Shmoocon 2011 Crypto Challenge Pack

Dcoder

1 Introduction

2011 brings us a new cryptographic challenge pack by andrewl [1]. This time around, the source code of all the $5+1$ problems is given (Python), which speeds up the process and lowers the entry barrier. The difficulty of the problems ranges from easy to very hard. So let's begin at the beginning!

2 Warming up

The first challenge, `warmup.py`, presents us with an interesting problem. Given a 10-character (either truncated or padded) user name and a serial, 10 strings are "rotated" until the first character of each rotated string is equal to each character of the user name.

Let's take a closer look at the rotation:

```
def rol_string(s, amount):
  width = len(s)
  # rotating len(s) is identity op, so reduce the work
  amount = amount % width
  for i in range(amount):
    s = s[1:width] + s[0]
  return(s)
```

It becomes obvious that the rotation operation is cyclic, i.e., that there is a finite amount of possible rotation amounts that result in different strings. Indeed, the challenge itself reduces this amount modulo the number of different possible strings, `len(s)`.

Back to the challenge. Now we we know that the serial is the amount of rotation needed for each serial modulo the length of each "rotor" string. There is one trivial solution: as long as each rotation amount is lower than the minimum string's length, we can just use that. But the challenge prevents that:

```
if name == check and serial > 139 and serial < 421336842070675358939:
  print "good"
else:
  print "bad"
```

Thus, we need to solve a modular system of equations to obtain the desired serial. First, let's formally define the problem. Let $c_0, \ldots, c_9$ be the characters of the user name, $l_0, \ldots, l_9$ and L be the lengths of each rotor string and their product and $r_0, \ldots, r_9$ the rotation amounts required. Each one of these arrays is trivial to find. The problem is to find the value S such that

$$S \bmod l_i = r_i, \quad i \in \{0, 1, \ldots, 9\}.$$

The solution to this problem is extremely well known and usually named Chinese remainder theorem (CRT) [9, §8.1]. Using the above notation, our solution is given by

$$S = \left(\sum_{i=0}^{10} r_i v_i L_i \right) \bmod L,$$

where L_i and v_i are the precomputed constants L/l_i and $L_i^{-1} \bmod l_i$. To generate valid serials, we can use the following SAGE [20] code:

```
sage: name = "Dcoder"
sage: if len(name) > 10:
....:   name = name[0:10]
sage: else:
....:   while len(name) < 10:
....:     name = name + 'S'
sage:
sage: rotors = [rotor0,rotor1,rotor2,rotor3,rotor4,
....:           rotor5,rotor6,rotor7,rotor8,rotor9]
sage:
sage: rot = range(10)
sage: for i in range(10):
....:   rot[i] = Integer(rotors[i].find(name[i]))
sage:
sage: serial = crt(rot, [Integer(len(rotors[i])) for i in range(10)])
sage: print serial
```

That's about it for the warmup challenge. A simple problem put in an unusual way.

3 Challenge 1: Euler's totient

The first real challenge from this pack, `crypto1.py` is even shorter than the warmup. It is a classic RSA scheme[1] with a twist: the RSA modulus is a product of 8 distinct primes:

```
sage: n = 1821668788150059822966422307930288186476927434330004078983325993614616857
sage: factor(n)
286331173 * 572662309 * 858993503 * 1145324633 * 1431655777 * 1717986953 *
2004318077 * 2290649227
```

While multi-prime RSA is not very common, it is certainly not unknown [4]. The existence of multi-prime variants relies on the existence of Euler's totient function, $\phi(n)$, for every positive integer. This is the case [9, §5.5]. It is given by

$$\phi(n) = \phi(p_0^{e_0} p_1^{e_1} \ldots p_r^{e_r}) = \phi(p_0^{e_0})\phi(p_1^{e_1}) \ldots \phi(p_r^{e_r}) = \prod_i (p_i - 1)p_i^{e_i - 1}.$$

In our case, since there are no prime powers, $\phi(n)$ is even easier to compute: $\prod_i^r (p_i - 1)$. After this, all is missing is to compute a modular inverse and we're set:

[1]If you do not know RSA by now, check one of the usual references, e.g., [6, §31.5] or [11, §4.5.4]

```
sage: d = 65537
sage: n = 1821668788150059822966422307930288186476927434330004078983325993614616857
sage: name = "Dcoder"
sage: m = 0;
sage: for char in name:
....:   m = m*256 + ord(char);
sage: m = m % n;
sage: phi = euler_phi(n)
sage: e = inverse_mod(d, phi)
sage: c = pow(m,e,n)
sage: print c
```

4 Challenge 2: Tap dat s

Challenge 2 moves on to a different ring: the integers modulo 2. In this challenge, the user's name is converted to a sequence of bits[2], which are then processed by a linear feedback shift register (LFSR) [2, §41.1][12, §8.1].

It is unclear what the hard problem to solve is in this challenge; we are given the connection polynomial ($x^{64} + x^4 + x^3 + x + 1$), as well as the starting and end sequences (user name and serial respectively). Generating a valid serial is trivial:

```
name = "Dcoder"
name_num = 0
for char in name:
  name_num = name_num*256 + ord(char)

name_num &= 2**64-1
state = name_num
for i in range(65537):
  state = (state >> 1) | ( (state&1) ^ (state >> 1)&1 ^ (state >> 3)&1 ^
          (state >> 4)&1 ^ 1) << 63

print state
```

As you can see, all we are doing is directly inverting the LFSR, which is easy as there's no extra data being injected into the state. Mathematically speaking, we are simply turning the linear recurrence

$$s_n = s_{n-1} + s_{n-3} + s_{n-4} + s_{n-63} + 1$$

into[3]

$$s_{n-1} = s_n - s_{n-3} - s_{n-4} - s_{n-63} - 1.$$

5 Challenge 3: Group hug

Challenge 3 is quite peculiar. We are given two tables (`sbox0` and `sbox1`) of 358 elements each, seemingly without much arithmetic meaning (is there one?), and we have two

[2]Semantically; all arithmetic is still done in the integers.
[3]Remember that in $\mathbb{F}_2$, $a + b = a - b$.

operations defined on them: “multiplication” and “exponentiation”. The problem here is to find a serial s such that, given a name n,

$$sbox_0^n = sbox_1^s.$$

The obvious way to solve this is to represent $sbox_0$ as $sbox_1^x$, for a yet unknown x. Then, $s = xn$. We can do this by computing individual orders and logarithms for each element and combining them with the Chinese remainder theorem. But there is a more elegant way.

Every possible finite group of n elements can be thought of as a subgroup of the symmetric group on $\{1, 2, \ldots, n\}$, i.e., S_n, the set of all possible n-permutations (or bijections) under function composition [17, Theorem 3.12]. Thus, we can think of $sbox_0$ and $sbox_1$ as elements (permutations) of S_{358}. They share the same order and cycle structure, which means they are conjugable, i.e., they are equivalent up to a reshuffling. The order of both points, as would be expectable from a divisor of a small permutation group of order 358!, is highly smooth. Thus, generic Pohlig-Hellman [15] and Baby-step Giant-step [18] as implemented in SAGE are enough to solve the logarithm:

```
sage: G = SymmetricGroup(358)
sage: s0 = G([x+1 for x in sbox0])
sage: s1 = G([x+1 for x in sbox1])
sage: s0.order()
18446693477654742300
sage: s1.order()
18446693477654742300
sage: discrete_log(s0,s1,s1.order())
1449402469971139457
```

Once we know this value, generating a valid serial is easy:

```
name = "Dcoder"
order = 18446693477654742300
name_num = 0;
for char in name:
  name_num = name_num*256 + ord(char);
serial = (1449402469971139457 * name_num)%order
print serial
```

6 Challenge 4: Curves and roundabouts

Challenge 4 is very interesting, as it is not at all clear what’s going on initially. It can be solved in two distinct ways: the simple one and the hard one. Let’s start with the hard one.

6.1 Option 1: Lies, damn lies, and polynomials

At first glance, the `encrypt` function appears simply as a large rational function evaluation over $\mathbb{F}_p$:

$$E(x,y) = \left(\frac{f_0(x)}{g_0(x)}, y\frac{f_1(x)}{g_1(x)}\right), \quad f_0, f_1, g_0, g_1 \in \mathbb{F}_p[X].$$

What is the meaning of this? Truth be told, we don't need to know. Computing polynomial roots over finite fields is known to be "fast" [7, §2.3.3][12, §4.3], so we can just calculate solutions on the polynomials and ignore their meaning altogether.

Solving rational functions is easy. Given a rational function $\frac{f(x)}{g(x)} = b$, we can solve for b by finding the roots of the polynomial $f(x) - bg(x)$. The second component of the serial doesn't require finding any roots, just computing a modular inverse. The following listing details how to do it in SAGE (polynomials are omitted, as they would take up considerable space):

```
sage: name = "Dcoder"
sage: name_x = 0
sage: for char in name:
....:   name_x = name_x*256 + ord(char)
sage: while 1:
....:   quadres = (name_x**3 + 3)%p
....:   name_y = pow(quadres, (p+1)/4, p)
....:   if pow(name_y, 2, p) == quadres:
....:     break;
....:   name_x += 1
sage:
sage: serial_x = (f0 - name_x*g0).roots()[0][0]
sage: serial_y = name_y / (f1(serial_x)/(g1(serial_x)))
sage: print "%d-%d" % (serial_x, serial_y)
```

6.2 Option 2: Occam's razor

OK. We might have found a way to find valid solutions. But is there any remaining meaning to this? Is all the elliptic curve innuendo simply misdirection? No.

Multiplication of a point $P \in E(\mathbb{F}_q)$ by a constant m can be given in terms of division polynomials [19, Exercise 3.7]. The mth division polynomial $\psi_m \in \mathbb{Z}[A, B, x, y]$ of the curve $y^2 = x^3 + Ax + B$ is defined by the recursion

$$\begin{aligned}
\psi_1 &= 1, \\
\psi_2 &= 2y, \\
\psi_3 &= 3x^4 + 6Ax^2 + 12Bx + A^2, \\
\psi_4 &= 4y(x^6 + 5Ax^4 + 20Bx^3 - 5A^2x^2 - 4ABx - 8B^2 - A^3, \\
&\dots \\
\psi_{2m+1} &= \psi_{m+2}\psi_m^3 - \psi_{m-1}\psi_{m+1}^3, \qquad \text{for } m \geq 2 \\
\psi_2\psi_{2m} &= \psi_{m-1}^2\psi_m\psi_{m+2} - \psi_{m-2}\psi_m\psi_{m+1}^2, \qquad \text{for } m \geq 3.
\end{aligned}$$

To achieve a multiplication-by-m rational map, we need to define two additional polynomials:

$$\phi_m = x\psi_m^2 - \psi_{m+1}\psi_{m-1},$$
$$4y\omega_m = \psi_{m-1}^2\psi_{m+2} + \psi_{m-2}\psi_{m+1}^2.$$

Then, multiplication of $P = (x, y)$ by m can be given by

$$mP = \left(\frac{\phi_m(x,y)}{\psi_m(x,y)^2}, \frac{\omega_m(x,y)}{\psi_m(x,y)^3}\right).$$

The degree of $\phi_m(x, y)$ is known to be m^2 [19, Proposition 5.4]. Since the degree of f_0 is $169 = 13^2$, we can take a good guess that the rational function we are given is the multiplication-by-13 map. We can confirm this in SAGE by multiplying the initial point by 13 or by recreating the map:

```
sage: p = 6277101735386680763835789423207666416102355444459739541047
sage: E = EllipticCurve(GF(p),[0,3])
sage: E.multiplication_by_m(13)
```

Inverting multiplication by 13 is simply a matter of multiplying by its inverse modulo the order of E, 6277101735386680763835789423061264271957123915200845512077. Generating a serial becomes simple:

```
sage: serial = E(P1)*inverse_mod(13, order)
sage: print serial
```

7 Challenge 5: Ignorance is BLS

The fifth challenge is definitely the hardest of the bunch. Luckily, the source is riddled with hints, clues and downright giveaways that help us considerably in figuring out what's going on. This challenge brings us into the world of pairings.

7.1 Pairings

There is much to say about pairings and their background. It is impossible for this small text to serve as an introduction; I'll defer to good references such as [13, 5, 19, 21]. In short, a pairing is a map

$$e : G_1 \times G_2 \to G_T,$$

where G_1 and G_2 are additive groups and G_T is a multiplicative group, all of prime order p. The pairing is called symmetric if $G_1 = G_2$. Let $P \in G_1$ and $Q \in G_2$. In a pairing, the following conditions hold:

Bilinearity For any $a, b \in \mathbb{Z}_p^*$, $e(aP, bQ) = e(P, Q)^{ab}$.

Non-degeneracy $e(P, Q) \neq 1$.

Computability $e(P, Q)$ can be efficiently computed[4].

[4]Not in Python, it can't.

The Weil pairing [22] employed in this challenge is the symmetric map

$$e : E(\mathbb{F}_{p^k})[n] \times E(\mathbb{F}_{p^k})[n] \to \mu_n,$$

where $E(\mathbb{F}_{p^k})[n]$, i.e., the subgroup of elements of $E(\mathbb{F}_{p^k})$ (points) whose order divides n. It results in μ_n, the group of nth roots of unity[5] of $\mathbb{F}_{p^k}$.

The properties of bilinear pairings raise some interesting possibilities. The BLS signature scheme [3] is one of the most noteworthy schemes resultant of bilinear pairings. BLS works as follows:

Select parameters Fix groups G_1, G_2, G_T of prime order p. Let $P \in G_1$ and $Q \in G_2$ be generators of their respective groups. Let ψ be an isomorphism from G_2 to G_1. Select an appropriate pairing e.

Key generation Choose random private key $x \in \mathbb{F}_p^*$. Compute $V = xQ$. $V \in G_2$ is the public key.

Signing Let m be a message. Let $R = H(m) \in G_1$ and $\Sigma = xR$. $\Sigma \in G_1$ is now the signature for m.

Verification Let m be a message, $R = H(m)$ and Σ its signature. Accept the signature only if $e(Q, \Sigma) = e(V, R)$.

Note that the safety of BLS is dependent on the hardness of the discrete logarithm on either G_1, G_2 or G_T. It can be attacked in G_2 by definition, since there is where the public key is defined. It can also be attacked in G_1 since, by having one valid signature, we know R and xR. Finally it can be attacked in G_T since $e(Q, xR) = e(Q, R)^x$.

7.2 Realization

In Challenge 5, the BLS scheme is realized as follows:

- $p = 9524793152874449521, n = 9524793149788155121$.
- $f = 6927354994984596761 + 3748481628317431011a^1 + 4620394961492627319a^2 + 5139526688006885996a^3 + 7434977584397438745a^4 + 5869627458528271108a^5 + 6614627009312766654a^6 + 7678097075714978717a^7 + 5204744523362322329a^8 + 9398406610703021891a^9 + 3699077701165988134a^{10} + 4097988535177883106a^{11} + a^{12} \in \mathbb{F}_p[a]$.
- $G_1 = E(\mathbb{F}_p)[n]$.
- $G_2 = E(\mathbb{F}_p[a]/f)[n]$.
- $G_T = \mu_n = \{x \in \mathbb{F}_p[a]/f | x^n = 1\}$.
- ψ is achieved by using the trace map.
- $H(m) = mP$.

[5]An nth root of unity is an element a such that $a^n = 1$.

- e is the Weil pairing.

Now that we have all the knowledge we need, how do we break this scheme? We can solve in G_T, a 757-bit discrete logarithm where the number field sieve has already reached [10], but at a significant cost. We can solve in G_2, which admits index calculus [8], but at a large cost. Finally, we can solve it in G_1, which has the least cardinality and element size of all three options.

There are two ways in which we can map the discrete logarithm to G_1. The easiest is to take advantage of one of the example serials, which gives us the points in G_1 quite explicitly. The other way is to map the logarithm in G_2 to G_1 using the trace map:

$$\psi(x, y) = \sum_{i=0}^{11} (x^{pi}, y^{pi}).$$

I solved this logarithm in G_1 using Wiener and Oorschot's version of Pollard's Rho method [14, 16] in about 1:20 hours[6]. This yielded the solution $x = 1223334444333221111$. Once we have the logarithm, generating a serial is straightforward (cf. 7.1):

```
sage: x = 1223334444333221111
sage: serial = g1 * (name_c*x)
sage: print serial
```

8 Final remarks

Both solving these challenges and writing this document has been great fun. This was a pack that did bring a lot of different constructs into the table, which made it quite challenging. I sincerely hope that next year brings more of it.

Special thanks to andrewl for making such imaginative and tricky challenges.

References

[1] andrewl: *Shmoocon 2011 Crypto Challenge Pack*. `http://crackmes.de/users/andrewl.us/shmoocon_2011_crypto_challenge_pack/`, January 2011.

[2] Arndt, Jörg: *Matters Computational: ideas, algorithms, source code*. Springer, 1st edition, 2011. `http://www.jjj.de/fxt/fxtpage.html#fxtbook`.

[3] Boneh, Dan, Ben Lynn, and Hovav Shacham: *Short signatures from the weil pairing*. J. Cryptol., 17:297–319, September 2004, ISSN 0933-2790. `http://portal.acm.org/citation.cfm?id=1028473.1028478`.

[4] Boneh, Dan and Hovav Shacham: *Fast Variants of RSA*. CryptoBytes, 5:1–9, 2002.

[5] Cohen, Henri and Gerhard Frey (editors): *Handbook of elliptic and hyperelliptic curve cryptography*. CRC Press, 2005, ISBN 1–58488–518–1.

[6]Note that we could have solved the logarithm in either group using this (generic) method. However, G_1 has, by far, the fastest arithmetic.

[6] Cormen, Thomas H., Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson: *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001, ISBN 0070131511.

[7] Crandall, Richard and Carl Pomerance: *Prime numbers. A computational perspective*. Springer-Verlag, New York, 2001, ISBN 0–387–94777–9.

[8] Diem, Claus: *On the discrete logarithm problem in elliptic curves*. Compositio Mathematica, 147(1):75–104, January 2011. `http://www.math.uni-leipzig.de/~diem/preprints/english.html`.

[9] Hardy, G. H., E. M. Wright, and A. Wiles: *An introduction to the theory of numbers*. Oxford University Press, USA, 6th edition, 2008, ISBN 0199219869.

[10] Kleinjung, Thorsten, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman Te Riele, Andrey Timofeev, and Paul Zimmermann: *Factorization of a 768-bit RSA modulus*. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 333–350, Berlin, Heidelberg, 2010. Springer-Verlag, ISBN 3-642-14622-8, 978-3-642-14622-0. `http://portal.acm.org/citation.cfm?id=1881412.1881436`.

[11] Knuth, Donald E.: *The art of computer programming, volume 2: seminumerical algorithms*. Addison-Wesley, Reading, 3rd edition, 1997, ISBN 0–201–89684–2.

[12] Lidl, Rudolf and Harald Niederreiter: *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, second edition, 1997, ISBN 0-521-39231-4.

[13] Menezes, Alfred: *An introduction to pairing-based cryptography*. Volume 477 of *Contemporary Mathematics*, pages 47–65. AMS-RSME, 2009, ISBN 978-0-8218-3984-3.

[14] Oorschot, Paul C. van and Michael Wiener: *Parallel collision search with cryptanalytic applications*. Journal of Cryptology, 12:1–28, 1999, ISSN 0933–2790. `http://www.springerlink.com/content/g7r4wq6qvn5vcwb2/`.

[15] Pohlig, Stephen C. and Martin E. Hellman: *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance*. IEEE Transactions on Information Theory, 24:106–110, 1978, ISSN 0018–9448.

[16] Pollard, John M.: *Monte Carlo methods for index computation mod p*. Mathematics of Computation, 32:918–924, 1978, ISSN 0025–5718.

[17] Rotman, Joseph J.: *An Introduction to the Theory of Groups*. Allyn and Bacon, Inc., Newton, Massachusetts, 4th edition, 1995, ISBN 978-0387942858.

[18] Shanks, Daniel: *Class number, a theory of factorization, and genera*. In Lewis, Donald J. (editor): *1969 Number Theory Institute*, volume 20 of *Proceedings of Symposia in Pure Mathematics*, pages 415–440, Providence, Rhode Island, 1971. American Mathematical Society, ISBN 0–8218–1420–6.

[19] Silverman, Joseph H.: *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, 2nd edition, 2009, ISBN 978-0-387-09493-9.

[20] Stein, W. A. *et al.*: *Sage Mathematics Software (Version 4.6.1)*. The Sage Development Team, 2011. http://www.sagemath.org.

[21] Washington, Lawrence C.: *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman & Hall/CRC, 2008, ISBN 9781420071467.

[22] Weil, André: *Sur les fonctions algébriques à corps de constantes fini*. C. R. Acad. Sci. Paris, 210:592–594, 1940. French.

A Full solution code

```
import sys
import math

rotor0 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~1akuEOY"
rotor1 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~2blvFPZ9hqz"
rotor2 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~3cmwGQ10irAIQ"
rotor3 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~4dnxHR2ajsBJRY6cj"
rotor4 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~5eoyIS3bktCKSZ7dkqw"
rotor5 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~6fpzJT4cluDLT18elrxCHMR"
rotor6 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~7gqAKU5dmvEMU29fmsyDINSW159cgkoswAEIM"
rotor7 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~8hrBLV6enwFNV30gntzEJOTX260dhlptxBFJNQTWZ"
rotor8 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~9isCMW7foxGOW4ahouAFKPUY37aeimquyCGKORU"
         "X13579ac"
rotor9 = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ`abcd"
         "efghijklmnopqrstuvwxyz{|}~0jtDNX8gpyHPX5bipvBGLQVZ48bfjnrvzDHLPSV"
         "Y24680bdef"

def xgcd(u, v):
  u1 = 1
  u2 = 0
  u3 = u
  v1 = 0
  v2 = 1
  v3 = v
  while v3 != 0:
    q = u3 / v3
    t1 = u1 - q * v1
    t2 = u2 - q * v2
    t3 = u3 - q * v3
```

```
      u1 = v1
      u2 = v2
      u3 = v3
      v1 = t1
      v2 = t2
      v3 = t3
    return u1, u2, u3

  def inverse_mod(a, p):
    u1,v1,d = xgcd(a, p)
    return u1 if u1 > 0 else u1+p

  def crt(n,p):
    r = len(p)
    L = reduce(lambda x,y: x*y, p)
    li = [L / x for x in p]
    vi = [inverse_mod(li[i], p[i]) for i in range(r)]
    return reduce(lambda x,y: x+y, map(lambda x,y,z: x*y*z, n,li,vi)) % L

  def ec_add(P,Q,a,b,p):
    Px = P[0]
    Py = P[1]
    Qx = Q[0]
    Qy = Q[1]
    if P == [0,1]:
      return Q
    if Q == [0,1]:
      return P
    if P == Q:
      s = ((3*Px**2 + a)*inverse_mod(2*Py, p)) % p
      Rx = (s**2 - 2*Px)%p
      Ry = (-P[1] + s*(Px-Rx)) % p
      return [Rx,Ry]
    else:
      s = (Qy - Py)*inverse_mod(Qx - Px, p) % p
      Rx = (s**2 - Px - Qx)%p
      Ry = (s*(Px - Rx) - Py)%p
      return [Rx,Ry]

  def ec_mul(P,e,a,b,p):
    Q = [0,1]
    for i in range(int(math.ceil(math.log(e,2)))):
      if e%2:
        Q = ec_add(Q,P,a,b,p)
      P = ec_add(P,P,a,b,p)
      e >>= 1
    return Q

  def warmup(name):
    if len(name) > 10:
      name = name[0:10]
    else:
      while len(name) < 10:
        name = name + 'S'
  rotors = [rotor0,rotor1,rotor2,rotor3,rotor4,
            rotor5,rotor6,rotor7,rotor8,rotor9]

  rot = range(10)
  for i in range(10):
    rot[i] = rotors[i].find(name[i])

  serial = crt(rot, [len(rotors[i]) for i in range(10)])
  return serial

def crypto1(name):
  d = 65537
  n = 1821668788150059822966422307930288186476927434330004078983325993614616857
  e = 174725878413998504300527427786354881690797517767573630493566745566969857
  m = 0;
  for char in name:
    m = m*256 + ord(char);

  if m > n:
    m = m % n;

  return pow(m, e, n)

def crypto2(name):
  name_num = 0
  for char in name:
      name_num = name_num*256 + ord(char)

  name_num &= 2**64-1
  state = name_num
  for i in range(65537):
      state = (state >> 1) | ( (state&1) ^ (state >> 1)&1 ^ (state >> 3)&1 ^
               (state >> 4)&1 ^ 1) << 63

  return state

def crypto3(name):
  order = 18446693477654742300
  dlog  = 14494024699711394757
  name_num = 0
  for char in name:
    name_num = name_num*256 + ord(char);
  return (dlog * name_num)%order

def crypto4(name):
  p =  6277101735386680763835789423207666416102355444459739541047
  a =  0
  b =  3
  o =  6277101735386680763835789423061264271957123915200845512077
  k = inverse_mod(13, o)

  name_x = 0;
  for char in sys.argv[1]:
    name_x = name_x*256 + ord(char)
  while 1:
    quadres = (name_x**3 + 3)%p
    name_y = pow(quadres, (p+1)/4, p)
    if pow(name_y, 2, p) == quadres:
      break;
    name_x += 1
  P1 = [name_x, name_y]
  P2 = ec_mul(P1,k,a,b,p)
  return "%d-%d" % (P2[0],P2[1])

def crypto5(name):
  a = 0
  b = 13
  p = 9524793152874449521
  x = 1223334444333221111
  o = 9524793149788155121
  g1 = [1,4577206343548535956]

  name_c = 0;
  for char in sys.argv[1]:
    name_c = name_c*256 + ord(char)
  S = ec_mul(g1, (x*name_c)%o, a,b,p)
  return "%d-%d" % (S[0],S[1])

if(len(sys.argv) != 2):
        print "Usage: ", sys.argv[0], " <name>"
        quit()

print "Warmup  : ", warmup(sys.argv[1])
print "Crypto 1: ", crypto1(sys.argv[1])
print "Crypto 2: ", crypto2(sys.argv[1])
print "Crypto 3: ", crypto3(sys.argv[1])
print "Crypto 4: ", crypto4(sys.argv[1])
print "Crypto 5: ", crypto5(sys.argv[1])
```