CS 5000: F24: Theory of Computability

Digital Cash Systems

Part 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

Primitive Roots Module p

Let p be a prime. A **primitive root** mod p (aka a primitive root of/for p) is a number whose powers yield every nonzero class modulo p.

Example: Let $p = 7$. There are 6 nonzero equivalence classes mod 7: $[1]_7, [2]_7, [3]_7, [4]_7, [5]_7, [6]_7$. 3 is a primitive root mod 7, because

1. $3^1 \equiv 3 \pmod 7$;
2. $3^2 \equiv 2 \pmod 7$;
3. $3^3 \equiv 6 \pmod 7$;
4. $3^4 \equiv 4 \pmod 7$;
5. $3^5 \equiv 5 \pmod 7$;
6. $3^6 \equiv 1 \pmod 7$.

There are exactly $\phi(p-1)$ primitive roots of p, where $\phi(n)$ is Euler's totient.

A Useful Theorem on Primitive Roots Module p

The following useful theorem (let's call it the UT) summarizes some facts we'll need later in our lecture series on digital cash.

Useful Theorem (UT): Let g be a primitive root of the prime p. Then

1. Let n be an integer. Then $g^n \equiv 1 \pmod{p}$ if and only if $n \equiv 0 \pmod{p-1}$;
2. Let j, k be integers. Then $g^j \equiv g^k \pmod{p}$ if and only if $j \equiv k \pmod{p-1}$.

Examples of the Useful Theorem

Example 1: 3 is a primitive root of 7. Then $p - 1 = 6$. Take $n = 6$ and observe that $3^6 \equiv 1 \pmod 7$ and $6 \equiv 0 \pmod 6$. Take $n = 12$. Then we have $3^{12} \equiv 1 \pmod 7$ and $12 \equiv 0 \pmod 6$.

Example 2: 3 is a primitive root of 7. Observe that $3^5 \equiv 5 \pmod 7$ and $3^{11} \equiv 5 \pmod 7$. We also have $5 \equiv 11 \pmod 6$.

One-Way Functions

f(x) = y is called a **one-way** function if computing y given x is easy but computing x given y is probably hard.

The phrase **probably hard** means that from the point of view of probability theory guessing a response is very unlikely.

Example 1: Factorization: Given two primes p and q computing $pq = n$ is easy. However, computing p and q from n is probably hard.

Example 2: Discrete Logs: Computing $\alpha^x \pmod{p}$ is easy. However, solving for x given $\beta \equiv \alpha^x \pmod{p}$ is probably hard. E.g. solve for $9 \equiv 2^x \pmod{11}$. Then $x = 6$, because $2^6 \equiv 9 \pmod{11}$. Of course, we have $2^6 \equiv 2^{16} \equiv 2^{26} \pmod{11}$, so $x = 16$ or $x = 26$ also work.

One-Way Functions

It's easy to solve the discrete log problems when p is small. The larger p gets, the harder it is to solve the discrete log problem.

Pubic-key systems (e.g., RSA) depend on the probable hardness of factorization. Many digital cash systems depend on the probably hardness of discrete logs.

Cryptographic Hash Functions

A **cryptographic hash function** h maps a message of arbitrary length an produces a **message digest** of fixed length (e.g., 10 bits, 20 bits, 256 bits, etc.)

Given m, $h(m) = d$ is computed quickly. But given d computing m such that $h(m) = d$ is probably hard. This is called **preimage resistance**.

Given m, it is probably hard to find m' such that $m \neq m'$ but $h(m) = h(m')$. This is called **weak collision freedom** or **second preimage resistance**.

Outline

Participants

1. The Regulatory Authority;
2. The Bank;
3. The Spender;
4. The Merchant.

Properties of Digital Cash Systems

1. The cash can be sent securely through digital communication channels.
2. The cash cannot be duplicated and reused.
3. The spender can remain anonymous, i.e., neither the merchant nor the bank can identify the spender (provided that the coin is spent legitimately).
4. All transactions can be done w/o any communication with any central authority (e.g., the Federval Reserve).
5. It is possible to transfer the cash to others.
6. A piece of cash is divisible into smaller units.

We will not tackle Properties 5 and 6. Property 5 can be handled with RSA (or some other encryption/decryption system) and Property 6 depends on the ability of the sytem to handle extremely small floats.

Primitive Roots of a Prime p

```
def is_primitive_root_of_p(r, p):
    """
    Is number r a primitive root of a prime p?
    """
    assert is_prime(p)
    equiv_classes = set(r**i % p for i in range(1, p))
    for i in range(1, p):
        if not i in equiv_classes:
            return False
    return True

def find_primitive_roots_of_p(p):
    """
    Find all primitive roots of a prime p.
    """
    return set(r for r in range(1, p) if is_primitive_root_of_p(r, p))

>>> is_primitive_root_of_p(13, 227)
True
>>> len(find_primitive_roots_of_p(227)) == euler_phi(226)
True
>>> find_primitive_roots_of_p(7)
{3, 5}
>>> len(find_primitive_roots_of_p(7)) == euler_phi(6)
True
```

Initialization

This initialization is done by some recognized regulatory authority (e.g., a city council).

1. The Authority chooses a "large" primes p such that $q = (p-1)/2$ is also a prime. Let's choose $p = 227$ and $q = (227-1)/2 = 113$ to keep things smaller and simple.

2. The Authority finds a primitive root r of p and computes $g = r^2$. There are 112 primitive roots of 227. 13 is one of them.

Let's assume the Authority chooses 13, then $g = 13^2 = 169$. Note that $g^{k_1} \equiv g^{k_2} \pmod{p}$ if and only if $k_1 \equiv k_2 \pmod{q}$.

Example: Take $k_1 = 2$ and $k_2 = 115$. Then $169^2 \equiv 186 \pmod{227}$ and $169^{115} \equiv 186 \pmod{227}$. So, we have $169^2 \equiv 169^{115} \pmod{227}$ and $2 \equiv 115 \pmod{113}$.

3. The Authority chooses two random exponents (e.g., $k_1 = 3$ and $k_2 = 5$) and computes $g_1 = g^{k_1} \pmod{p}$ and $g_2 = g^{k_2} \pmod{p}$. So, if $g = 169$, then $g_1 = 169^3 \pmod{227}$ and $g_2 = 169^5 \pmod{227}$, which gives us $g_1 = 108$ and $g_2 = 112$.

Initialization: Steps 1 – 3

```
## 1. two primes: p and q=(p-1)/2
p, q = 227, 113

## 2. 13 is a primitive root of p=227.
g = 13**2

## 3. The authority chooses 2 random exponents
## k1, k2. These are thrown away after g1, g2
## are created.
k1, k2 = 3, 5

g1 = g**k1 % p
g2 = g**k2 % p
assert g1 == 108 and g2 == 112
```

Initialization

4. The Authority creates two hash functions: H and H_0. H takes 5-tuples of integers and maps them to an integer mod q. H_0 takes 4-tuples of integers and maps them to an integer mod q.

5. The authority makes g, g_1, g_2, H, and H_0 public. The random exponents k_1 and k_2 are destroyed. They served their purpose. If a hacker discovers them, the system may be compromised.

The initialization is now done. From then on, all legal issues (e.g., fraud) will be resolved by referencing these numbers and functions and talking to the Regulatory Authority.

Initialization: Steps 4 – 5

```
## 4. The authority chooses 2 hash functions H and H0.
import hashlib

def H(int_5_tup):
    assert len(int_5_tup) == 5
    for i in range(5):
        assert isinstance(int_5_tup[i], int)
    return int(hashlib.sha256(bytes(int_5_tup)).hexdigest(), 16) % q

def H0(int_4_tup):
    assert len(int_4_tup) == 4
    for i in range(4):
        assert isinstance(int_4_tup[i], int)
    return int(hashlib.sha224(bytes(int_4_tup)).hexdigest(), 16) % q

## 5. The authority makes g, g1, g2, H, H0 public.

>>> H((1, 2, 3, 4, 5))
86
>>> hashlib.sha256(bytes((1, 2, 3, 4, 5))).hexdigest()
'74f81fe167d99b4cb41d6d0ccda82278caee9f3e2f25d5e5a3936ff3dcec60d0'
>>> int(hashlib.sha256(bytes((1, 2, 3, 4, 5))).hexdigest(), 16)
52906688538785375202382687936624367891607472723899426723844832637138437431504
>>> int(hashlib.sha256(bytes((1, 2, 3, 4, 5))).hexdigest(), 16) % 113
86
>>> H0((1, 2, 3, 4))
23
>>> hashlib.sha224(bytes((1, 2, 3, 4))).hexdigest()
'fe19672fa20acdc6a794f7ed68dd7563c27f647920164b991f1ad037'
>>> int(hashlib.sha224(bytes((1,2, 3, 4))).hexdigest(), 16)
26759772300912386439784424296664021303067358984775725034829571084343
>>> int(hashlib.sha224(bytes((1,2, 3, 4))).hexdigest(), 16) % 113
23
```

The Bank

1. The Bank (e.g., a community bank or a credit union) chooses its secret identity x. The term **secret** means that the participant does not reveal this number to anybody.

2. The Bank computes:
   - $h \equiv g^x \pmod{p}$;
   - $h_1 \equiv g_1^x \pmod{p}$;
   - $h_2 \equiv g_2^x \pmod{p}$.

3. The Bank makes h, h_1, and h_2 public.

The Bank: Steps 1 – 3

```
## 1. The Bank chooses its secrete ID x.
x = 19

## 2. The Bank computes:
h  = g**x  % p
h1 = g1**x % p
h2 = g2**x % p

assert h == 30
assert h1 == 214
assert h2 == 104

## 3. The Bank makes h, h1, h2 public.
```

The Spender and the Merchant

1. The Spender chooses a secret identity u and computes the account number $I \equiv g_1^u \pmod{p}$.

2. The Spender sends I to the bank.

3. The Bank associates I with the Sender (how that association is done is application dependent) and sends $z' \equiv (Ig_2)^x \pmod{p}$ to the Spender.

4. The Merchant chooses an identity number M and registers it with the Bank.

The Spender and the Merchant

```
## 1. The Spender chooses a secrete identity
u = 23

## and generates the account number
I = g1**u % p
assert I == 121

## 2. The Spender sends I is sent to the Bank.

## 3. The Bank sends associates I with the Spender and
##    computes z' and sends to the Spender.
z_prime = (I*g2)**x % p
assert z_prime == 57

## 4. The Merchant creates an ID M and registers it
##    with the Bank.
M = 29
```

Coin Creation

The coins are created by the Bank and the Spender. Each coin is a 6-tuple of positive natural numbers (A, B, z, a, b, r).

The Bank chooses a random number w (w is different for each issued coin).

1. The Bank then computes:
   - $g_w = g^w \pmod{p}$;
   - $\beta = (Ig_2)^w \pmod{p}$

.

Coin Creation

2. The Spender chooses a secret random 5-tuple of integers $(s, x_1, x_2, \alpha_1, \alpha_2)$, $s \neq 0$ (a new random 5-tuple for each coin).

3. The Spender computes:

   - $A = (Ig_2)^s \pmod{p}$;
   - $B = g_1^{x_1} g_2^{x_2} \pmod{p}$;
   - $z = (z')^s \pmod{p}$;
   - $a = g_w^{\alpha_1} g^{\alpha_2} \pmod{p}$;
   - $b = \beta^{s\alpha_1} A^{\alpha_2} \pmod{p}$, where $A \neq 1$.

The last condition simply means coins $(1, B, z, a, b)$ are not allowed.

Coin Creation: Steps 1 – 3

```
## The Bank creates a random number w (different
## for each coin).
w = 53

## 1. The Bank computes gw and beta and sends gw and beta
##    to the Spender.
gw   = g**w % p
beta = (I*g2)**w % p
assert gw == 27
assert beta == 213

## 2. The Spender generates a secret random 5-tuple. A new 5-tuple for every coin.
## (s, x1, x2, alpha1, alpha2) = [random.randint(1, 100) for i in range(5)]
s, x1, x2, alpha1, alpha2 = 81, 28, 3, 2, 7
sender_5_tup = (s, x1, x2, alpha1, alpha2)
assert s != 0

## 3. The Spender computes
A = (I*g2)**s % p
assert A != 1
B = ((g1**x1)*(g2**x2)) % p
z = (z_prime)**s % p
a = ((gw**alpha1)*(g**alpha2)) % p
b = ((beta**(s*alpha1))*(A**alpha2)) % p
```

Coin Creation

4. The Spender computes $c \equiv \alpha_1^{-1} H(A, B, z, a, b) \pmod{q}$ and sends c to the Bank.

5. The Bank computes $c_1 = cx + w \pmod{q}$, sends c_1 to the Spender, and immediately deducts the amount equal to 1 coin from the Spender's account.

6. The Spender computes $r = \alpha_1 c_1 + \alpha_2 \pmod{q}$.

7. The Spender now has the coin (A, B, z, a, b, r).

Coin Creation: Steps 4, 5

```
## my implemented mult_inv in ntutils
## 4. the Spender computes
from ntutils import mult_inv
mult_inv_alpha1 = mult_inv(alpha1, q)
assert alpha1 * mult_inv_alpha1 % q == 1

## and sends c to the Bank. c stands for the
## coin requested.
c = mult_inv_alpha1 * H((A,B,z,a,b)) % q

## 5. The Bank computes c1 (one created coin)
##    and c1 to the Spender and deducts one
##    coin unit from the Sender's account.
c1 = (c*x + w) % q
```

Coin Creation: Steps 6, 7

```
## 6. The Spender computes r, the last piece
##    of the created coin.
r = (alpha1 * c1 + alpha2) % q

## 7. The Spender now has the newly issued coin:
issued_coin = (A,B,z,a,b,r)
```

Spending the Coin

1. The Merchant checks the validity of the coin
   - $g^r \equiv ah^{H(A,B,z,a,b)} \pmod{p}$;
   - $A^r \equiv z^{H(A,B,z,a,b)} b \pmod{p}$.

2. The Merchant now computes $d = H_0(A, B, M, t)$, where t represents the time of the transaction different for each coin so that d will be different for each transaction. The Merchant sends d to the Spender.

3. The Spender computes r_1 and r_2 and sends them to the Merchant:
   - $r_1 \equiv dus + x_1 \pmod{q}$;
   - $r_2 \equiv ds + x_2 \pmod{q}$.

Recall that u is the Spender's identity and s, x_1, x_2 are part of the secret random 5-tuple that the Spender generated before.

Spending the Coin

4. The Merchant accepts the coin if and only if the following congruence holds

- $g_1^{r_1} g_2^{r_2} \equiv A^d B \pmod{p}$;

If the Merchant rejects the coin, the Merchant may appeal to the authority.

The Spender has now spent the coin.

Spending the Coin: Steps 1 – 4

```
## 1. The Merchant checks the validity of the coin
assert g**r % p == a*(h**H((A,B,z,a,b))) % p
assert A**r % p == (z**(H((A,B,z,a,b))))*b % p

## 2. The Merchant now computes d, where
##    t is the date and time of the transaction
##    arbitrarily set to 1 here for simplicity
##    The Merchant sends d to the Spender.
t = 1
d = H0((A,B,M,t))
assert d == 30

## 3. The Spender computes r1 and r2 and sends them
##    to the Merchant.
r1 = (d*u*s + x1) % q
r2 = (d*s + x2) % q
assert r1 == 96 and r2 == 60

## 4. The Merchant checks if the assertion below holds
##    and only then accepts the coin.
assert (g1**r1)*(g2**r2) % p == (A**d)*B % p
```

Depositting the Coin

1. The Merchant submits the coin (A, B, z, a, b, r) and the triple (r_1, r_2, d) to the Bank. The Bank checks if the coin has been previously deposited. If yes, then the Bank initiates fraud control. The coin is $(112, 34, 104, 1, 1, 6)$ has not been deposited before. So, no fraud is detected. Then the Bank checks the validity of the coin itself by computing:

- $g^r \equiv a h^{H(A,B,z,a,b)} \pmod{p}$;
- $A^r \equiv z^{H(A,B,z,a,b)} b \pmod{p}$;
- $g_1^{r_1} g_2^{r_2} \equiv A^d B \pmod{p}$.

If the coin is valid (and it is), the Bank credits the Merchant's account.

The transaction is now complete. The coin $(112, 34, 104, 1, 1, 6)$ is now in the bank's log.

Depositting the Coin

```
## 1. The Merchant sends the coin (A,B,z,a,b,r) and
## the triple (r1, r2, d) to the Bank.
## The Bank checks if (A,B,z,a,b,r) has not been
## previously deposited. If it has not been, the Bank
## takes it. If it has been, the Bank initiates fraud
## control.

## 2. The Bank checks the validity of the coin
assert g**r % p == a*(h**H((A,B,z,a,b))) % p
assert A**r % p == (z**H((A,B,z,a,b)))*b % p
assert (g1**r1)*(g2**r2) % p == (A**d)*B % p

## if all assertions pass, the Bank accepts the coin.
```

Spending the Coin Twice

The Spender uses the same coin (A, B, z, a, b, r) with another merchant Vendor. We have the the situation when the Merchant submitted this coin along with (r_1, r_2, d) to the Bank.

1. The Vendor computes $d' = H_0(A, B, M, t')$. Recall that t' represents the time of the transaction different for each coin and sends d' to the Spender.

2. The Spender computes r_1' and r_2' and sends them to the Vendor:
   - $r_1' \equiv d'us + x_1 \pmod{q}$;
   - $r_2' \equiv d's + x_2 \pmod{q}$.

3. The Vendor accepts the coin, because the congruence below holds.
   - $g_1^{r_1'} g_2^{r_2'} \equiv A^{d'} B \pmod{p}$;

Spending the Coin Twice: Steps 1–3

```
## 1. The Vendor computes d_prime, where
##    t is the date and time of the transaction
##    arbitrarily set to 1 here for simplicity.
##    The Vendor sends d_prime to the Spender.
t_prime = 2
d_prime = H0((A,B,M,t_prime))
assert d_prime == 24

## 2. The Spender computes r1_prime and r2_prime and
##    sends them to the Vendor.
r1_prime = (d_prime*u*s + x1) % q
r2_prime = (d_prime*s + x2) % q
assert r1_prime == 105 and r2_prime == 26

## 3. The Vendor checks if the assertion below holds
##    and only then accepts the coin.
assert (g1**r1_prime)*(g2**r2_prime) % p == (A**d_prime)*B % p
```

Spending the Coin Twice: Steps 4,5

4. The Vendor sends the coin (A, B, z, a, b, r) and the triple (r_1', r_2', d') to the Bank. The Bank discovers (A, B, z, a, b, r) has been deposited before. The Bank knows that the following congruences hold.

- $r_1 - r_1' \equiv u * s * (d - d') \pmod{q}$;
- $r_2 - r_2' \equiv s * (d - d') \pmod{q}$.

5. Then Bank now computes the Spender's ID by computing:
$u \equiv (r_1 - r_1')(r_2 - r_2')^{-1} \pmod{q}$.

Spending the Coin Twice: Steps 4,5

```
## 4. The Vendor sends the coin (A,B,z,a,b,r) and
## the triple (r1_prime, r2_prime, d_prime) to the Bank.
## The Bank checks if (A,B,z,a,b,r) and discovers
## that the coin has been spent before.
## The Bank knows that the following assertions will hold.
assert (r1 - r1_prime) % q == u*s*(d - d_prime) % q
assert (r2 - r2_prime) % q == s*(d - d_prime) % q

## 5. Then Bank now computes the Spender ID.
r2_diff = r2 - r2_prime
mult_inv_r2_diff = mult_inv(r2_diff, q)
Spender_ID = ((r1 - r1_prime)*mult_inv_r2_diff) % q

## The Spender's ID has been discovered.
assert Spender_ID == u
```

Outline

Anonymity

The Merchant and the Spender do not provide any identification. This is identical with conventional cash.

The Bank never sees the actual coin (A, B, z, a, b, r) until it is deposited by the Merchant. The Bank provides w and c_1 and sees only c. The coin contains information to identify the Spender but only in case of double spending.

The Bank could try to keep a list of all values c it has received along with the values of H for every deposited coin. Then try all combinations to find α_1.

The previous step is provably hard in a system with trillions of coins, the possible values of α_1 it too large. The numbers α_1, α_2 provide what is called a **restricted blind signature** for the coin.

The system is **probably** anonymous.

Other Cases of Fraud

The system must address other cases of fraud. All are **probably** difficult.

1. The Merchant attempts to submit the coin twice.
2. Another player makes an unauthorized coin.
3. Another merchant receives the coin from the Spender, deposits it with the Bank and attempts to spend the same coin with the Merchant.
4. A bank worker forges a coin from the inside.
5. The coin is stolen from the Spender.
6. The coin is stolen from the Merchant.

Two Main Critiques

The amount of coins is unlimited, which makes it hard to assign the real value to each issued coin.

Infinite coin divisibility is not allowed.

References

1. T. Okamoto, K. Ohta. "Universal Electronic Cash," *Advances in Cryptography*, Lecture Notes in Computer Science 576, Springer-Verlag, 1992, pp. 324-337.
2. S. Brands. "Untraceable off-line cash in wallets with observers," *Advances in Cryptography*, Lecture Notes in Computer Science 773, Springer-Verlag, 1994, pp. 302-318.
3. I. Niven and H. S. Zuckerman. "An Introduction to the Theory of Numbers," Second Edition. John Wiley and Sons.
4. T. Cormen, C. Leiserson, R. Rivest. "Introduction to Algorithms," Ch. 33. MIT Press.