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,
Nicko van Someren

September 6, 2013

Problems with non-randomness

- 2012 Heninger–Durumeric–Wustrow–Halderman,
- 2012 Lenstra–Hughes–Augier–Bos–Kleinjung–Wachter.
- Factored tens of thousands of public keys on the Internet
  ... typically keys for your home router, not for your bank.
- Why? **Many deployed devices shared prime factors.**
- Most common problem: horrifyingly bad interactions between OpenSSL key generation, `/dev/urandom` seeding, entropy sources.
- The Heninger team has lots of material online at
  `http://factorable.net`

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
$\gcd\{N_1, N_2N_3N_4\cdots\}$ (this gcd is > 1 if N_1 shares a prime);
$\gcd\{N_2, N_1N_3N_4\cdots\}$ (this gcd is > 1 if N_2 shares a prime);
$\gcd\{N_3, N_1N_2N_4\cdots\}$ (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)]
```

Nice followup student projects in data mining

1. Download all certificates of type X; extract RSA keys.
2. Check for common factors.
3. Write report that you've done the work and there are none.

Nice followup student projects in data mining

1. Download all certificates of type X; extract RSA keys.
2. Check for common factors.
3. Write report that you've done the work and there are none.

This started as such a student project on a very nice system:
MOICA: Certificate Authoritiy of MOI (Ministry of the Interior).
In Taiwan all citizens can get a smartcard with signing and encryption ability to

- file personal income taxes,
- update car registration,
- make transactions with government agencies (property registries, national labor insurance, public safety, and immigration),
- file grant applications,
- interact with companies (e.g. Chunghwa Telecom).

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.
- About 3,002,000 certificates (all using RSA keys) 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)

         .
         .
         .
```

This project took a slightly different turn

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 Taiwan Citizen Digital Certificates
(out of 2.26 million keys with 1024 bits).

This project took a slightly different turn

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 Taiwan Citizen Digital Certificates
(out of 2.26 million keys with 1024 bits).

Wrote report that some keys are factored, informed MOI.

This project took a slightly different turn

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 Taiwan Citizen Digital Certificates
(out of 2.26 million keys with 1024 bits).

Wrote report that some keys are factored, informed MOI.

End of story.

This project took a slightly different turn

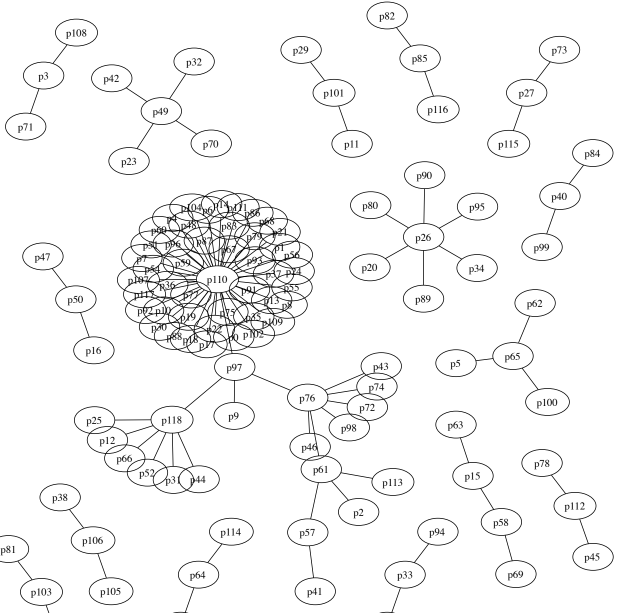
HITCON 2012 (July 20–21):
Prof. Li-Ping Chou presents “Cryptanalysis in real life”
(based on work with Yun-An Chang and Chen-Mou Cheng)

Factored 103 Taiwan Citizen Digital Certificates
(out of 2.26 million keys with 1024 bits).

Wrote report that some keys are factored, informed MOI.

End of story?

January 2013: Closer look at the 119 primes



Look at the primes!

Prime factor p110 appears 46 times

```
c0000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
000000000000000000000000000002f9
```

Look at the primes!

Prime factor p110 appears 46 times

```
c0000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
000000000000000000000000000002f9
```

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

```
c9242492249292499249492449242492
24929249924949244924249224929249
92494924492424922492924992494924
492424922492924992494924492424e5
```

Several other factors exhibit such a pattern.

How is this pattern generated?

1100100100100100001001001001001000100100100100101001001001001001
1001001001001001010010010010010001001001001001000010010010010010
0010010010010010100100100100100110010010010010010100100100100100
0100100100100100001001001001001000100100100100101001001001001001
1001001001001001010010010010010001001001001001000010010010010010
0010010010010010100100100100100110010010010010010100100100100100
0100100100100100001001001001001000100100100100101001001001001001
1001001001001001010010010010010001001001001001000010010011100101

How is this pattern generated?

Swap every 16 bits in a 32 bit word

0010010010010010 1100100100100100 1001001001001001 0010010010010010
0100100100100100 1001001001001001 0010010010010010 0100100100100100
1001001001001001 0010010010010010 0100100100100100 1001001001001001
0010010010010010 0100100100100100 1001001001001001 0010010010010010
0100100100100100 1001001001001001 0010010010010010 0100100100100100
1001001001001001 0010010010010010 0100100100100100 1001001001001001
0010010010010010 0100100100100100 1001001001001001 0010010010010010
0100100100100100 1001001001001001 0010010011100101 0100100100100100

How is this pattern generated?

Realign

001001001001001011001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001
00100100100100100100100100111001010100100100100100

How is this pattern generated?

Realign

0010010010010010110010010010010010010010010010010010010010010010010010010
001001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001001
001001001001001001001001001001001001001001001001001001001001001001001001001
00100100100100100100100111001010100100100100100

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

Prime generation

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

Factoring by trial division

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

Do this for any pattern:
0,1,001,010,011,100,101,110
00001,00010,00011,00100,00101,0011,00111,01000,01001,01010,. . .
00000001,0000011,0000101,0000111,0001001,. . .

Factoring by trial division

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

Do this for any pattern:
0,1,001,010,011,100,101,110
00001,00010,00011,00100,00101,0011,00111,01000,01001,01010,. . .
00000001,0000011,0000101,0000111,0001001,. . .
Computing GCDs factored 105 moduli, of which 18 were new.

Factoring by trial division

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

Do this for any pattern:
0,1,001,010,011,100,101,110
00001,00010,00011,00100,00101,0011,00111,01000,01001,01010,...
00000001,0000011,0000101,0000111,0001001,...
Computing GCDs factored 105 moduli, of which 18 were new.
Factored 4 more keys using patterns of length 9.

Patterns do not find all factors

These primes

```
c0000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
0000000000000000000000000002030b
```

```
c0000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000100000177
```

were found via GCDs, but not from the patterns.

Patterns do not find all factors

These primes

```
c0000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
000000000000000000000000000 2030b
```

```
c0000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000100000177
```

were found via GCDs, but not from the patterns.
Looks like base pattern 0 with some bits flipped.

Coppersmith's method of finding roots mod N

Assume that prime factor p of N has form

$$p = a + r,$$

a is one of the 512-bit patterns
r is a small integer to account for bit errors (and incrementing to next prime.
Coppersmith and Howgrave-Graham:

- Define polynomial
$$f(x) = a + x;$$
- find root r of f modulo a large divisor of N (of size approximately $N^{1/2} \approx p$).

Coppersmith's method of finding roots mod N

Assume that prime factor p of N has form

$$p = a + r,$$

a is one of the 512-bit patterns
r is a small integer to account for bit errors (and incrementing to next prime.
Coppersmith and Howgrave-Graham:

- Define polynomial
$$f(x) = a + x;$$
- find root r of f modulo a large divisor of N (of size approximately $N^{1/2} \approx p$).
- Yes, we have seen millions of papers on this ...

Coppersmith's method of finding roots mod N

Assume that prime factor p of N has form

$$p = a + r,$$

a is one of the 512-bit patterns
r is a small integer to account for bit errors (and incrementing to next prime.
Coppersmith and Howgrave-Graham:

- Define polynomial
$$f(x) = a + x;$$
- find root r of f modulo a large divisor of N (of size approximately $N^{1/2} \approx p$).
- Yes, we have seen millions of papers on this ... but to our knowledge this is the first application of Coppersmith's method in the wild.

Find root r of $f(x) = a + x$

- Let $r \le X$.
- Use lattice basis reduction to construct a new polynomial $g(x)$ where $g(r) = 0$ over the integers, and thus we can factor g to discover it.
- Construct the lattice L as

$$\begin{bmatrix} X^2 & Xa & 0 \\ 0 & X & a \\ 0 & 0 & N \end{bmatrix}$$

  corresponding to the coefficients of the polynomials $N, f(Xx), Xxf(Xx)$;
- run LLL lattice basis reduction;
- regard the shortest vector as coefficients of polynomial $g(Xx)$.
- Compute the roots r_i of $g(x)$ and check if $a + r_i$ divides N.

Bounds on the error part in $f(x) = a + x$

- Each lattice vector g is linear combination of N and f, i.e. $g(r_i) \equiv 0 \bmod p$.
- p is found if $g(r_i) = 0$.
- Holds if coefficients of g are sufficiently small.
- The shortest vector v_1 found by LLL is of length

  $$|v_1| \leq 2^{(\dim L - 1)/4} (\det L)^{1/\dim L},$$

  which must be smaller than p for the attack to be guaranteed to succeed.
- In our situation this translates to

  $$2^{1/2} \left(X^3 N\right)^{1/3} < N^{1/2} \Leftrightarrow X < 2^{-1/2} N^{1/6},$$

  so for $N \approx 2^{1024}$ we can choose X as large as 2^{170},

Factors!

- Ran this one all 164 patterns; about 1h/pattern.
- Factored 160 keys, including 39 previously unfactored keys.
- Found all but 2 of the 103 keys factored with the GCD method.

Factors!

- Ran this one all 164 patterns; about 1h/pattern.
- Factored 160 keys, including 39 previously unfactored keys.
- Found all but 2 of the 103 keys factored with the GCD method.
- Missing 2 keys have factor e0000. . . 0f,
  so we included e000 as pattern, but didn't find more factors.

Handling more errors

Increase lattice dimension:
For dimension 5 we used basis

$$\{N^2, Nf(xX), f^2(xX), xXf^2(xX), (xX)^2 f^2(xX)\}$$

which up to LLL constants handles $X < N^{1/5}$,
i.e. up to 204 erroneous bottom bits.

Coppersmith's method can find primes with errors in up to $1/2$ of their bits, i.e. $X < N^{1/4}$ using lattices of higher dimension.
But getting close to this bound is prohibitively expensive

Errors in the top bits

- How to find e000...f ($= 2^{511} + 2^{510} + 2^{509} + 15$)?
- How about this prime?

```
ffffaa55ffffffff3cd9fe3ffff676
fffffffffe00000000000000000000
000000000000000000000000000000
00000000000000000000000000009d
```

- Not found by the lattice attacks with the basic patterns.
- Can use Coppersmith on $f(x) = a + 2^t x$ and vary bottom bits of a to account for `nextprime`.
- To get 50% chance of success, need to study 128 new patterns for every old pattern.

Bivariate Coppersmith

- Better approach: Change the lattice!
- Assume p has the form

$$p = a + 2^t s + r$$

a is one of the 512-bit patterns
r is a small integer to account for bit errors (and incrementing to next prime,
s is a small integer to account for bit errors,
t is the offset where top errors occur.

- Build lattice around bivariate polynomial
  $f(x, y) = a + 2^t x + y$ and N.
- Lattice naturally has higher dimension and higher powers of N — need N, xN, and $f(x, y)$.
- Approach similar to Herrmann and May (Asiacrypt 2008), but basis optimized for speed (not asymptotics).

Bivariate Coppersmith for $f(x, y) = a + 2^t x + y$

- Get basis as vectors in $\{1, x, y, x^2, \dots, y^{k-1}x, y^k\}$ of $\{N, xXN, f, (xX)^2N, (xX)f, \dots, (yY)^{k-2}(xX)f, (yY)^{k-1}f\}$.
- Determinant of this lattice is

$$\det L = N^{k+1}(XY)^{\binom{k+2}{3}}.$$

and the dimension is $\binom{k+2}{2}$. Omitting the approximation factor of LLL, we want to ensure that

$$(\det L)^{1/\dim L} < p$$

$$\left(N^{k+1}(XY)^{\binom{k+2}{3}}\right)^{1/\binom{k+2}{2}} < N^{1/2}.$$

- Concretely:
  - $k = 3$ for $N \approx 2^{1024}$ gives $XY < 2^{102}$
  - $k = 4$ should let us find $XY < 2^{128}$.
  - $k = 2$ results in a theoretical bound $XY < 1$,

Bivariate Coppersmith for $f(x, y) = a + 2^t x + y$

- Get basis as vectors in $\{1, x, y, x^2, \ldots, y^{k-1}x, y^k\}$ of $\{N, xXN, f, (xX)^2N, (xX)f, \ldots, (yY)^{k-2}(xX)f, (yY)^{k-1}f\}$.
- Determinant of this lattice is

$$\det L = N^{k+1}(XY)^{\binom{k+2}{3}}.$$

and the dimension is $\binom{k+2}{2}$. Omitting the approximation factor of LLL, we want to ensure that

$$(\det L)^{1/\dim L} < p$$

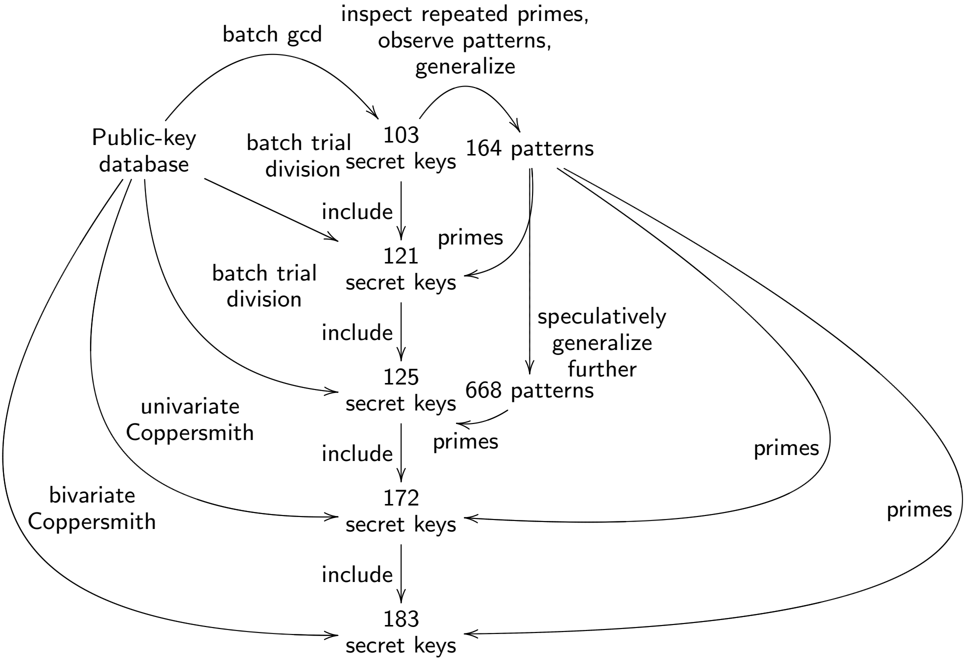
$$\left(N^{k+1}(XY)^{\binom{k+2}{3}}\right)^{1/\binom{k+2}{2}} < N^{1/2}.$$

- Concretely:
  - $k = 3$ for $N \approx 2^{1024}$ gives $XY < 2^{102}$
  - $k = 4$ should let us find $XY < 2^{128}$.
  - $k = 2$ results in a theoretical bound $XY < 1$, but was useful.

Results

- $k = 3$: used base pattern $a = 0$,
  10-dimensional lattices
  $Y = 2^{30}$, $X = 2^{70}$, and $t = 442$.
- $k = 4$: used base pattern $a = 2^{511} + 2^{510}$,
  15-dimensional lattices
  $Y = 2^{28}$ and $X = 2^{100}$,
  five different error offsets: $t = 0$ with $Y = 2^{128}$ and $X = 1$,
  and $t \in \{128, 228, 328, 428\}$ with $Y = 2^{28}$ and $X = 2^{100}$.
- $k = 2$: used base pattern $a = 2^{511} + 2^{510}$,
  6-dimensional lattices
  $X = 4$, $Y = 4$, all choices of t as above.

k	$\log_2(XY)$	# t	# factored keys	total running time
2	4	5	105	4.3 hours
3	100	1	112	2 hours
4	128	5	109	20 hours


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

Why are government-issued smartcards generating weak keys?

Why are government-issued smartcards generating weak keys?

Card behavior very clearly not FIPS-compliant.

Why are government-issued smartcards generating weak keys?

Card behavior very clearly not FIPS-compliant.

Hypothesized failure:

- Hardware ring oscillator gets stuck in some conditions or does not output quickly enough.
- Card software not post-processing RNG output.

Important Lesson:

- Nontrivial GCD is not the only way RSA can fail with bad RNG.