Factoring RSA keys from certified smart cards: Coppersmith in the wild

Daniel J. Bernstein[1,2], Yun-An Chang[3], Chen-Mou Cheng[3], Li-Ping Chou[4], Nadia Heninger[5], Tanja Lange[2], and Nicko van Someren[6]

[1] Department of Computer Science, University of Illinois at Chicago, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, the Netherlands
tanja@hyperelliptic.org
[3] Research Center for Information Technology Innovation
Academia Sinica, Taipei, Taiwan
{ghfjdksl,doug}@crypto.tw
[4] Department of Computer Science and Information Engineering
Chinese Culture University, Taipei, Taiwan
randomalg@gmail.com
[5] Department of Computer and Information Science, University of Pennsylvania
nadiah@cis.upenn.edu
[6] Good Technology Inc.
nicko@good.com

Abstract. This paper explains how an attacker can efficiently factor 184 distinct RSA keys out of more than two million 1024-bit RSA keys downloaded from Taiwan's national "Citizen Digital Certificate" database. These keys were generated by government-issued smart cards that have built-in hardware random-number generators and that are advertised as having passed FIPS 140-2 Level 2 certification.

These 184 keys include 103 keys that share primes and that are efficiently factored by a batch-GCD computation. This is the same type of computation that was used last year by two independent teams (USENIX Security 2012: Heninger, Durumeric, Wustrow, Halderman; Crypto 2012: Lenstra, Hughes, Augier, Bos, Kleinjung, Wachter) to factor tens of thousands of cryptographic keys on the Internet.

The remaining 81 keys do not share primes. Factoring these 81 keys requires taking deeper advantage of randomness-generation failures: first using the shared primes as a springboard to characterize the failures, and then using Coppersmith-type partial-key-recovery attacks. This is the first successful public application of Coppersmith-type attacks to keys found in the wild.

Keywords: RSA, smart cards, factorization, Coppersmith, lattices

---

This work was supported by NSF (U.S.) under grant 1018836, by NWO (Netherlands) under grants 639.073.005 and 040.09.003, and by NSC (Taiwan) under grant 101-2915-I-001-019. Cheng worked on this project while at Technische Universität Darmstadt under the support of Alexander von Humboldt-Stiftung. Heninger worked on this project while at Microsoft Research New England. Permanent ID of this document: 278505a8b16015f4fd8acae818080edd. Date: 2013.09.10.

1 Introduction

In 2003, Taiwan introduced an e-government initiative to provide a national public-key infrastructure for all citizens. This national certificate service allows citizens to use "smart" ID cards to digitally authenticate themselves to government services, such as filing income taxes and modifying car registrations online, as well as to a growing number of non-government services. RSA keys are generated by the cards, digitally signed by a government authority, and placed into an online repository of "Citizen Digital Certificates".

On some of these smart cards, unfortunately, the random-number generators used for key generation are fatally flawed, and have generated real certificates containing keys that provide no security whatsoever. This paper explains how we have computed the secret keys for 184 different certificates.

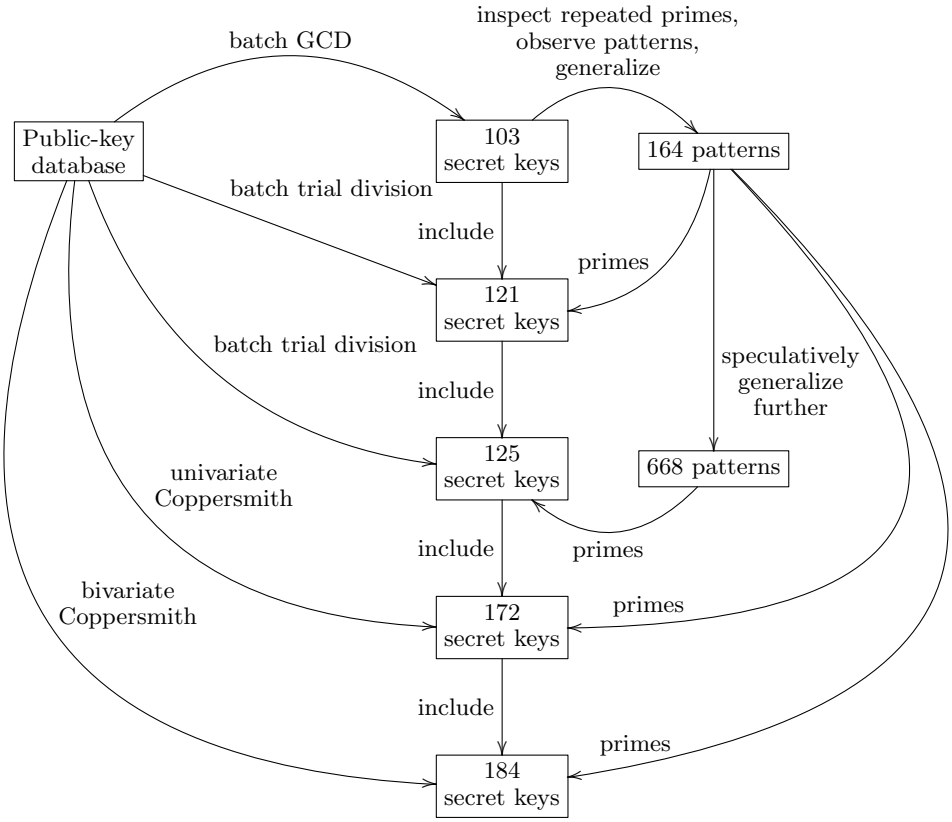

Fig. 1. Retrospective summary of the data flow leading to successful factorizations. After successfully factoring keys using a batch GCD algorithm, we characterized the failures, and used trial division to check for broader classes of specified primes (input on the right) as exact divisors. We then extended the attack and applied Coppersmith's method to check for the specified primes as approximate divisors.

1.1 Factorization techniques

Bad randomness is not new. Last year two independent research teams [13,17] exploited bad randomness to break tens of thousands of keys of SSL certificates on the Internet, a similar number of SSH host keys, and a few PGP keys.

Our starting point in this work is the same basic attack used in those papers against poorly generated RSA keys, namely scanning for pairs of distinct keys that share a common divisor (see Section 3). The basic GCD attack, applied to the entire database of Citizen Digital Certificates, shows that 103 keys factor into 119 different primes.

We go beyond this attack in several ways. First, the shared primes provide enough data to build a model of the prime-generation procedure. It is surprising to see *visible* patterns of non-randomness in the primes generated by these smart cards, much more blatant non-randomness than the SSL key-generation failures identified by [13,17]. One expects smart cards to be controlled environments with built-in random-number generators, typically certified to meet various standards and practically guaranteed to avoid such obvious patterns. For comparison, the SSL keys factored last year were typically keys generated by low-power networked devices such as routers and firewalls running the Linux operating system while providing none of the sources of random input that Linux expects.

The next step is extrapolation from these prime factors: we hypothesize a particular model of randomness-generation failures consistent with 18 of the common divisors. The same model is actually capable of generating 164 different primes, and testing all of those primes using batch trial division successfully factors further keys. One might also speculate that the cards can generate primes fitting a somewhat broader model; this speculation turns out to be correct, factoring a few additional keys and bringing the total to 125. See Section 4 for a description of the patterns in these primes.

There are also several prime factors that are similar to the 164 patterns but that contain sporadic errors: some bits flipped here and there, or short sequences of altered bits. We therefore mount several Coppersmith-style lattice-based partial-key-recovery attacks to efficiently find prime divisors close to the patterns. The univariate attacks (Section 5) allow an arbitrary stretch of errors covering the bottom 40% of the bits of the prime. The bivariate attacks (Section 6) allow two separate stretches of errors. The internal structure of the patterns makes them particularly susceptible to these attacks. These attacks produce dozens of additional factorizations, raising the total to 184.

In the end nearly half of the keys that we factored did *not* share any common divisors with other keys; most of these were factored by the Coppersmith-style attacks. This is, to our knowledge, the first publicly described instance of a Coppersmith-style attack breaking keys in the wild.

1.2 Certification

The flawed keys were generated by government-issued smart cards that both the certification authority and manufacturer advertise as having passed stringent standards certifications. See Section 2.1.

It is clear from their externally visible behavior, as shown in this paper, that the random-number generators used to generate the vulnerable keys actually fall far short of these standards. This demonstrates a failure of the underlying hardware and the card's operating system, both of which are covered by certification.

1.3 Response to vulnerabilities

When we reported the common-divisor vulnerabilities to government authorities, their response was to revoke exactly the certificates sharing common factors and to issue new cards only to those users. See Section 7 for more details.

Our further factorizations demonstrate how dangerous this type of response is. Randomness-generation failures sometimes manifest themselves as primes appearing twice, but sometimes manifest themselves as primes that appear only once, such as the primes that we found by Coppersmith-type attacks. Both cases are vulnerable to attackers with adequate models of the randomness-generation process, while only the first case is caught by central testing for repeated primes.

We endorse the idea of centrally testing RSA moduli for common divisors as a mechanism to detect some types of randomness-generation failures. We emphasize that finding repeated primes is much more than an indication that those particular RSA keys are vulnerable: it shows that the underlying randomness-generation system is malfunctioning. The correct response is not merely to eliminate those RSA keys but to revoke all keys generated with that generation of hardware and throw away the entire randomness-generation system, replacing it with a properly engineered system.

We also emphasize that an absence of common divisors is not an indication of security. If the primes generated by these smart cards had been modified to include a card serial number as their top bits then the keys would have avoided common divisors but the primes would still have been reasonably predictable to attackers. Our work illustrates several methods of translating different types of malfunctioning behavior into concrete vulnerabilities. There are many potential vulnerabilities resulting from bad randomness; it is important to thoroughly test every component of a random-number generator, not merely to look for certain types of extreme failures.

2 Background

2.1 The Taiwan Citizen Digital Certificate program

Taiwan's Citizen Digital Certificates (CDCs) are a standard means of authentication whenever Taiwanese citizens want to do business over the Internet with the government and an increasing number of private companies.

CDCs are issued by the Ministry of Interior Certificate Authority (MOICA), a level 1 subordinate CA of the Taiwanese governmental PKI. Since the program's launch in 2003, more than 3.5 million CDCs have been issued, providing public key certificate and attribute certificate services. These digital certificates

form a basis for the Taiwanese government's plan to migrate to electronic certificates from existing paper certificates for a range of applications including national and other identification cards, driver's licenses, and various professional technician licenses.

Today, Taiwanese citizens can already use the CDC to authenticate themselves over the Internet in a number of important government applications, e.g., to file personal income taxes, update car registration, and make transactions with government agencies such as property registries, national labor insurance, public safety, and immigration. In addition, the CDC is accepted as a means of authentication by a variety of organizations such as the National Science Council, several local governments, and recently some private companies such as Chunghwa Telecom. Overall, the CDC program appears quite successful as a two-sided network, as it has attracted an increasing number of both applications and subscribers.

Certificate registration: In order to generate CDCs, citizens bring their (paper) ID cards to a government registration office. A government official places the (smart) ID card into a registration device. The device prompts the card to generate a new cryptographic key, and the public key is incorporated into a certificate to be signed by MOICA. The certificate is made available in a database online for authentication purposes. In general, an individual will have two certificates: one for signing, and one for encryption, each with distinct keys.

Standards certifications: MOICA states that these cards are "high security", and "have been accredited to FIPS 140-1 level 2", and also that "A private key is created inside and the private key can't export from IC card after key created". (See [20] or search for "FIPS" on MOICA's website `http://moica.nat.gov.tw/html/en/index.htm`.) For comparison, the SSL keys factored last year were generated by software-hardware combinations that had never claimed to be evaluated for cryptographic security, such as Linux running on a home router.

2.2 Collecting certificates

In March 2012, inspired by the results of [13] and [17], we retrieved 3002273 CDCs from the MOICA LDAP directory at `ldap://moica.nat.gov.tw`. Out of these CDCs, 2257569 have 1024-bit RSA keys, while the remaining, newer 744704 have 2048-bit RSA keys, as in 2010 MOICA migrated to 2048-bit RSA and stopped issuing certificates of 1024-bit RSA keys.

The 1024-bit CDCs contain 2086177 distinct moduli, of which 171366 moduli appear more than once. The repeated moduli appear to all be due to expired certificates still contained in the database, which contain the same keys as renewal certificates issued to the same individuals.

2.3 Random number generation

While generating high-quality random numbers is critical to the security of cryptographic systems, it is also notoriously difficult to do. Non-deterministic behavior is considered to be a fault in almost every other component of a computer, but it is a crucial component of generating random numbers that an attacker cannot predict. Several national and international standards for random number generation [22,1,11] specify correct behavior for these types of systems. In general, software pseudo-random number generators require a significant amount of entropy before their output is useful for cryptographic purposes.

As we will see later in the paper, the smart cards used in the PKI we examined fail to follow many well-known best practices and standards in hardware random number generation: they appear to utilize a source of randomness that is prone to failing, they fail to perform any run-time testing before generating keys, and they clearly do not apply any post-processing to the randomness stream. The lack of testing or post-processing causes the initial randomness-generation failure to be much more damaging than it would have been otherwise.

Analog RNG circuits: An analog circuit is the standard choice when hardware designers have the luxury of designing dedicated circuits for random-number generation. An analog circuit allows the designer to obtain randomness from simple quantum effects. While the use of radioactive decay is rare in commercial products, the quantum noise exhibited by a current through a suitably biased diode can be amplified and sampled to deliver a high-quality entropy source.

On-chip RNG circuits: Mixing analog and digital circuits on the same die is costly, so chip designers often seek other sources of unpredictability. These sources can include variation in gate propagation delays or gate metastability, which exhibit inherent randomness. Designers can explicitly harness gate-delay variation by building sets of free-running ring oscillators and sampling the behavior at hopefully uncorrelated intervals. To take advantage of randomness in gate metastability, designers build circuits that output bits based on the time it takes for the circuit to settle to a steady state, a variable which should be hard to predict. These designs are often tricky to get right, as the chip fabrication process can reduce or eliminate these variations, and subtle on-chip effects such as inductive coupling or charge coupling between components can cause free-running oscillators to settle into synchronised patterns and metastable circuits to predictably land one way or the other depending on other components nearby on the chip.

Handling entropy sources: Even with a perfectly unpredictable source of randomness, care needs to be taken to convert the raw signal into usable random numbers. Generally, designers characterize circuits in advance to understand the entropy density, test the signal from the entropy source at run time, and run the output through a compression function such as a cryptographically secure hash function. These practices are required by a number of security standards such as FIPS 140 [21].

3 Batch GCD

This section reviews the approach of [13,17] for detecting common factors in a collection of RSA keys, and reports the results of this approach applied to the collection of Citizen Digital Certificates.

If there are two distinct RSA moduli $N_1 = pq_1$ and $N_2 = pq_2$ sharing exactly one prime factor p, then the greatest common divisor of N_1 and N_2 will be p. Computing this GCD is fast, and dividing it out of N_1 and N_2 produces the other factors q_1 and q_2.

Of course, this type of vulnerability should never arise for properly generated RSA keys. However, since [13,17] had observed weak random-number generators producing keys with repeated factors in the wild, we began by checking whether there were repeated factors among the Citizen Digital Certificates.

Instead of the naive quadratic-time method of doing this computation (checking each N_1 against each N_2), we used a faster batch-GCD algorithm using product and remainder trees described in [2,13]. We used the C implementation available at `https://factorable.net/resources.html`.

We ran this implementation on the 3192962 distinct RSA moduli and found that 103 moduli were factored due to nontrivial common factors. This computation, parallelized across four cores of a 3.1GHz AMD FX-8120, finished in just 45 minutes.

4 Attacking patterned factors

A properly functioning random number generator would never generate identical 512-bit primes, so the discovery of repeated prime factors described in the previous section immediately indicates that the random-number-generation process producing these keys is broken. This section analyzes the structure of the repeated factors generated by the flawed random-number generator and designs a targeted attack against this structure.

The 103 moduli with repeated factors show a remarkable distribution of the shared factors; see Figure 2. The complete list of factors found using the GCD approach is given in Appendix A.

One prime factor, p110, appears a total of 46 times with different second primes. The hexadecimal representation of this factor is

```
0xc0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000002f9
```

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

The next most common factor, repeated 7 times, is

```
0xc924249224929249924949244924249224929249924949244924249224922492924
99249492449242492249292499249492449242492249292499249494924492424e5
```

which displays a remarkable periodic structure. The binary representation of this integer, excluding a few most and least significant bits, is a repeated sequence of the string 001 with a “hiccup” every 16 bits.

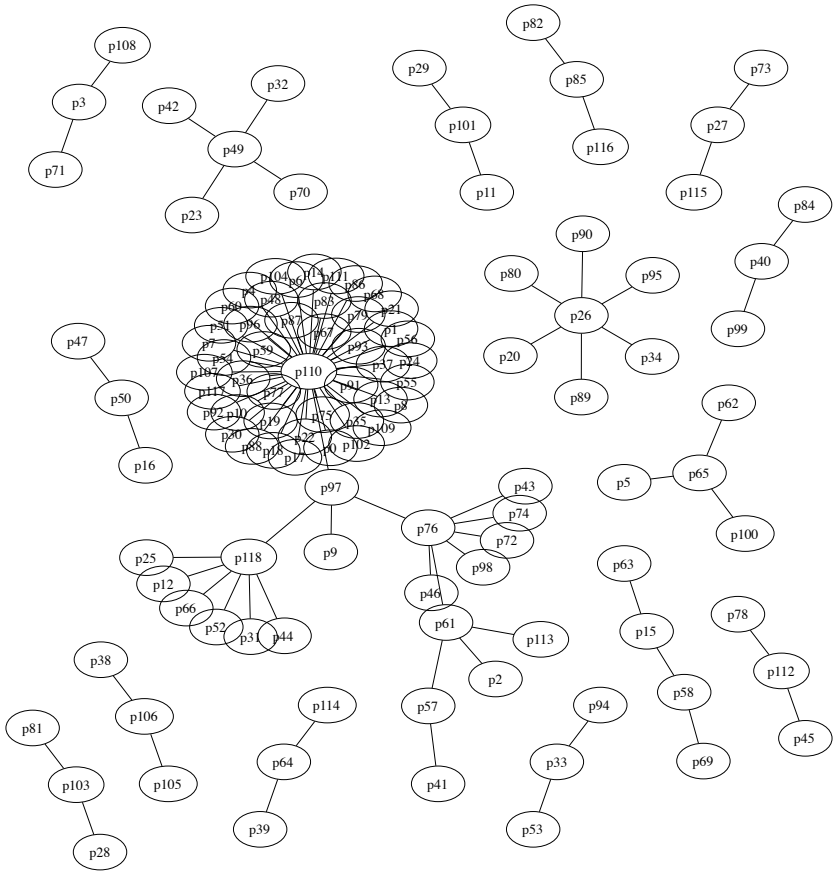

Fig. 2. Relationships between keys with shared factors. Each ellipse represents a prime; edges connect prime factors dividing the same modulus.

Nearly all of the shared prime factors had a similar and immediately apparent periodic structure. We hypothesized that nearly every repeated prime factor had been generated using the following process:

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.

We generated the 164 distinct primes of this form corresponding to all patterns of length 1, 3, 5, and 7 and tested divisibility with each modulus. This factored a total of 105 moduli, including 18 previously unfactored moduli, for a total of 121.

None of the repeated primes exhibit a (minimal) period of length 9 or larger. On the other hand, the data for period lengths 1, 3, 5, 7 shows that patterns with longer periods typically appear in fewer keys than patterns with shorter periods, and are thus less likely to appear as divisors of two or more keys, raising the question of whether there are primes with larger periods that appear in only one key and that are thus not found by the batch-GCD computation. We therefore extended this test to include length-9 periods and length-11 periods. The length-9 periods factored 4 more keys but the length-11 periods did not factor any new keys, leading us to speculate that 3, 5, and 7 are the only factors of the period length. We then ran a complete test on all length-15 patterns but did not find any further factors. The total number of certificates broken by these divisibility tests, together with the initial batch-GCD computation, is 125.

Sporadic errors: The handful of shared prime factors in our sample of GCD-factored keys that did not match the above form were differing from patterns in very few positions. We experimented with finding more factors using brute-force search starting from `0xc0...0` and found a few new factors, but these factors are more systematically and efficiently found using LLL in Coppersmith's method, as described in the next section.

We also experimented with searching for sporadic errors in factors using the techniques of Heninger and Shacham [14] and Paterson, Polychroniadou, and Sibborn [23]. The main idea is to assume that both of the factors of a weak modulus share nearly all bits in common with a known pattern, with only sporadic errors in each. It is then possible to recover the primes by enumerating, bit by bit, a search tree of all possible prime factors, and using depth- or breadth-first search with pruning to find a low-Hamming weight path through the search tree of all solutions.

Unfortunately, there are a few difficulties in applying this idea to the case at hand. The first is that because the primes are generated by incrementing to the next prime, a single sporadic error is likely to cause the least significant 9 bits of each prime to appear random (except for the least significant bit which is set to 1), so generating a solution tree from the least significant bits necessarily begins with that much brute forcing. Second, there is only a single constraint on the solutions (the fact that $pq = N$), instead of four constraints, which results in a lower probability of an incorrect solution being pruned than in the examples considered by [14,23]. And finally, in order to apply the algorithms, we must guess the underlying pattern, which in our case requires applying the algorithm to 164^2 possibilities for each modulus.

Applying this algorithm using only the all-zeros pattern for both factors to the 45 moduli with 20 bits of consecutive zeros took 13 minutes and factored 5 moduli. All of these moduli were also factored by the GCD method or Coppersmith methods described in the next section.

5 Univariate Coppersmith

Several of the factors computed via the GCD algorithm in Section 3 follow the bit patterns described in Section 4, but are interrupted by what appear to be sporadic errors. Coppersmith's method [6,7] factors RSA moduli if the top bits of the primes are known, which matches our situation if the errors appear in the bottom few bits of a factor. The method uses lattice basis reduction to factor in polynomial time if at least half of the most significant bits of a prime factor are known; however, since the running time scales very poorly as one approaches this bound, we will be more interested in less optimal parameters that are efficient enough to apply speculatively to millions of keys.

This section presents this method following Howgrave-Graham [16] for lattices of dimension 3 and 5 and gives an outlook of how more keys could be factored using larger dimensions. The idea is as follows: we assume that some prime factor p of N is of the form

$$p = a + r$$

where a is a known 512-bit integer (one of the bit patterns described in the previous section) and r is a small integer error to account for a sequence of bit errors (and incrementing to next prime) among the least significant bits of p.

In the Coppersmith/Howgrave-Graham method, we can write a polynomial

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

and we would like to find a root r of f modulo a large divisor of N (of size approximately $N^{1/2} \approx p$). Let X be the bound on the size of the root we are searching for. We will 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 r.

Let L be the lattice generated by the rows of the basis matrix

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

corresponding to the coefficients of the polynomials $Xxf(Xx), f(Xx), N$. Any vector in L can be written as an integer combination of basis vectors, and, after dividing by the appropriate power of X, corresponds to the coefficients of a polynomial $g(x)$ which is an integer combination of f and N, and is thus divisible by p by construction. A prime p is found by this method if we can find g such that $g(r_i) \equiv 0 \bmod p$ holds not only modulo p but over the integers. The latter is ensured if the coefficients of g are sufficiently small, which corresponds to finding a short vector in L.

To find such a short vector, we apply the LLL lattice basis reduction algorithm [18]. To finish the algorithm, we regard the shortest vector in the reduced basis as the coefficients of a polynomial $g(Xx)$, compute the roots r_i of $g(x)$, and check if $a + r_i$ divides N. If so, we have factored N.

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 succeed.

In our situation this translates to

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

so for $N \approx 2^{1024}$ we can choose X as large as 2^{170}, meaning that for a fast attack using dimension-3 lattices up to the bottom third of a prime can deviate from the pattern a. In the following we ignore the factor $2^{(\dim L-1)/4}$ since all lattices we deal with are of small dimension and the contribution compared to N is negligible.

5.1 Experimental results

A straightforward implementation using Sage 5.8 took about one hour on one CPU core to apply this method for one of the 164 patterns identified in Section 4. Running it for all 164 patterns factored 160 keys, obviously including all 105 keys derived from the patterns without error, and found 39 previously unfactored keys.

It is worth noting that the 160 keys included all but 2 of the 103 keys factored with the GCD method, showing that most of the weak primes are based on the patterns we identified and that errors predominantly appeared in the bottom third of the bits. The missing 2 keys are those divisible by `0xe0000...0f`. Including `0xd0000...0, 0xe0000...0, 0xf0000...0` as additional bit patterns did not reveal any factors beyond the known ones, ruling out the hypothesis that the prime generation might set the top 4 bits rather than just 2. Instead this prime must have received a bit error in the top part.

5.2 Handling more errors

Coppersmith's method can find primes with errors in up to 1/2 of their bits using lattices of higher dimension. Getting close to this bound is prohibitively expensive, but trying somewhat larger dimensions than 3 is possible. For dimension 5 we used basis

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

which up to LLL constants handles $X < N^{1/5}$, i.e. up to 204 erroneous bottom bits in p for N of 1024 bits. The computation took about 2 hours per pattern and revealed 6 more factors.

We did not use higher dimensions because the "error" patterns we observed are very sparse making it more profitable to explore multivariate attacks (see Section 6).

5.3 Errors in the top bits

The factor `0xe000...f` ($2^{511}+2^{510}+2^{509}+15$) appeared as a common factor after taking GCDs but was not found by the lattice attacks described in this section applied to the basic patterns described in Section 4. We can apply Coppersmith's attack to search for errors in higher bits of p by defining the polynomial f as $f(x) = a + 2^t x$. Here t is a bit offset giving the location of the errors we hope to learn. The method and bounds described in this section apply as well to this case.

However, since we hypothesize that the prime factors are generated by incrementing to the next prime after a sequence of bits output by the flawed RNG, we will not know the least significant bits of a because they have been modified in the prime generation process. This problem might speculatively be overcome by brute forcing the m least significant bits of each pattern: for each application of the algorithm to a single pattern a, we would apply the algorithm to the 2^{m-1} patterns generated by fixing a and varying the bottom m bits, with the least significant bit always fixed to 1. This will find factors if finding the next prime from the base string with errors did not require incrementing by more than those bottom m bits.

The following rough analysis suggests that for this attack to have a 50% chance of success, we need to apply the algorithm to 128 new patterns for every old pattern. Recall that the chance that a number around z is prime is approximately $1/\log z$, where log is the natural logarithm. In particular, each number around 2^{512} has about a $1/355$ chance of being prime. Since $1-(1-1/355)^{256} \approx 0.5$, trying 128 patterns for the bottom eight bits for odd patterns has a 50% chance of covering a sufficiently large interval to find a prime. See [12] for more precise estimates. Applying this to our 164 base patterns, our implementation would require 20992 core hours, or close to 2.5 core years. It is fairly likely that more factors would be found with this search but the method presented in the following section is more efficient at handling errors in top and bottom positions unless a very large portion of the top bits are erroneous.

6 Bivariate Coppersmith

The lattice attacks described in the previous section let us factor keys with unpredictable bits occurring in the least significant bits of one of the factors, with all of the remaining bits of the factor following a predictable pattern. In this section, we describe how we extended this attack to factor keys with unpredictable bits among the middle or most significant bits of one of the factors, without resorting to brute-forcing the bottom bits.

In the basic setup of the problem, we assume that one of the factors p of N has the form

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

where a is a 512-bit integer with a predictable bit pattern (as described in Section 4), t is a bit offset where a sequence of bit errors s deviating from the

predictable pattern in a occurred during key generation, and r is an error at the least significant bits to account for the implementation incrementing to the next prime.

To apply Coppersmith's method, we can define an equation $f(x, y) = a + 2^t x + y$ and try to use lattice basis reduction to find new polynomials $Q_i(x, y)$ with the property that if $f(s, r)$ vanishes modulo a large unknown divisor p of N and s and r are reasonably small, then $Q_i(s, r) = 0$ over the integers. In that case, we can attempt to find appropriate zeros of Q_i. The most common method to do this is to look at multiple distinct polynomials Q_i and hope that their common solution set is not too large.

These types of bivariate Coppersmith attacks have many cryptanalytic applications, perhaps most prominently Boneh and Durfee's attack against RSA private key $d < N^{0.29}$ [3]. Our approach is very similar to that described by Herrmann and May for factoring RSA moduli with bits known [15], although for the application we describe here, we are less interested in optimal parameters, and more in speed: we wish to find the keys most likely to be factored using very low dimensional lattices.

Algebraic independence: Nearly all applications of multivariate Coppersmith methods require a heuristic assumption that the attacker can obtain two (or several) algebraically independent polynomial equations determined by the short vectors in a LLL-reduced lattice; this allows the attacker to compute a finite (polynomially-sized) set of common solutions. Most theorem statements in these papers include this heuristic assumption of algebraic independence as a matter of course, and note briefly (if at all) that it appears to be backed up experimentally.

Notably, in our experiments, this assumption *did not* hold in general. That is, most of the time the equations we obtained after lattice basis reduction were not algebraically independent, and in particular, the algebraic dependencies arose because all of the short vectors in the lattice were polynomial multiples of a single bivariate linear equation. This linear equation did in fact vanish at the desired solution, but without further information, there are an infinite number of additional solutions that we could not rule out. However, we were often able to find the solution using a simple method that we describe below.

Herrmann and May [15] describe one case where the assumption of algebraic independence did not hold in their experiments, namely when X and Y were significantly larger than the values of s and r. Similar to our case they observed that the polynomials of small norm shared a common factor but unlike in our case this factor was the original polynomial f. Note that the linear polynomial in our case vanishes over the integers at (s, r) while f vanishes only modulo p.

We experimented with running smaller dimensional lattice attacks in order to generate this sublattice more directly. The attack worked with smaller degree equations than theoretically required to obtain a result, but when we experimented with lattices generated from linear equations, this sublattice did not appear. Note that we specify a slightly different basis for the lattice, in terms of monomial powers rather than powers of f, which may have an effect on the

output of the algorithm compared to the examples in [15] and might explain why we find a useful linear equation in the sublattice instead of the useless factor f.

6.1 Implementation details

Lattice construction: Let X and Y be bounds on the size of the roots at x and y we wish to find. Our lattice is constructed using polynomial multiples of $f(x,y) = a + 2^t xX + yY$ and N up to degree k vanishing to degree 1 modulo p. Our lattice basis consists of the coefficient vectors of the set of polynomials

$$\begin{aligned}\{(Yy)^h (Xx)^i f^j N^\ell \mid j+\ell = 1, 0 \le h+i+j \le k\} \\ = \{N, xXN, f, (xX)^2 N, (xX)f, \ldots, (yY)^{k-2}(xX)f, (yY)^{k-1} f\},\end{aligned}$$

using coefficients of the monomials $\{1, x, y, x^2, \ldots, y^{k-1}x, y^k\}$. The 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

$$\begin{gathered}(\det L)^{1/\dim L} < p \\ \left(N^{k+1}(XY)^{\binom{k+2}{3}}\right)^{1/\binom{k+2}{2}} < N^{1/2}.\end{gathered}$$

So for $N \approx 2^{1024}$, setting $k = 3$ should let us find $XY < 2^{102}$ and $k = 4$ should let us find $XY < 2^{128}$. The parameter choice $k = 2$ results in a theoretical bound $XY < 1$, but we also experimented with this choice; see below.

Solving for solutions: After running LLL on our lattice, we needed to solve the system of equations it generated over the integers to find our desired roots. The usual method of doing this in bivariate Coppersmith applications is to hope that the two shortest vectors in the reduced basis correspond to algebraically independent polynomials, and use resultants or Gröbner bases to compute the set of solutions. Unfortunately, in nearly all of our experiments, this condition did not hold, and thus there were an infinite number of possible solutions.

However, a simple method sufficed to compute these solutions in our experiments. In general, the algebraic dependencies arose because the short vectors in the reduced basis corresponded to a sublattice of multiples of the same degree-one equation, with seemingly random coefficients, which vanished at the desired roots. (The coefficient vectors were linearly independent, but the underlying polynomials were not algebraically independent.) The other polynomial factors of these short polynomials did not vanish at these roots. This linear equation has an infinite number of solutions, but in our experiments our desired roots corresponded to the smallest integer solution, which we could obtain by rounding.

Let

$$ux + vy - w = 0$$

be an equation we want to solve for x and y. If u and v are relatively prime, then we can write $c_1u + c_2v = 1$, and parametrize an integer family of solutions

$$x = c_1w + vz$$
$$y = c_2w - uz$$

with $z = c_2x - c_1y$.

In experiments with the already-factored moduli, we observed that the solution was often the minimum integer value of x or y among the solution family. So we searched for z among the rounded values of $-c_1w/v$ and c_2w/u. This solution successfully factored the moduli in our dataset whenever the shortest-vector polynomial returned by lattice basis reduction was not irreducible.

For the handful of cases where the lattice did result in independent equations, we computed the solutions using a Gröbner basis generated by the two shortest vectors.

6.2 Experimental results

We ran our experiments using Sage 5.8 [24] parallelized across eight cores on a 3.1GHz AMD FX-8120 processor. We used fpLLL [4] for lattice basis reduction, and Singular [8] to factor polynomials and compute Gröbner bases. For each lattice, we attempted to solve the system of equations either by factoring the polynomial into linear factors and looking for small solutions of the linear equations as described above or using Gröbner bases.

We attempted to factor each of the 2,086,171 1024-bit moduli using several different parameter settings. For $k = 3$, we had 10-dimensional lattices, and attempted to factor each modulus with the base pattern $a = 0$ using $Y = 2^{30}$, $X = 2^{70}$, and $t = 442$. We then experimented with $k = 4$, $Y = 2^{28}$, and $X = 2^{100}$, which gave us 15-dimensional lattices, and experimented with a base pattern $a = 2^{511} + 2^{510}$ and five different error offsets: $t = 0$ with $Y = 2^{128}$ and $X = 1$, and $t = 128, t = 228, t = 328$, and $t = 428$ with $Y = 2^{28}$ and $X = 2^{100}$. Finally, we experimented with the choice $k = 2$, $X = 4$, $Y = 4$ and the choices of t and a used in the $k = 4$ experiments, which used 6-dimensional lattices and theoretically should not have produced output, but in fact turned out to produce nearly all of the same factorizations as the choices above. We ran one very large experiment, using $k = 2$, $t = 1$, $Y = 2^{28}$, $X = 2^{74}$, $t = 438$, and running against all 164 patterns, which produced 155 factored keys, including two previously undiscovered factorizations. The choice $k = 1$ with the same parameter choices as $k = 2$ did not produce results.

6.3 Handling more errors

From these experimental settings, it seems likely that many more keys could be factored by different choices of parameters and initial pattern values; one is limited merely by time and computational resources. We experimented with iterating over all patterns, but the computation quickly becomes very expensive.

k	$\log_2(XY)$	# t	# patterns	# factored keys	# alg. indep. eqns.	running time
2	4	5	1	104	3	4.3 hours
2	4	1	164	154	21	195 hours
3	100	1	1	112	-	2 hours
4	128	5	1	108	4	20 hours

Table 1. Experimental results from factoring keys using a bivariate Coppersmith approach, using the parameters listed in the text. Where we collected data, we noted the very small number of cases where the lattice produced algebraically independent polynomials; all of the other cases were solved via the heuristic methods described above.

Patterned factors: Mysteriously, using the base patterns $a = 0$ and $a = 2^{511} + 2^{510}$, the algorithm produced factorizations of keys with other patterned factorizations. This is because the product of the bit pattern of the relevant factor multiplied with a small factor produced an integer of the form we searched for, but we are as yet unable to characterize this behavior in general.

Higher powers of p: Similar to the univariate case we can construct higher-dimensional lattices in which each vector is divisible by higher powers of p, e.g. using multiples of N^2, Nf, and f^2 for divisibility by p^2. However, this approach is successful in covering larger ranges of XY only for lattices of dimension at least 28, which would incur a significantly greater computational cost to run over the entire data set of millions of keys.

More variables: More isolated errors can be handled by writing $p = a + \sum_{i=1}^{c} 2^{t_i} s_i$ with appropriate bounds on the $s_i < X_i$ so that the intervals do not overlap. The asymptotically optimal case is described in [15] and reaches similar bounds for $\prod_{i=1}^{c} X_i$ as in the univariate and bivariate case. However, the lattice dimension increases significantly with c. For $c = 3$, i.e. two patches of errors together with changed bottom bits to generate a prime, the condition $(\det L)^{\dim 1/L} < p$ holds only for lattices of dimension at least 35 at which point $X_1 X_2 X_3 < N^{1/14}$ can be found. A lattice of dimension 20 leads to the condition $X_1 X_2 X_3 < 1$. A sufficiently motivated attacker can run LLL on lattices of these dimensions but we decided that factors found thus far were sufficient to prove our point that the smart cards are fatally flawed.

6.4 Extension to implicit factoring

Ritzenhofen and May [19] and Faugère, Marinier, and Renault [9] give algorithms to factor RSA moduli when it is known that two or more moduli have prime factors that share large numbers of bits in common. Unfortunately, these results seem to apply only when the moduli have prime factors of unbalanced size, whereas in our case, both prime factors have 512 bits.

7 Hardware details, disclosure, and response

Around 2006–2007, MOICA switched card platforms from their initial supplier and began to use Chunghwa Telecom's HiCOS PKI smart cards, specifically Chunghwa Telecom HD65145C1 cards (see [5]), using the Renesas AE45C1 smart card microcontroller (see [10]). We have confirmed these details with MOICA.

Unfortunately, the hardware random-number generator on the AE45C1 smart card microcontroller sometimes fails, as demonstrated by our results. These failures are so extreme that they should have been caught by standard health tests, and in fact the AE45C1 does offer such tests. However, as our results show, those tests were not enabled on some cards. This has now also been confirmed by MOICA. MOICA's estimate is that about 10000 cards were issued without these tests, and that subsequent cards used a "FIPS mode" (see below) that enabled these tests.

The random numbers generated by the batch of problematic cards obviously do not meet even minimal standards for collecting and processing entropy. This is a fatal flaw, and it can be expected to continue causing problems until all of the vulnerable cards are replaced.

The AE45C1 chip was certified conformant with Protection Profile BSI-PP-0002-2001 at CC assurance level EAL4+ [10]. The HD65145C1 card and HICOS operating system were accredited to FIPS 140-2 Level 2 [5]. The CC certification stated "The TOE software for random number postprocessing shall be implemented by the embedded software developer", and the FIPS certification was limited to "FIPS mode" (see `http://www.cryptsoft.com/fips140/out/cert/614.html`). However, neither certification prevented the same card from also offering a non-FIPS mode, and neither certification caught the underlying RNG failures. We recommend that industry move to stronger certifications that prohibit error-prone APIs and that include assessments of RNG quality.

In April 2012 we shared with MOICA our preliminary list of 103 certificates compromised by GCD. We announced these results in a talk in Taiwan in July 2012. We provided an extended list of compromised certificates to MOICA and Chunghwa Telecom in June 2013, along with an early draft of this paper. MOICA and Chunghwa Telecom subsequently confirmed our results; asked the cardholders to come in for replacement cards; revoked the compromised certificates; and initiated the task of contacting 408 registration offices across Taiwan to manually trace and replace all of the vulnerable cards from the same batch.

8 Acknowedgements

We thank J. Alex Halderman and Mark Wooding for discussion during this project. We also thank Kenny Paterson and the anonymous reviewers for helpful comments and suggestions, and in particular the encouragement to experiment with sparse key recovery methods.

References

1. ANSI. *ANSI X9.31:1998: Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA)*. American National Standards Institute, 1998.
2. Daniel J. Bernstein. How to find the smooth parts of integers, May 2004. `http://cr.yp.to/papers.html#smoothparts`.
3. Dan Boneh and Glenn Durfee. Cryptanalysis of RSA with private key d less than $n^{0.292}$. In Jacques Stern, editor, *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1999.
4. David Cadé, Xavier Pujol, and Damien Stehlé. *fpLLL*, 2013. `http://perso.ens-lyon.fr/damien.stehle/fplll/`.
5. Ltd. Chunghwa Telecom Co. Hicos pki smart card security policy, 2006. `http://www.cryptsoft.com/fips140/vendors/140sp614.pdf`.
6. Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In Ueli M. Maurer, editor, *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 1996.
7. Don Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptology*, 10(4):233–260, 1997.
8. W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 3-1-6 — A computer algebra system for polynomial computations, 2012. `http://www.singular.uni-kl.de`.
9. Jean-Charles Faugère, Raphaël Marinier, and Guénaël Renault. Implicit factoring with shared most significant and middle bits. In Phong Q. Nguyen and David Pointcheval, editors, *Public Key Cryptography*, volume 6056 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2010.
10. Bundesamt für Sicherheit in der Informationstechnik. Certification report BSI-DSZ-CC-0212-2004 for Renesas AE45C1 (HD65145C1) smartcard integrated circuit version 01, 2004. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Reporte02/0212a_pdf.pdf?__blob=publicationFile`.
11. Bundesamt für Sicherheit in der Informationstechnik. Evaluation of random number generators, 2013. `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Zertifierung/Interpretation/Evaluation_of_random_number_generators.pdf?__blob=publicationFile` and `https://www.bsi.bund.de/DE/Themen/ZertifizierungundAnerkennung/ZertifizierungnachCCundITSEC/AnwendungshinweiseundInterpretationen/AISCC/ais_cc.html`.
12. Andrew Granville. Harald Cramér and the distribution of prime numbers. *Scand. Actuarial J.*, 1995(1):12–28, 1995.
13. Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.
14. Nadia Heninger and Hovav Shacham. Reconstructing rsa private keys from random key bits. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.
15. Mathias Herrmann and Alexander May. Solving linear equations modulo divisors: On factoring given any bits. In Josef Pieprzyk, editor, *ASIACRYPT*, volume 5350 of *Lecture Notes in Computer Science*, pages 406–424. Springer, 2008.
16. Nick Howgrave-Graham. Approximate integer common divisors. In Joseph H. Silverman, editor, *CaLC*, volume 2146 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2001.

17. Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 626–642. Springer, 2012.
18. Arjen K. Lenstra, Hendrik W. Lenstra jun., and László Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.
19. Alexander May and Maike Ritzenhofen. Implicit factoring: On polynomial time factoring given only an implicit hint. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2009.
20. MOICA. Safety questions, 2013. `http://moica.nat.gov.tw/html/en_T2/faq22-066-090.htm`.
21. National Institute of Standards and Technology (NIST). Security requirements for cryptographic modules. Federal Information Processing Standards Publication (FIPS PUB) 140-2, May 2001. `http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`, updated 2002-12-03. See `http://csrc.nist.gov/publications/nistpubs/800-29/sp800-29.pdf` for differences between this and FIPS-140-1.
22. National Institute of Standards and Technology (NIST). Recommendation for random number generation using deterministic random bit generators. NIST Special Publication (NIST SP) 800-90A, January 2012.
23. Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 386–403. Springer, 2012.
24. William A. Stein et al. *Sage Mathematics Software (Version 5.8)*. The Sage Development Team, 2013. `http://www.sagemath.org`.

A Appendix: Raw data

The following data presents all primes found using the GCD method (Section 3); the initial number indicates how often that particular prime was found.


```
46, 0xc00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000002f9
7, 0xc92424922492924992494924492424922492924992494924492424922492924992494924492424922492924992494924492424922492924992494924492424e5
7, 0xc0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000101ff
6, 0xd24949244924249224929249924949244924249224929249924949244924249224929249924949244924249224929249924949244924249224929249924949d7
4, 0xf6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdbc1
4, 0xdb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6c6e23
4, 0xedb6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b867
3, 0xd0840842421021080842842121081084842142101084084242102108084284212108108484214210108408424210210808428421210810848421421010840985
2, 0xe000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000f
2, 0xf5ad5ad6d6b56b5a5ad6ad6b6b5ab5adad6bd6b5b5ad5ad6d6b56b5a5ad6ad6b6b5ab5adad6bd6b5b5ad5ad6d6b56b5a5ad6ad6b6b5ab5adad6bd6b5b5ad5d39
2, 0xc28550a128500a14850aa14250a114280a144285a14228501428850a428550a128500a14850aa14250a114280a144285a14228501428850a428550a128500a6f
2, 0xfdefdef7f7bd7bdedef7ef7b7bdebdefef7bf7bdbdefdef7f7bd7bdedef7ef7b7bdebdefef7bf7bdbdefdef7f7bd7bdedef7ef7b7bdebdefef7bf7bdbdefe0b1
2, 0xd2494924492424922492924992494924492424922492924992494924492424922492924992494924492424922492924992494924492424922492924992484a0f
2, 0xe94a94a5a529529494a54a525294294a4a52a529294a94a5a529529494a54a525294294a4a52a529294a94a5a529529494a54a525294294a4a52a529294b9af5
2, 0xdb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d7015
2, 0xca52a529294a94a5a529529494a54a525294294a4a52a529294a94a5a529529494a54a525294294a4a52a529294a94a5a529529494a54a525294294a4a52a601
2, 0xc00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000002030b
2, 0xd8c68c636318318c8c63c631318c18c6c631631818c68c636318318c8c63c631318c18c6c631631818c68c636318318c8c63c631318c18c6c631631818c69107
2, 0xf18c18c6c631631818c68c636318318c8c63c631318c18c6c631631818c68c636318318c8c63c631318c18c6c631631818c68c636318318c8c63c631318c1907
2, 0xf7bd7bdedef7ef7b7bdebdefef7bf7bdbdefdef7f7bd7bdedef7ef7b7bdebdefef7bf7bdbdefdef7f7bd7bdedef7ef7b7bdebdefef7bf7bdbdefdef7f7bd8289
2, 0xc42142101084084242102108084284212108108484214210108408424210210808428421210810848421421010840842421021080842842121081084842143 69
2, 0xef7bf7bdbdefdef7f7bd7bdedef7ef7b7bdebdefef7bf7bdbdefdef7f7bd7bdedef7ef7b7bdebdefef7bf7bdbdefdef7f7bd7bdedef7ef7b7bdebdefef7bf969
1, 0xd4e682e94f1d6018a02056c0db850a74b3591b0f840514ce4017b2f5d25925ba2429a66e384b5be96e6a0a03d4a11eba10416018de3b3e354477250037b6f813
1, 0xcac05be5c1eabf0c21f8e95ce5d3c0777904282d1fd0c1738d727e197a0a32fda4cc59cc50b99d29f7fa8d07c972402ab88573e255db6bab05505812c73c2911
1, 0xcf052499061243cd82cd1b2059446c963487834d929ac929d92b259245254c7828ed3e92259292c924d24947d4896d1545f4001029b3b265d0ea4d144e242dbd
1, 0xfa94a972e2dcff068ee1257e228b53e9b9fcf46877f07daaa4d13c2bedf132d07730f549f4691f68553f84be8ff405f16a663d8fb8f82987bd9e073a8108edc3
1, 0xef7befbdbdef9ef6f7bd7bde9ef7ef7b7bdd9dcfef7b37bd9feddef7b7bd7bdedee6ef3b5bde3de7ed7bfa99adebdef7b7bd77d7cff1ee7b7bdebdeeef79f8ab
1, 0xeeb2919e1dc9ce33c2a0d9e190465b164a53c7c03e9a3d009ecf8fd6bdf743e04444332b7ff4a0e8f53b5123a5422563a06a487cd6cb5f36cd5411f0ae4dbc69
1, 0xf51576e530188d59bbc5f4f6ec9e824d7a9e70142952b11c49a6f38188ad9dbe3d29d1d9498b7aeffc4d9b0420f71895f62e2a7b79d4887e45b6227e0b84fb97
1, 0xd83f22a49af67d7f196df580d514464d6dbb880b03bea50ddcc1f931ef7f09af2f880de26d88cbf24567302a0d6eed7c8eab859aa0c1cc18bd8efacdce194c13
```

1, 0xc1df3e8db5f7b7f456edc1f60d23f60360536565836ce37af6f02e55de24a8dc373f3c5d49c93ba6fee0d44d08bc5fb0655781adee5c05777fd4da2bcd803d0f
1, 0xe279872638463a0a32a1412b13efccfa5ed68db44963c7f6955a3816bcaa33f94794c8b75298ddf4a8664e485ef99e6d9469f5187939e395cb1f09e666786741
1, 0xce73e73939ce9ce7e738739c9ce7ce73739c39cece73e73939ce9ce7e739739c9ce7ce73739c39cece73e73939ce9ce7e739739c9ce7ce73739c39cece73ead7
1, 0xd92ae5c6453efec55c5614207827de2b77bf3ef027f4230f8aac1fd9b0d69fdc61934132766f8dd1d8cb22ec38d834037eff6d9dd3535b9e582fbdd2327c9ce5
1, 0xc0800000000000000000000000000000000000000000000000000002000000000000000000000000000000000000000000000000000000000000000000001003f
1, 0xfff7fffffefffffffffffffffffffffe7ffffff7fffffffffffffffffffffd7ffffffbffffff7bfffbfcffff7fffffffffbf00000000000000000000000000c1
1, 0xeb6f80ff65b4a6d462cfa5961f542f25e207667752b0482f5ac9dc091f4dc854de9c73b288aaa5da5298a33928f7b2920f89b81e3635932bc9db99a34e52b82b
1, 0xfdf7b9bffbffdebeb28592b76f69bbffbffdafaeffd9f7bdf1ee7bfa6e2f33bb67d5a5b5676d2bf6a1de3626f06be367ffde73db1e01f5d3855f21f0eda8b4db
1, 0xe643203b22b4048427210bd390d45a3a62ac132c0063990067686123d50128812e09411f27098400c841e09183400431018100a2b1cc0954c0405026420e8c7f
1, 0xffefef7ffde6ffff7fffffbfffffffbffeffbffdfffffffffffffffffffffffffffffff1fcffffef46fffdfdfff7fffffffffffffffffffffffeefefeceffe8d
1, 0xf6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66b37b6db019a4697
1, 0xc000b8000000000000000000000000000000000000000000000000000000000000000680000000000000000000000000000000000000000000000000000000251
1, 0xccc5ebfea2f4beb8b62dfef5429f97f06af0af8d08159d21df4540a0197ffdb8386c8ebb18bd70b0f46c9615d2fcd0ea38a2cadb522cf79f2c3ab27d9564a197
1, 0xedb6b6dbb6dbdb6ddb6d6db66db6b6dbb6dbdb6ddb6d6db66db6b6dbb6dadb6d9b6f6db66da6b6fbf6cb9b7ddb656d9e6d36a7dbb673ba6ddb6f6db66df6b5e5
1, 0xe7fa15ab6c3d2c3d13960f598cd2bbf74a688580e5fdc70064563a10558f1dfd36d5e8aec88897c79d73ebdcbec1b5f0121175c8aae69e3a31a63f9e66e0bfc5
1, 0xffb308867fee16267feb2b1af212ffefffffe4308866fff5fffefe13ffcf869aff4bf907ff1f9393fff0fff3fffcfff7ff3ef703ffaa8c7ffffe491affeff3b1
1, 0xc010208a48c18021210810848421423010a4084242006309ca468d2123081084a520431000c40a425210210a084a8ce1290810cc84204a9011ac2842401022e1
1, 0xe739729c9ce7ce73539c29cec126e7383b8e89bd2207faed08428421318c1084c410631858c68c63e31035cc8c63ce31318810c64331231818c60e63623b32a3
1, 0xfeb1b9efa29f64ed53628a10a924b5268163dd887f653a6b82edb063b6874c2039e4938018ab949a3c28cdc785fe2be58872c0c8a9ec5171e37ea6a82d5d46d7
1, 0xc010000000000000000000000000900000426040 0c00000000001800000000000402080000000000202000010800000000000000000000000000280000000002f3
1, 0xf9d5834f918b673e1f7eaae3cc5d97dd2706dd8de9c5b2fbef679b2c196933fe30f62ac3f7fcc1c593fb63a0bbb8838b8486eac959cc3949ea9182c46396fbcb
1, 0xdac45d37aadacfec73b3184ef43d52d6314754abd38414dde03ade396bd809aa2811047f015c9c71f0cbb0a91028190adeacc36165b0e0e6fce64549f947e0d5
1, 0xf49808713746a41a331625a7cb389611eaa3905984245f99e828f17f867413cfae91230478715024db5ead44beb20fbc73a23a271d627a11747b5823f753eb03
1, 0xd67a7b111c0401971f57806a2be12a174b8923fd3972ec64fe3de3ee96594a14207831d12f16f545851cad6356bb16221bee68eb2fee9427e0da0ca5f98e5861
1, 0xe83071df5288c373a5bc43fb20309e25e99fd85b61a9a4e6f3f71511b98f7ec87047fb32520d94cd7753dbe173304445ca648231f601dd19d3cd40c74190c71d
1, 0xed4294b5a529529c94250ad35394214a4a52a569a94a94a5e56b52948ca74a52529429524a5aa529294a9ca5e1295294d4a74a727394696a4a13a529236a968d
1, 0xd621eb6e5ab7992c6efba5f34a7b7b28026fc93138998c113831dbaaaca1a15738a7b7a9d191bcd77955b92b75263ad9f6bbd4ce0b4edca1efd5f3e24b3a2889
1, 0xd9a43ff058df6b8d55085028eac413a7439e1dc89e5d6e8b5de09e7bc7483d762788ff9e36527ff67c39360cfc0d2a75986b7fb35614027cffb932ee1112ee8d
1, 0xe492924992494924492524922492924992494924492424922492924992494924492424922492924992494925492424922492924992494925492524922492938f
1, 0xf9cf9b29d767edb655b2f6bf964bce697f652fb669b322eb63dffb6e7a6c69bb798396d284d85169883d42a6ec96b292761d6dcd7ab595b2ad0a9a5d7e97fe41
1, 0xfffefffefffffffffffefffefffefffefffffffefffffffefffefffefffffffffffefffefffefffefffeffffffffffff000000000000000000000000000000bd
1, 0xf9ce9ce7e738738c9ce7ce73739cb9cece73e738398e9ce7e719739c9ce7ce7373dc39cece73e53839ce9ce7e7b9739c9ce7ce73739c39cece73e73839ce9d63
1, 0xd53bd2f169ab7fb38abb7f05cb1550e200914674b65ce176001ffeb29dbd1e90c21a77e28c6dbfd6e6a782baaba532e2a98eff9ed8e924986af702c48504d0d1
1, 0xc36e8f2addb602d9d18b2b040bc7a00bc7046b2030c2d3e91c4c161ed562a31d2d056afc759042a46c28e218e25e7c7882fb1cb2d66039ed961dace5ea69c5d7
1, 0xed15cb0fde1567b278ef2422ee01ed658173594b0bcb71594a18df455fc75ca7c5b529bb6b9ec229be6ba977773eca917ac08a1e9f557adf079ab8bceb2bc01b
1, 0xd00b0dd78fd35c88db31806803799deab89b8b36c39dc0321574801fb936f90e2920f3dd65400ddc00be90ebcefdd62d5c5c062c200bdb04aa6a5acf697e2a0d
1, 0xd0054c94020831e800450e05811840282088a906825002d9a0c340938dc0b20628072f800334102c08010309c020800710200c04a604083700aa440088411987
1, 0xc7592d7dc9ee1031dcd3d30f43028858305ac46ac981cafa164a8000a9c6eeb698181505242ac9dfee9e51c92460b987dbc8161def71863d35ac18fa1235a903
1, 0xfffbfcf7f7ffdf3dfffef5fffffbfffff9ffffffffdff7dfffffff9ffffffef7dffffffffbfffdfffeffbffffffffffffff77fffdfffffffffff1fffffffff35
1, 0xf5eb05d73ad4df3cdaf4fd2eaf41e8e405952b7a327479147fffa33eb829039e77ff116f9e4958a3f604743ed2c55ba67b47631842905dbc2f12c66fb6c4e40f
1, 0xc3ff4d30474f40df0e7ffffdfa92ffff11d59d35d214ffff85c357c5c85ed72acaf1fb7d43f76d85ee6b4fb3ffdd60d5095ef1f290df4ff888e7e37efe4f9e8f
1, 0xcc7b18295347824ccb395bed351993c598c7cf7f4e32dcb9ab7a5d7e0baa7626d1b8dc651b34f5e4f5d3f2530b52fb9bd10e75259b36d774f059141bf9ede911
1, 0xe675a7059b1e6df20198f8a75a0ab28123fff79a67f59c7049fd37d48128f3b3a9b69475b902f4bc854ca1deecbce73cdab89b17ae3c6401a9d43594775a926b
1, 0xef7b77bd3defdef7f7b47bdedef76f7b7bdebdefef7bf7bdbdee7ef7b73d7b9edef7ef7b7b9e3deeef7bf7b9bdefd6ffffb97bdedef7ef7b7b9ebdafef3bf845
1, 0xfd23b110962000d598488c43407369898cd0086df780826dcfa14784f38388874362851b7711dc13564441351335c71fbd7c564d5d5008f5de20d43f2476d715
1, 0xe918f1658790911a71a9ae1895cfe56dbed767816e337e2f950462affb3280d8a8dcb1240620ec8f1d19c3750afcfe295c58cca117b36632414cd9e114fdb097
1, 0xffffaa55fffffffff3cd9fe3ffff676ffffffffffe00000000000000000000000000000000000000000000000000000000000000000000000000000000000000009d
1, 0xc02100004804100000100010008001ce2064004242c812186250154c00000088ba78008a43a9713bc0abb849220e8362cc838b53cf88fcdbdd7fca83c8df8145
1, 0xe318318c8c63c631318c18c6c631631818c68c636318318c8c63c631318c18c6c631631818c68c636318318c8c63c631318c18c6c631631818c68c63631831d3
1, 0xd501973162d4017f4e3b3c9d6803d4cc46a1d457c91feb5b6c2ae77423ba41c9cfbd5f4b9235667874507e9cafb4123e1992d1c5ae75ee295087011a822a6ccf
1, 0xe28ecce1de7a0326423076465160c1b03f8e721181e046ef4860ae94d7802a082f9f6007c0011f20056de200677aa7d8a47118e6692ee4b3f862c24e04b543b5
1, 0xda2f36d74bc2dc29de4de92f4b37b03942173e15a2dfb67e8f09e790ed1656af5a8aadef14b696426f1e929699da0ee3ad9f21a9f66ede57d945fc165b27d217
1, 0xd28550a3a8520a1c850aa14250a114ba0a144285a14228501428850a428550e128400a14850aa14250a114080a144285a14228511428852a4685d0a128500a2d
1, 0xe79082499b094b2459266493608a9249b2410d3409242692a490824d934941254935265a341086119449d824691524922697926bb24949044027108a8c939a5d
1, 0xedfc373f783ffbfff7fefed3fffafffefffd5fffffdefff5ffdffff3ff5ffffefefd3fef7fbf5dfff613bb59f9fb5f5bd52aefd78ebddfe6edeeffe3f3fb3df5
1, 0xfff71fb6fbfffefffffffeeffffeefffef7fffebfffefffffffffefff9fffffcbfdbff0fafffdfff7f7edffee7adfffafffbfb7effffdffde7fadfdef63e806b
1, 0xcaf67d473c10f4e73d6678d4a27e4eb04a743925d12c31f97efa510ca68558b2c56d839acecbe75e935f86cec7dae7c95aa0b93065a3aa924594fdfb9f521535
1, 0xf6b43e3bd52841756d1a27f22a8590a8a1c43c1c36b95cc72d0102f26b6da1b238236856f7c6e6faa83cc70e84f2db44088487fd94a175f22a0d990cc1afea6b
1, 0xd2a20d1b986de2152b9d93cf60bf98f68e9f9e050feb9820b006e5dc581f17a82f35a78d23fb34fab3962ae95bcf3a1e442eb5b1d72cd6956fa599483eee38c1
1, 0xe52e529494a54a535294294a4a51a509294a96a5a529529494a54a52529d29ea4a52a522298a94a5a522529494a44a525294294a4842a52e294af4a5a52f554b
1, 0xc942c4644b1169461581e0713ba400570237a55c9ae69e3fe58d189aa751d218208421934f2132a888e796bc1f0914a8c9b4f116358cca22c69c35596bd961e5
1, 0xed7f7e78afc7d3735fc1dfb0d13887cddcd715c9fe530530e0efceaa4bcaffbaebac9e601623db36fffef47fffffefffffff00000000000000000000000003ed
1, 0xf6db9b6dda6d29b61dfe73dbba5bdb6ddead69beedf6a6dbf7dadb6ddf6c6cb66db6f6d3b6db9b64997c6dbe6cb4364b96dbdb6ddb6c67be6da4b7cbaedadf35
1, 0xc080a10000000000000000000000000000000000000000000030080008800000026080200101200044080010042020800008000000500000001000000000900e1
1, 0xe5335f76a97c5e29d4557170cd9ef3ed53efc819fda87a566a5efe247ef102b85c7ad90c484ade030c7ebc23455e0dcbca2cec6afdf0e8c978cb6fbed5733fa5
1, 0xc0000000000000000000000000000000000000000000000000000000004000000000000000000000000000000000000400000000000000000000000000000165
1, 0xc924249324929249b2cf49244924649264921a4892494936593320b93f9292e992497d14492424922292934992494924492624932 48e96fff3c9104432f4cdbb
1, 0xc924249224929249924949244924249224929249924949244924249224929249924949244924249224929249924949244924249224929249924949244 9342b29
1, 0xc0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100000177
1, 0xc000004100240000508000802100004900000010000000208000000000010210051800000000000000002100806000400820004000001830010000002000 20f1
1, 0xcad0ca7166b2aaf6c82b0eadfeb13409da7c2679517d4fd96f89719659133e0492d209da600753dc5c2570ce128cf985332f944143204b706bf6e990c0e43dcb
1, 0xe739739c9ce7ce73739c39cece73e73939ce9ce7e739739c9ce7ce73739c39cece73e73939ce9ce7e739739c9ce7ce73739c39cece73e73939ce9ce7e73973df
1, 0xe0929249924949244924249a249292499249492449242493249292499a494924493424922492924992494924492424922492b24992494924492424922482926f
1, 0xc0001000000000000000000000000000000000000000000000000000000000000000100802000cb0040908809180008c8000000010c00012101b20000000002ad
1, 0xcec727009ef07418dc89e2c96e796d44bc2244d88a0bb8ca90b4d661736b486b6e1d8352822a4697cdd0702a3d8b7c4b23ada2285a2af09234a71346ba141795
1, 0xc8dce72c0e38ecaf2e3e11aef07326e3431a92ad87ef296d3d0b5d4b3d00646bebd7b3af6c9e424e074e1486d186d26997a4d9c131acb524881aecace287c057
1, 0xee09f0be62014c7299e188527ab8cd004809c631f1fd50a20013331678ccad20631879842b8a122569eb18c4b1dd5e4b11bce7a14f4ae76973debf4ca768c4bd
1, 0xd66cdaeef275bfd3d1ee65df430dd7ae015bd0e9a5e43890e7835e2a2a0fb702703d6c3fd50d5917f3ba77aeb851c016d26135d754c114adf303d091500462bd
1, 0xfa147ea58cddaaabe6dfa04ff891009db3ff37e1272d573b7a3da5334f24f9512fda7ff4f163a72482a0edffa9140001aae21f5a64fd330f93e819a968acafb7
1, 0xf718c0bc8c57cc318c99fa15236191a531828a95856d6ac833a7e3a2110dded25226ea4344cabbb2fe19de14863b8c46e31b44038c87e8ce4aea42a10afabf91
1, 0xd86d99e183ba3c0870238db37f1d3f673cdec3112196acfaa1239657bcb3a3a7f6749f3229f550d5097510e5a5df0626a641e2112112f95080c5629973b1c975
1, 0xfc1fc95ee7482142bccb7f0bc5cd674ad82edca61fe2653c78622ee673485cc11c993aaeeb15f77d90dfe1c6a945e239ab47e5ca3eb2aeb702f2de36626858db
1, 0xf7c6bf218fcfadcba926ac5efdf60f97aeba8d5f70ceb27eff0f5d57e763bfe86dc7a86ee76b8ba9d076bf1a8f4a7fcfb0297a96c6c5a70ea7e5e3c38326ff83
1, 0xc594391e8e8c24c8a7fe971d78db784d43c96ba3384f02acf71fc2506736c65f7c44ef6c3bbf7b05659b954c6b9ce96f648c900b56c5f3ca01e47384ad4de577
1, 0xda975410693d3120b32997c8c728f09d09610f5fef089a7cf63ff1dcf673ffffb493c19c64167e0457646aaba4f3409f9648ff7c390c25d4a8a3d7c9b2f16b2d
1, 0xf16fead9af03cfcb36571b8b3fe3cf24e313aece858b7d4e800838329c9b729ecc6d691df4ee8547a9fdb18debbca338af8214fa1e03ad53f8e3a0503bfb6735
1, 0xddf3b56a7bb556afa1476addb54aa95e569c94ab62d5fa95c054af04b5a3b56adff55e2dbb466ed1b56aad5a1629c5a93ad55bf5ad1e3ba4e5ab9722daf5d7bd
1, 0xd9958fe30334b89c8c02ac210c4dc8e6e610d1c958cb4d436e11aede0f72e3b8a88e18b7c663533218c68ed560b031ad4ce38aa13bbc10b6c73fe3911acc8de1
1, 0xfcb0663ef5e3c922936834039fd787a0de9fdd178017021129cfb592570fd3c5e60787fc59128bce5bfcb38be0c064b08c087fd8fe6b960207c93ca4cf3c5add