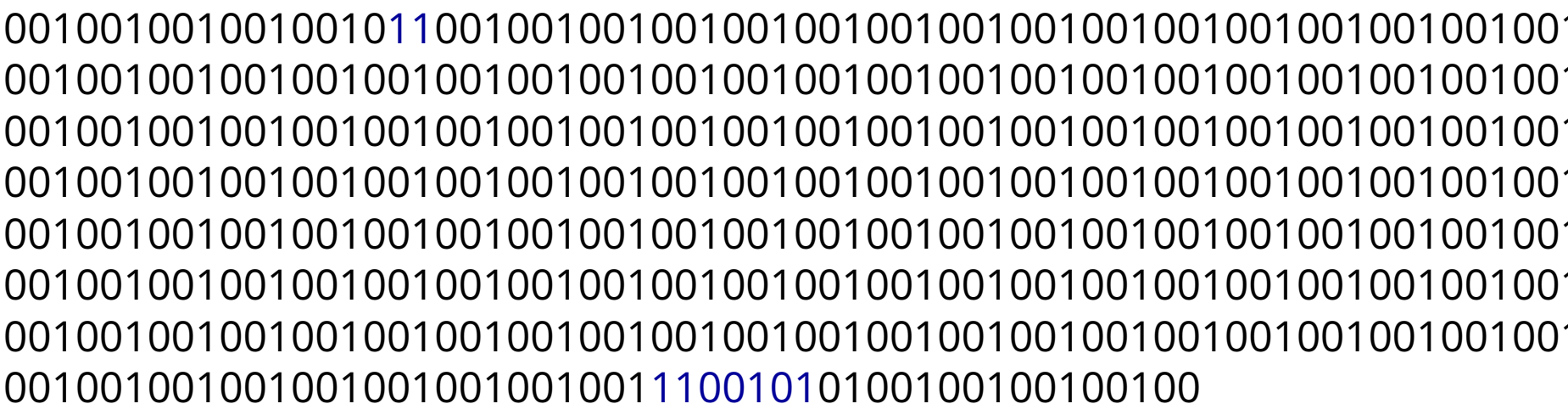
How is this pattern generated?

Swap every 16 bits in a 32 bit word

00100100100100101100100100100100 10010010010010010010010010010010
01001001001001001001001001001001 00100100100100100100100100100100
10010010010010010010010010010010 01001001001001001001001001001001
00100100100100100100100100100100 10010010010010010010010010010010
01001001001001001001001001001001 00100100100100100100100100100100
10010010010010010010010010010010 01001001001001001001001001001001
00100100100100100100100100100100 10010010010010010010010011001010
01001001001001001001001001001001 00100100100100100100100100100100

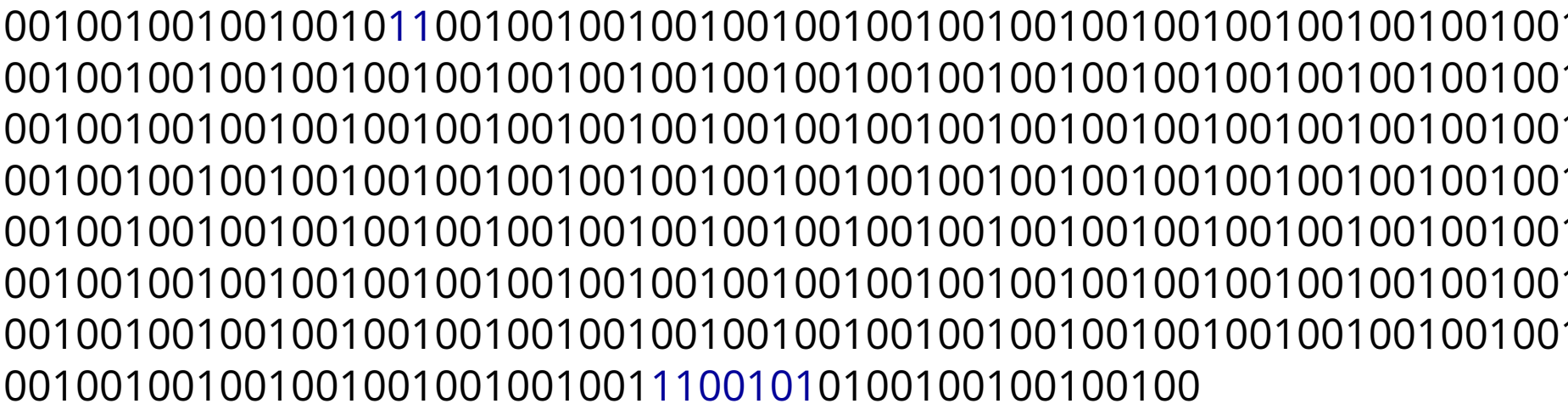
How is this pattern generated?

Realign



How is this pattern generated?

Realign



The 119 factors had patterns of period 1, 3, 5, and 7.

Shared prime generation

Hypothesized key generation process for weak primes:

1. Choose a bit pattern of length 1, 3, 5, or 7 bits.
2. Repeat it to cover 512 bits.
3. For every 32-bit word, swap the lower and upper 16 bits.
4. Fix the most significant two bits to 11.
5. Find the next prime greater than or equal to this number.

Shared prime generation

Hypothesized key generation process for weak primes:

1. Choose a bit pattern of length 1, 3, 5, or 7 bits.
2. Repeat it to cover 512 bits.
3. For every 32-bit word, swap the lower and upper 16 bits.
4. Fix the most significant two bits to 11.
5. Find the next prime greater than or equal to this number.

Factoring by trial division

Enumerating all patterns of this form factored 18 more keys.

Extending to patterns of length 9 gave us 4 more keys.

Some more prime factors

```
c0000000000000000000000000000000
0000000000000000000000000000000000
0000000000000000000000000000000000
00000000000000000000000000101ff

c0000000000000000000000000000000
0000000000000000000000000000000000
0000000000000000000000000000000000
00000000000000000000100000177
```

Some more prime factors

```
c0000000000000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
000000000000000000000000101ff

c0000000000000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
00000000000000000000100000177
```

Hypothesis: There might be more prime factors of the form

$$p = 2^{511} + 2^{510} + x$$

where x is “small”.

What's wrong with this RSA example?

```
message = Integer('squeamishossifrage',base=35)
N = random_prime(2^512)*random_prime(2^512)
c = message^3 % N
```

What's wrong with this RSA example?

```
message = Integer('squeamishossifrage',base=35)
N = random_prime(2^512)*random_prime(2^512)
c = message^3 % N

sage: Integer(c^(1/3)).str(base=35)
'squeamishossifrage'
```

What's wrong with this RSA example?

```
message = Integer('squeamishossifrage',base=35)
N = random_prime(2^512)*random_prime(2^512)
c = message^3 % N

sage: Integer(c^(1/3)).str(base=35)
'squeamishossifrage'
```

The message is too small.

This is why we use padding.

```
N = random_prime(2^128)*random_prime(2^128)
message = Integer('thepasswordfortodayisswordfish',base=35)
c = message^3 % N
```

```
N = random_prime(2^128)*random_prime(2^128)
message = Integer('thepasswordfortodayisswordfish',base=35)
c = message^3 % N

sage: int(c^(1/3))==message
False
```

```
N = random_prime(2^128)*random_prime(2^128)
message = Integer('thepasswordfortodayisswordfish',base=35)
c = message^3 % N
```

This is a stereotyped message. We might be able to guess the format.

```
N = random_prime(2^128)*random_prime(2^128)
message = Integer('thepasswordfortodayisswordfish',base=35)
c = message^3 % N


a = Integer('thepasswordfortodayisxxxxxxxxxx',base=35)
```

```
N = random_prime(2^128)*random_prime(2^128)
message = Integer('thepasswordfortodayisswordfish',base=35)
c = message^3 % N


a = Integer('thepasswordfortodayisxxxxxxxxxx',base=35)

X = Integer('xxxxxxxxxx',base=35)
M = matrix(7)
```

M is coefficient vectors of polynomials
$N^2, N^2xX, N^2(xX)^2, N((a+xX)^3-c), \ldots, ((a+xX)^3-c)^2$.

```
N = random_prime(2^128)*random_prime(2^128)
message = Integer('thepasswordfortodayisswordfish',base=35)
c = message^3 % N


a = Integer('thepasswordfortodayisxxxxxxxxxx',base=35)

X = Integer('xxxxxxxxxx',base=35)
M = matrix(7)
```

M is coefficient vectors of polynomials
$N^2, N^2xX, N^2(xX)^2, N((a+xX)^3-c), \ldots, ((a+xX)^3-c)^2$.

```
B = M.LLL()
f = sum(B[0][i]*(x/X)^i for i in range(7))

sage: f.factor()[0]
(x + 11340606574691, 1)
```

```
N = random_prime(2^128)*random_prime(2^128)
message = Integer('thepasswordfortodayisswordfish',base=35)
c = message^3 % N


a = Integer('thepasswordfortodayisxxxxxxxxxx',base=35)

X = Integer('xxxxxxxxxx',base=35)
M = matrix(7)
```

M is coefficient vectors of polynomials
$N^2, N^2xX, N^2(xX)^2, N((a+xX)^3-c), \ldots, ((a+xX)^3-c)^2$.

```
B = M.LLL()
f = sum(B[0][i]*(x/X)^i for i in range(7))

sage: f.factor()[0]
(x + 11340606574691, 1)

sage: (a-11340606574691).str(base=35)
'thepasswordfortodayisswordfish'
```

What's going on here? Coppersmith's method.

Theorem (Coppersmith)

We can efficiently compute up to $1/e$*-fraction of the bits of an RSA-encrypted message with public exponent* e *if we know the rest of the plaintext.*

Theorem (Coppersmith)

Given a polynomial f *of degree* d *and* N*, we can efficiently find all roots* r_i *satisfying*

$$f(r_i) \equiv 0 \bmod N$$

when $|r_i| < N^{1/d}$.

The stereotyped message problem gives us an a satisfying

$$(a + x)^3 - c \equiv 0 \bmod N$$

Why is this an interesting theorem?

1. A general method to solve polynomials mod N would break RSA:

$$x^e - c \equiv 0 \bmod N$$

2. There is an efficient algorithm to solve equations mod primes.
   - For a composite, factor into primes, solve mod each prime, and use Chinese remainder theorem to lift solution mod N.

3. By accepting a bound on solution size, Coppersmith's method lets us solve equations without factoring N.

Coppersmith's Algorithm Outline

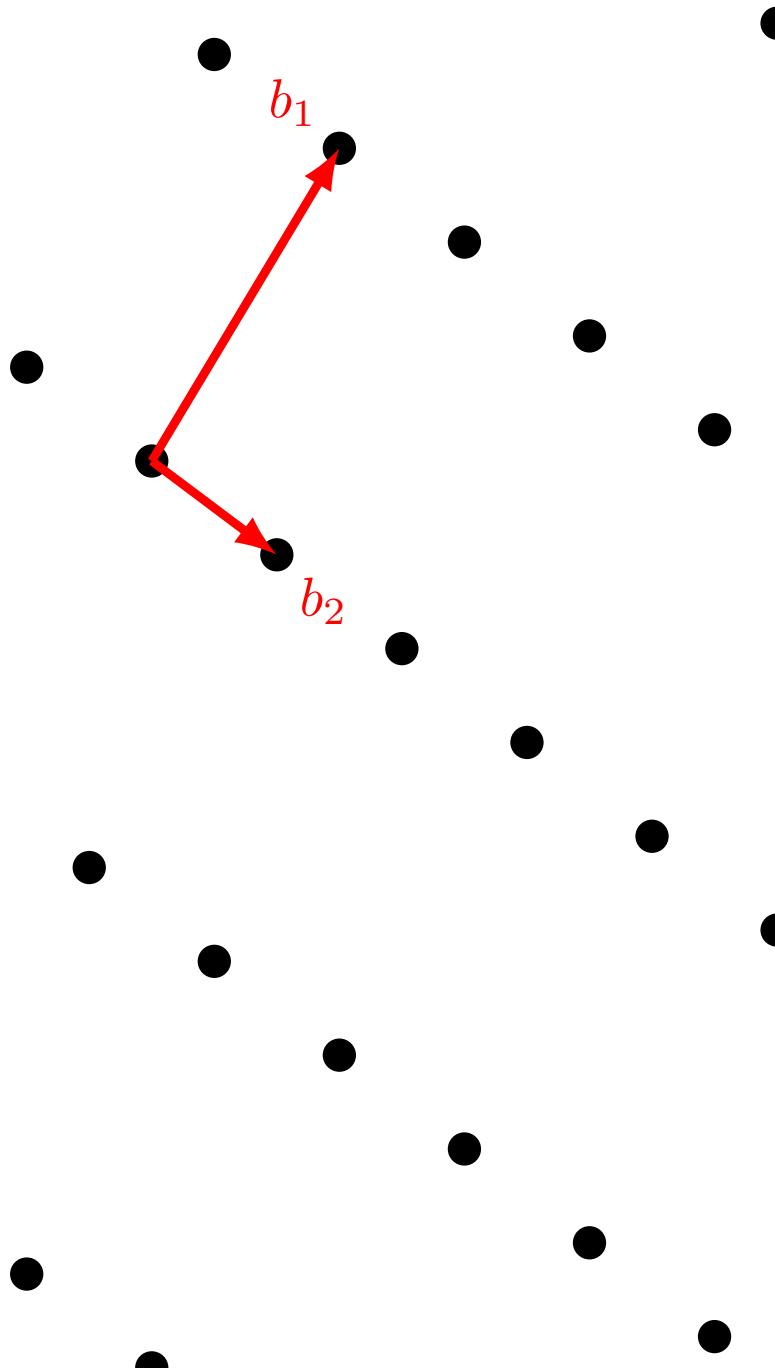
Input: polynomial f, modulus N

1. Construct a matrix of coefficient vectors of powers of f and N:
$$N^k, xN^k, \ldots, N^{k-1}f, \ldots, f^k, xf^k, \ldots$$
2. Run a lattice basis reduction algorithm on this matrix.
3. Construct a polynomial Q from the shortest vector output.
4. Factor Q to find its roots.

What is a lattice?

Definition

A **lattice** is a set of points in space generated by integer linear combinations of some basis vectors $\{b_1, \ldots, b_n\}$.



Theorem (LLL)

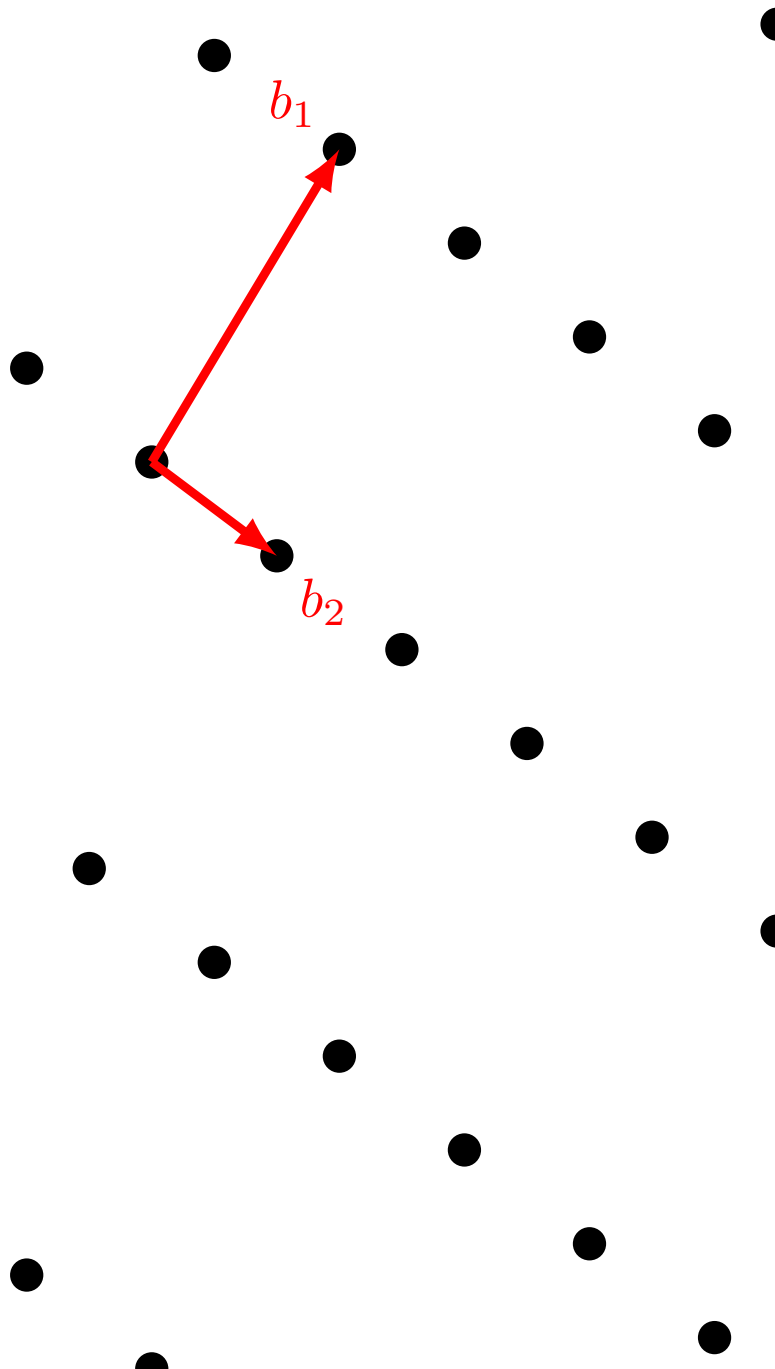
We can find a vector of length

$$|v| < 2^{\dim L} (\det L)^{1/\dim L}$$

Lattices in practice

- We use LLL as a black box.
- In practice, LLL finds vectors of length $1.02^{\dim L}(\det L)^{1/\dim L}$ for random inputs.

  Open problem: Explain this behavior. (Nguyen, Stehle 2006)
- All our lattices are small dimension, so ignore approximation factor.
- All our matrices are diagonal, so $\det L$ is product of diagonal entries.



```
p = random_prime(2^512); q = random_prime(2^512)
N = p*q

a = p - (p % 2^86)
```

```
p = random_prime(2^512); q = random_prime(2^512)
N = p*q

a = p - (p % 2^86)


sage: hex(a)
'a9759e8c9fba8c0ec3e637d1e26e7b88befeb03ac199d1190
76e3294d16ffcaef629e2937a03592895b29b0ac708e79830
4330240bc00000000000000000000'
```

Key recovery from partial information.

```
p = random_prime(2^512); q = random_prime(2^512)
N = p*q

a = p - (p % 2^86)

X = 2^86
M = matrix([[X^2, 2*X*a, a^2], [0, X, a], [0, 0, N]])
B = M.LLL()
```

```
p = random_prime(2^512); q = random_prime(2^512)
N = p*q

a = p - (p % 2^86)

X = 2^86
M = matrix([[X^2, 2*X*a, a^2], [0, X, a], [0, 0, N]])
B = M.LLL()

f = B[0][0]*x^2/X^2+B[0][1]*x/X+B[0][2]

sage: f.factor()[0]
(x - 27753385000165998643777709, 1)

sage: a+27753385000165998643777709 == p
True
```

Partial key recovery and finding solutions modulo divisors

Theorem (Coppersmith)

Given half the bits (most or least significant) of p, we can factor N in polynomial time.

Theorem (Howgrave-Graham)

Given degree d polynomial f, integer N, we can find roots r modulo divisors B of N satisfying

$$f(r) \equiv 0 \bmod B$$

for $|B| > N^{\beta}$, *when* $|r| < N^{\beta^2/d}$.

For RSA partial key recovery, we have

$$f(x) = a + x$$

and we want to find a solution vanishing modulo $p \approx N^{1/2}$.

Partial key recovery and related attacks

RSA particularly susceptible to partial key recovery attacks.

- 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 $d \bmod (p-1)$. [Blömer May 03]

Factoring Taiwanese keys from partial information

Previously: not clear what kind of natural attack would reveal half the bits of a factor.

Taiwanese cards: We observed RNG getting "stuck". What if RNG becomes unstuck in least significant bits?

Exactly scenario from first example:

- A 3-dimensional lattice can let us find errors as big as $N^{1/6}$.

- Ran 3-dimensional lattice with every pattern against every key (1 hour per pattern, 164 patterns).
- Factored 39 new keys (and all but 2 of keys factored via GCD).

```
p = random_prime(2^512); q = random_prime(2^512)
N = p*q

d = random_prime(floor(N^(1/4))/2)
e = inverse_mod(d,(p-1)*(q-1))
```

d is relatively small. (But not that small.)

```
p = random_prime(2^512); q = random_prime(2^512)
N = p*q

d = random_prime(floor(N^(1/4))/2)
e = inverse_mod(d,(p-1)*(q-1))

X = ceil(N^(1/4)/2); Y = ceil(N^(1/2))
M = matrix([[X*Y, -X*(N+1), -1], [0, e*X, 0], [0,0,e]])

B = M.LLL()
```

```
p = random_prime(2^512); q = random_prime(2^512)
N = p*q

d = random_prime(floor(N^(1/4))/2)
e = inverse_mod(d,(p-1)*(q-1))

X = ceil(N^(1/4)/2); Y = ceil(N^(1/2))
M = matrix([[X*Y, -X*(N+1), -1], [0, e*X, 0], [0,0,e]])

B = M.LLL()

sage: B[0][0]/(X*Y) == (e*d-1)/((p-1)*(q-1))
True
sage: k = B[0][0]/(X*Y)
```

```
p = random_prime(2^512); q = random_prime(2^512)
N = p*q

d = random_prime(floor(N^(1/4))/2)
e = inverse_mod(d,(p-1)*(q-1))

X = ceil(N^(1/4)/2); Y = ceil(N^(1/2))
M = matrix([[X*Y, -X*(N+1), -1], [0, e*X, 0], [0,0,e]])

B = M.LLL()

sage: B[0][0]/(X*Y) == (e*d-1)/((p-1)*(q-1))
True
sage: k = B[0][0]/(X*Y)
sage: p+q == (1-B[0][1]/X)/k
True
sage: s = (1-B[0][1]/X)/k
```

```
p = random_prime(2^512); q = random_prime(2^512)
N = p*q

d = random_prime(floor(N^(1/4))/2)
e = inverse_mod(d,(p-1)*(q-1))

X = ceil(N^(1/4)/2); Y = ceil(N^(1/2))
M = matrix([[X*Y, -X*(N+1), -1], [0, e*X, 0], [0,0,e]])

B = M.LLL()

sage: B[0][0]/(X*Y) == (e*d-1)/((p-1)*(q-1))
True
sage: k = B[0][0]/(X*Y)
sage: p+q == (1-B[0][1]/X)/k
True
sage: s = (1-B[0][1]/X)/k
sage: (1+k*(N-s+1))/e == d
True
```

Small RSA private exponent: Bivariate Coppersmith

Theorem (Wiener)

We can efficiently compute d when $d < N^{1/4}$.

Theorem (Boneh Durfee)

We can efficiently compute d when $d < N^{0.292}$.

The *RSA equation* is

$$ed \equiv 1 \bmod (p-1)(q-1)$$
$$ed = 1 + k(N - (p+q) + 1)$$

Boneh and Durfee use Coppersmith's method to find small solutions $x = k, y = (p+q)$ to

$$xy - (N+1)x - 1 \equiv 0 \bmod e$$

Multivariate Coppersmith

Scenario: Given multivariate polynomial $f(x_1, \ldots, x_m)$ and wish to find roots

$$f(r_1, \ldots, r_m) \equiv 0 \bmod N$$

Same approach works in this case, with some tweaks:

- To find solutions we solve a system of m equations taken from the short vectors in our lattice.
- Theorems are generally heuristic because we don't know solution set is finite.
- Results are more ad hoc in general.

Open problem: Give a useful characterization of when multivariate Coppersmith's method produces algebraically independent equations.

```
ffffaa55fffffffff3cd9fe3ffff676
fffffffffffe0000000000000000000
00000000000000000000000000000000
0000000000000000000000000000009d

c000b80000000000000000000000000
00000000000000000000000000000000
00000680000000000000000000000000
0000000000000000000000000000251
```

Factoring Taiwanese keys with bivariate Coppersmith

Returning to Taiwan example. What if RNG becomes stuck after most significant bits?

Want to find solutions to the equation

$$a + 2^t x + y \equiv 0 \bmod p$$

This lets us factor keys with pattern errors in most, least, or middle bits by setting t.

Ran on 20 most common patterns and factored 13 more keys.

Factoring with Bivariate Coppersmith

Search for prime factors of the form

$$p = a + 2^t x + y$$

Factoring with Bivariate Coppersmith

Search for prime factors of the form

$$p = a + 2^t x + y$$

Algorithm (Expected Algorithm)

1. *Generate lattice from multiples of* $f(x,y) = a + 2^t x + y$, N.
2. *Run LLL and take two short polynomials* $Q_1(x,y)$, $Q_2(x,y)$.
3. *Solve for* r_1, r_2 *satisfying* $Q_1(r_1,r_2) = Q_2(r_1,r_2) = 0$.
4. *Check if* $\gcd(a + 2^t r_1 + r_2, N)$ *is nontrivial.*

Factoring with Bivariate Coppersmith

Search for prime factors of the form

$$p = a + 2^t x + y$$

Algorithm (Expected Algorithm)

1. *Generate lattice from multiples of* $f(x,y) = a + 2^t x + y$, N.
2. *Run LLL and take two short polynomials* $Q_1(x,y)$, $Q_2(x,y)$.
3. *Solve for* r_1, r_2 *satisfying* $Q_1(r_1, r_2) = Q_2(r_1, r_2) = 0$.
4. *Check if* $\gcd(a + 2^t r_1 + r_2, N)$ *is nontrivial.*

- Analysis says 10-dimensional lattices let us solve for

$$|r_1 r_2| < N^{1/10}.$$

- For 1024-bit N, should have $|r_1 r_2| < 2^{102}$.

Tricky Details: Algebraic Dependence

- Need *two* equations $Q_1(x, y)$, $Q_2(x, y)$.
- Coefficient vectors in lattice are linearly independent, but polynomials might have algebraic relation.

Tricky Details: Algebraic Dependence

- Need *two* equations $Q_1(x, y)$, $Q_2(x, y)$.
- Coefficient vectors in lattice are linearly independent, but polynomials might have algebraic relation.

Assumption

The short vectors of the LLL-reduced basis correspond to algebraically independent polynomials.

Tricky Details: Algebraic Dependence

- Need *two* equations $Q_1(x, y)$, $Q_2(x, y)$.
- Coefficient vectors in lattice are linearly independent, but polynomials might have algebraic relation.

Assumption

The short vectors of the LLL-reduced basis correspond to algebraically independent polynomials.

This assumption **failed** in our experiments.

Tricky Details: Algebraic Dependence

- Need *two* equations $Q_1(x, y)$, $Q_2(x, y)$.
- Coefficient vectors in lattice are linearly independent, but polynomials might have algebraic relation.

Assumption

The short vectors of the LLL reduced basis correspond to algebraically independent polynomials.

This assumption **failed** in our experiments.

- In most cases polynomials shared linear common factors

$$q_1 x + q_2 y + q_3 = 0$$

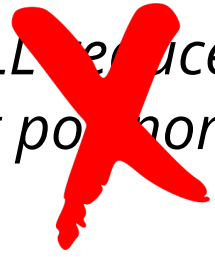
and thus had infinitely many potential solutions.

Tricky Details: Algebraic Dependence

- Need *two* equations $Q_1(x, y)$, $Q_2(x, y)$.
- Coefficient vectors in lattice are linearly independent, but polynomials might have algebraic relation.

Assumption

The short vectors of the LLL reduced basis correspond to algebraically independent polynomials.



This assumption **failed** in our experiments.

- In most cases polynomials shared linear common factors

$$q_1 x + q_2 y + q_3 = 0$$

  and thus had infinitely many potential solutions.
- By experimenting, we learned that the *smallest* solution seemed to work.

Tricky Details: Theory vs. Practice

Solution Sizes

- Standard analysis told us algorithm should work with lattice dimension ≥ 10.
- But in practice lattice dimension 6 worked!

Patterns

- When we experimented with pattern

  $$x000\ldots000y$$

  method also found factors of form

  $$x9924\ldots4929y$$

  and other repeating patterns!

Experimental Results

dim	XY	offsets	patterns	keys factored	running time
6	2^4	5	1	104	4.3 hours
6	2^4	1	164	154	195 hours
10	2^{100}	1	1	112	2 hours
15	2^{128}	5	1	108	20 hours

11 additional keys factored.

Bivariate Coppersmith details

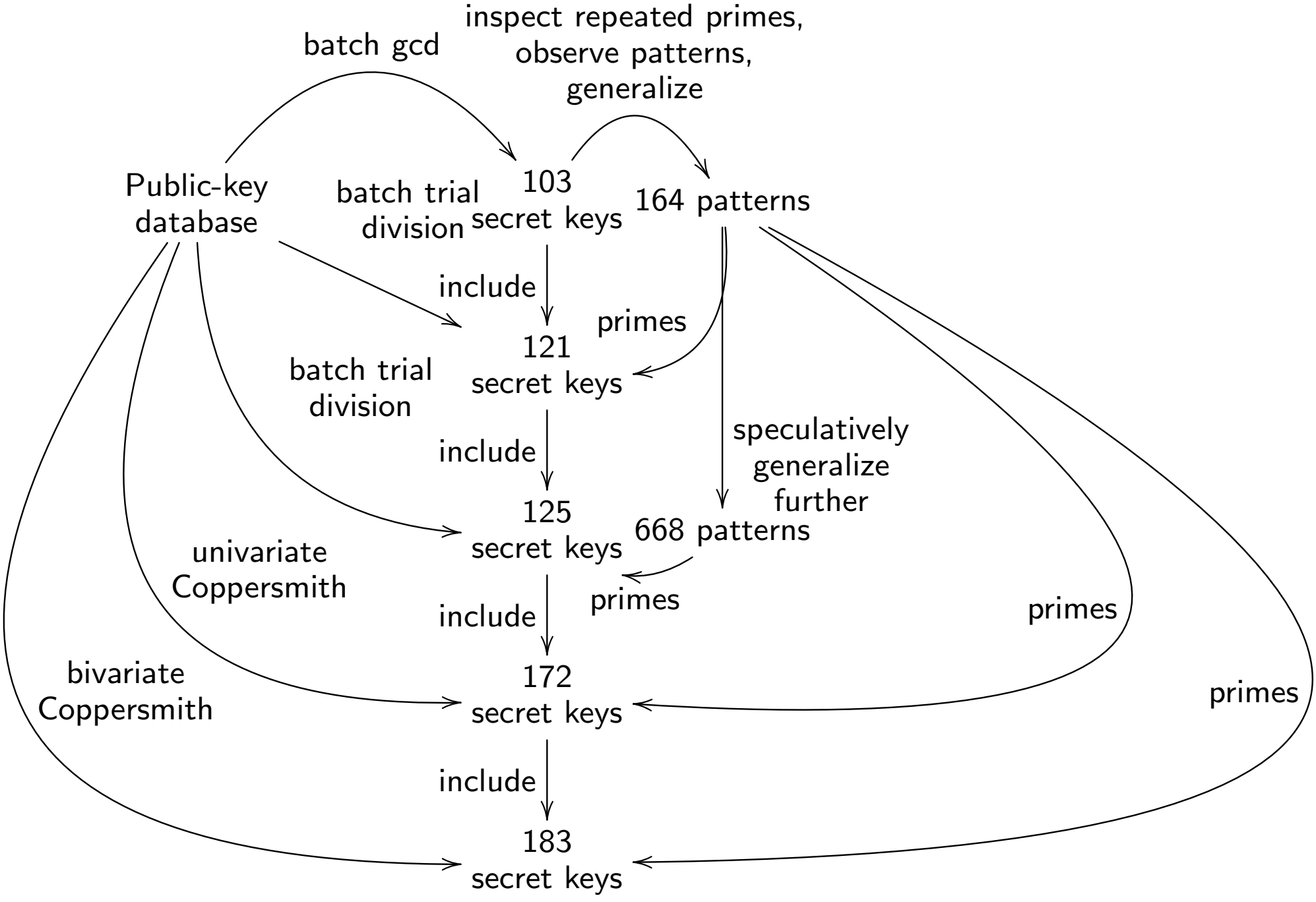
A mystery: The attack works *much* better in practice than theory guarantees.

Smallest lattice with guaranteed solution has dimension 10. But dimension 6 worked too.

Open problem: Why?

For finding solution, equations were not algebraically independent, but could find a solution anyway.

Open problem: Why?


batch gcd
inspect repeated primes,
observe patterns,
generalize
Public-key
database
batch trial
division
103
secret keys
164 patterns
include
primes
121
secret keys
batch trial
division
include
speculatively
generalize
further
125
secret keys
668 patterns
univariate
Coppersmith
primes
include
primes
bivariate
Coppersmith
172
secret keys
primes
include
183
secret keys

Why are government-certified smartcards generating weak keys?

Best practices and standards in hardware random number generation require

- designers to characterize the entropy generated by circuits
- testing of the signal from the entropy source at run time
- post-processing by running output through cryptographic hash function

Card behavior very clearly not FIPS-compliant.

Why are government-certified smartcards generating weak keys?

Best practices and standards in hardware random number generation require

- designers to characterize the entropy generated by circuits
- testing of the signal from the entropy source at run time
- post-processing by running output through cryptographic hash function

Card behavior very clearly not FIPS-compliant.

Hypothesized failure:

- Hardware RNG has underlying weakness that causes failure in some situations.
- Card software not operated in FIPS mode
  $\Longrightarrow$ no testing or post-processing RNG output.

Disclosure and Response

- Disclosure to Taiwanese government in April 2012, June 2013.
- July 2012: MOICA replaced cards for GCD vulnerable certificates.
- July 2013: MOICA told us they planned to replace full “bad batch” of cards.

Disclosure and Response

August 2013: From Email to Research Team

"It took more effort than we expected to locate the affected cards. . . Now, we believe that have revoked all the problematic certificates we found and informed those affected cards holder to replace their cards. Furthermore, we are now implementing the coppersmith method based on your paper to double confirm that there are no any affected cards slipped away."

Disclosure and Response

August 2013: From Email to Research Team

"It took more effort than we expected to locate the affected cards. . . Now, we believe that have revoked all the problematic certificates we found and informed those affected cards holder to replace their cards. Furthermore, we are now implementing the coppersmith method based on your paper to double confirm that there are no any affected cards slipped away."

September 2013: Public Press Release (In Chinese)

"Regarding the internet news about CDC weak keys and how we have dealt with this problem. . . the paper cited in the news is a result of government sponsored research. . . As a result, we have replaced all vulnerable cards in July 2012. . . So all the keys used now are safe."

Conclusions

Widely used crypto can fail in interesting ways that affect real-world security.

Entropy generation is a hard problem and failures can be masked in cryptographic output.

Complex systems can experience cascading failures that make otherwise good crypto trivial to break.

We’ve seen several examples of these failures that weren’t detected until we beat some public keys over the head with math.

Factoring RSA keys from certified smart cards: Coppersmith in the wild
Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren, Asiacrypt 2013.