Paillier cryptosystem

The scheme works as follows:

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Paillier_cryptosystem&action=edit&section=2)**] Key generation**

1. Choose two large [prime numbers](http://en.wikipedia.org/wiki/Prime_number) *p* and *q* randomly and independently of each other such that gcd(*pq*,(*p* − 1)(*q* − 1)) = 1. This property is assured if both primes are of equivalent length, i.e., p, q \in 1 || \{0,1\}^{s-1}for security parameter *s*.[[1]](http://en.wikipedia.org/wiki/Paillier_cryptosystem#cite_note-katzLindell-0)

Compute *n* = *pq* and \lambda=\operatorname{lcm}(p-1,q-1). Where lcm can be computed as follows: \operatorname{lcm}(a,b)=\frac{|a\cdot b|}{\operatorname{gcd}(a,b)}.

1. Select random integer *g* where g\in \mathbb Z^{*}_{n^{2}}
2. Ensure *n* divides the order of *g* by checking the existence of the following [modular multiplicative inverse](http://en.wikipedia.org/wiki/Modular_multiplicative_inverse): \mu = (L(g^{\lambda} \mod n^{2}))^{-1} \mod n,

where function *L* is defined as L(u) = \frac{u-1}{n}.

Note that the notation \frac{a}{b}does not denote the modular multiplication of *a* times the [modular multiplicative inverse](http://en.wikipedia.org/wiki/Modular_multiplicative_inverse) of *b* but rather the [quotient](http://en.wikipedia.org/wiki/Quotient) of *a* divided by *b*, i.e., the largest integer value v \ge 0to satisfy the relation a \ge vb.

* The public (encryption) key is (*n*,*g*).
* The private (decryption) key is (λ,μ).

If using p,q of equivalent length, a simpler variant of the above key generation steps would be to set g = n+1, \lambda = \varphi(n),and \mu = \varphi(n)^{-1} \mod n, where \varphi(n) = (p-1)(q-1).[[1]](http://en.wikipedia.org/wiki/Paillier_cryptosystem#cite_note-katzLindell-0)

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Paillier_cryptosystem&action=edit&section=3)**] Encryption**

1. Let *m* be a message to be encrypted where m\in \mathbb Z_{n}
2. Select random *r* where r\in \mathbb Z^{*}_{n} 
3. Compute ciphertext as:  c=g^m \cdot r^n \mod n^2 

**[**[**edit**](http://en.wikipedia.org/w/index.php?title=Paillier_cryptosystem&action=edit&section=4)**] Decryption**

1. Ciphertext c\in \mathbb Z^{*}_{n^{2}} 
2. Compute message: m = L(c^{\lambda} \mod n^{2}) \cdot \mu \mod n

Cramer–Shoup cryptosystem

Cramer–Shoup consists of three algorithms: the key generator, the encryption algorithm, and the decryption algorithm.

The key generator works as follows:

* [Alice](http://en.wikipedia.org/wiki/Alice_and_Bob) generates an efficient description of a [cyclic group](http://en.wikipedia.org/wiki/Cyclic_group) *G* of order *q* with two distinct, random [generators](http://en.wikipedia.org/wiki/Generating_set_of_a_group) *g*1,*g*2.
* Alice chooses five random values (*x*1,*x*2,*y*1,*y*2,*z*) from \{0, \ldots, q-1\}.
* Alice computes c = {g}_{1}^{x_1} g_{2}^{x_2}, d = {g}_{1}^{y_1} g_{2}^{y_2}, h = {g}_{1}^{z}.
* Alice publishes (*c*,*d*,*h*), along with the description of *G*,*q*,*g*1,*g*2, as her [public key](http://en.wikipedia.org/wiki/Public_key). Alice retains (*x*1,*x*2,*y*1,*y*2,*z*) as her [secret key](http://en.wikipedia.org/wiki/Secret_key). The group can be shared between users of the system.

The encryption algorithm works as follows: to encrypt a message *m* to Alice under her public key (*G*,*q*,*g*1,*g*2,*c*,*d*,*h*),

* Bob converts *m* into an element of *G*.
* Bob chooses a random *k* from \{0, \ldots, q-1\}, then calculates:
  + u_1 = {g}_{1}^{k}, u_2 = {g}_{2}^{k}
  + e = h^k m \,
  + \alpha = H(u_1, u_2, e) \,, where H() is a collision-resistant [cryptographic hash function](http://en.wikipedia.org/wiki/Cryptographic_hash_function).
  + v = c^k d^{k\alpha} \,
* Bob sends the ciphertext (*u*1,*u*2,*e*,*v*) to Alice.

The decryption algorithm works as follows: to decrypt a ciphertext (*u*1,*u*2,*e*,*v*) with Alice's secret key (*x*1,*x*2,*y*1,*y*2,*z*),

* Alice computes \alpha = H(u_1, u_2, e) \,and verifies that {u}_{1}^{x_1} u_{2}^{x_2} ({u}_{1}^{y_1} u_{2}^{y_2})^{\alpha} = v \,. If this test fails, further decryption is aborted and the output is rejected.
* Otherwise, Alice computes the plaintext as m = e / ({u}_{1}^{z}) \,.

The decryption stage correctly decrypts any properly-formed ciphertext, since

 {u}_{1}^{z} = {g}_{1}^{k z} = h^k \,, and m = e / h^k. \,

## Rabin cryptosystem

## Key generation

* Choose two large distinct primes *p* and *q*. One may choose p \equiv q \equiv 3 \pmod{4}to simplify the computation of square roots modulo *p* and *q* (see below). But the scheme works with any primes.
* Let n = p \cdot q. Then *n* is the public key. The primes *p* and *q* are the private key.

To encrypt a message only the public key *n* is needed. To decrypt a ciphertext the factors *p* and *q* of *n* are necessary.

## [[edit](http://en.wikipedia.org/w/index.php?title=Rabin_cryptosystem&action=edit&section=3)] Encryption

Let *P* = {0,...,*n* − 1} be the plaintext space (consisting of numbers) and m \in Pbe the [plaintext](http://en.wikipedia.org/wiki/Plaintext). Now the [ciphertext](http://en.wikipedia.org/wiki/Ciphertext) *c* is determined by

c = m^2 \, \bmod \, n.

## [[edit](http://en.wikipedia.org/w/index.php?title=Rabin_cryptosystem&action=edit&section=4)] Decryption

To decode the ciphertext, the private keys are necessary. The process follows:

If *c* and *r* are known, the plaintext is then m \in \{ 0,  ..., n-1 \}with m^2 \equiv c\pmod{r}. For a [composite](http://en.wikipedia.org/wiki/Composite_number) *r* (that is, like the Rabin algorithm's n = p \cdot q) there is no efficient method known for the finding of *m*. If, however r \in \mathbb{P}(as are *p* and *q* in the Rabin algorithm), the [Chinese remainder theorem](http://en.wikipedia.org/wiki/Chinese_remainder_theorem) can be applied to solve for *m*.

Thus the [square roots](http://en.wikipedia.org/wiki/Square_root)

m_p = \sqrt{c} \, \bmod \, p

and

m_q = \sqrt{c} \, \bmod \, q

must be calculated (see section below).

In our example we get *mp* = 1 and *mq* = 9.

By applying the [extended Euclidean algorithm](http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm), *yp* and *yq*, with y_p \cdot p + y_q \cdot q = 1are calculated. In our example, we have *yp* = − 3 and *yq* = 2.

Now, by invocation of the Chinese remainder theorem, the four square roots + *r*, − *r*, + *s* and − *s* of c + n \mathbb{Z} \in \mathbb{Z} / n \mathbb{Z}are calculated (\mathbb{Z} / n \mathbb{Z} here stands for the [ring of congruence classes](http://en.wikipedia.org/wiki/Modular_arithmetic#The_ring_of_congruence_classes) modulo *n*). The four square roots are in the set {0,...,*n* − 1}):

\begin{matrix}
r  & = & ( y_p \cdot p \cdot m_q + y_q \cdot q \cdot m_p) \, \bmod \, n  \\
-r & = & n - r  \\
s  & = & ( y_p \cdot p \cdot m_q - y_q \cdot q \cdot m_p) \, \bmod \, n  \\
-s & = & n - s 
\end{matrix}

One of these square roots \mod \, nis the original plaintext *m*. In our example, m \in \{ 64, \mathbf{20}, 13, 57 \}.

## [[edit](http://en.wikipedia.org/w/index.php?title=Rabin_cryptosystem&action=edit&section=5)] Computing square roots

The decryption requires to compute square roots of the ciphertext *c* modulo the primes *p* and *q*. Choosing p \equiv q \equiv 3\pmod{4}allows to compute square roots by

m_p = c^{\frac{(p+1)}{4}} \, \bmod \, p

and

m_q = c^{\frac{(q+1)}{4}} \, \bmod \, q.

We can show that this method works for *p* as follows. First p \equiv 3\!\!\!\pmod{4}implies that (*p*+1)/4 is an integer. The assumption is trivial for *c*≡0 (mod *p*). Thus we may assume that *p* does not divide *c*. Then

m_p^2 \equiv c^{\frac{(p+1)}{2}} \equiv c\cdot c^{\frac{(p-1)}{2}} \equiv c \cdot\left({c\over p}\right) \pmod{p},

where \left({c\over p}\right)is a [Legendre symbol](http://en.wikipedia.org/wiki/Legendre_symbol). From c\equiv m^2\pmod{pq}follows that c\equiv m^2\pmod{p}. Thus *c* is a [quadratic residue](http://en.wikipedia.org/wiki/Quadratic_residue) modulo *p*. Hence \left({c\over p}\right)=1and therefore

m_p^2 \equiv c \pmod{p}.

The relation p \equiv 3\pmod{4}is not a requirement because square roots modulo other primes can be computed too. E.g. Rabin proposes to find the square roots modulo primes by using a special case of [Berlekamp's algorithm](http://en.wikipedia.org/wiki/Berlekamp%27s_algorithm).

# Sigma protocols

## Sigma protocol of DLOG

**PROTOCOL 6.1.1 (Schnorr’s Protocol for Discrete Log)**

*•* **Common input:** The prover *P* and verifier *V* both have (*p, q, g, h*)

*•* **Private input:** *P* has a value *w ∈* Z*q* such that *h* = *gw* mod *p*

* **Default behavior on wrong input:**
* **Prover’s Output:**nothing
* **Verifier’s output:** accept or reject

*•* **The protocol:**

1. V Checks that:
   1. p, q are prime
   2. g, h have order q, if not aborts with error
2. The prover *P* chooses a random *r ←R* Z*q* and sends *a* = *gr* mod *p* to *V* .
3. *V* chooses a random *challenge e ←R {*0*,* 1*}t* and sends it to *P*, where *t* is

fixed and it holds that 2*t < q*. If V chooses a longer challenge abort with error

1. *P* sends *z* = *r* + *ew* mod *q* to *V* ,
2. *V* does the following *:*
   1. checks that *gz* = *ahe* mod *p*,
   2. accepts if and only if all the above statement are true.

# Sigma protocol of DH tuple

**PROTOCOL 6.2.1 (Protocol Template for Relation** *R***)**

*•* **Common input:** The prover *P* and verifier *V* both have *x*

*•* **Private input:** *P* has a value *w* such that (*x,w*) *∈ R*

* **Default behavior on wrong input:**
* **Prover’s Output:**nothing
* **Verifier’s output:** accept or reject
* **The protocol template:**

1. *P* sends *V* a message *a*.
2. *V* sends *P* a *random t*-bit string *e*.
3. *P* sends a reply *z*, and *V* decides to accept or reject based solely on the data it has seen; i.e., based only on the values (*x, a, e, z*).

**PROTOCOL 6.2.4 (***Σ* **Protocol for Diffie-Hellman Tuples)**

* **Common input:** The prover *P* and verifier *V* both have (G*, q, g, h, u, v*). G denotes a concise representation of a finite group of prime order q, and g and h are generators of G.
* **Private input:** *P* has a value *w* such that *u* = *gw* and *v* = *hw*.
* **Default behavior on wrong input:**
* **Prover’s Output:**nothing
* **Verifier’s output:** accept or reject
* **The protocol:**

1. V checks that:
   1. G is a group of order *q*
   2. *g* and *h are generators of G.*
   3. *u, v ∈* G
2. The prover *P* chooses a random *r ←R* Z*q* and computes *a* = *gr* and *b* = *hr*.

It then sends (*a, b*) to *V* .

1. *V* chooses a random challenge *e ←R {*0*,* 1*}t* where 2*t < q* and sends it to *P*.
2. *P* sends *z* = *r* + *ew* mod *q* to *V*
3. *V* does the following *:*
   1. Checks that *gz* = *aue* and *hz* = *bve*
   2. accepts if and only if all the above statement are true.

## AND of any number of Sigma protocols

**PROTOCOL 6.4.1 (AND Protocol for Relation** *R* **Based on** *π0 and π1***)**

* **Common input:** The prover *P* and verifier *V* both have a pair (*x*0*, x*1)
* **Private input:** *P* also has a pair (*w*0*, w*1) such that (*x*0*, w*0) *∈ R0 and*  (*x*1*, w*1) *∈ R1 (it might be that* R0=R1)
* **Default behavior on wrong input:**
* **Verifier’s Output:**Accept or reject
* **Prover’s output:** nothing
* **The protocol:**

1. *P* sends *V* a message (*a0, a1)*
2. *V* sends *P* a *random t*-bit string *e*.
3. *P* sends a reply *(z0,z1)*,
4. *V* decides to accept if the following 2 statements are correct.
   1. transcripts (*a0, e, z*0) is accepting in *π0*, on inputs *x*0
   2. transcript(*a1, e, z*1) is accepting in *π1*, on inputs *x*1.

## OR of any two Sigma protocols

**PROTOCOL 6.4.1 (OR Protocol for Relation** *R* **Based on** *π***)**

* **Common input:** The prover *P* and verifier *V* both have a pair (*x*0*, x*1)
* **Private input:** *P* has a value *w* and a bit *b* such that (*xb,w*) *∈ R*
* **Default behavior on wrong input:**
* **Verifier’s Output:**Accept or reject
* **Prover’s output:** nothing
* **The protocol:**

1. *P* computes the first message *ab* in *π*, using (*xb,w*) as input.
2. *P* chooses *e*1*−b* at random and runs the simulator *M* on input (*x*1*−b, e*1*−b*),
3. Let (*a*1*−b, e*1*−b, z*1*−b*) be the output of *M*.

The output of *M* is computed as follows:

* 1. Choose *z*1*−b* at random from group G. (e.g. in DL it is *z ←R* Zp*\**)
  2. Choose *e*1*−b ←R {*0*,* 1*}t*
  3. Calculate *a*1*−b* as a function of *(e*1*−b ,z*1*−b* ). (e.g. in DL it is

*a*1*−b* = *gzh− e*1*−b* mod *p*)

1. *P* sends (*a*0*, a*1) to *V*.
2. *V* chooses a random *t*-bit string *s* and sends it to *P*.
3. *P* sets *eb* = *s⊕e*1*−b* and computes the answer *zb* in *π* to challenge *eb* using

(*xb, ab, eb,w*) as input.

1. *P* sends (*e*0*, z*0*, e*1*, z*1) to *V*.
2. *V* checks that:
   1. *e*0 *⊕ e*1 = *s*
   2. transcripts (*a*0*, e*0*, z*0) is accepting in *π*, on inputs *x*0
   3. transcript(*a*1*, e*1*, z*1) is accepting in *π*, on inputs *x*1.
   4. If all the above statements are true V accepts. Otherwise reject.

# Zero-knowledge

These are ways to transform a sigma-protocol to a ZKPOK.

We do this by requiring the verifier to commit to *e* before it receives the first message from P. This way we can consider the Verifier to be honest.

**Conceptual question: why a not-honest verifier makes the sigma-protocol not to behave like a ZKPOK?**

**Another question: what is the difference between Zero-Knowledge Proof for** *LR* **Based on** *π* and **ZK Proof of Knowledge for** *R* **Based on** *π?*

## Zero-knowledge for every Sigma-protocol using any commitment

**PROTOCOL 6.5.1 (Zero-Knowledge Proof for** *LR* **Based on** *π***)**

* **Common input:** The prover *P* and verifier *V* both have *x*
* **Private input:** *P* has a value *w* such that (*x,w*) *∈ R.*
* **Default behavior on wrong input:**
* **Verifier’s Output:**Accept or reject
* **Prover’s output:** nothing
* **The protocol:**

1. *V* chooses a random *t*-bit string *e* and interacts with *P* via the commitment protocol **com** in order to commit to *e*.
2. *P* computes the first message *a* in *π*, using (*x,w*) as input, and sends it to *V* .
3. *V* decommits to *e* to *P*.
4. *P* verifies the decommitment and aborts if it is not valid. Otherwise, it

computes the answer *z* to challenge *e* according to the instructions in *π* .

1. P sends *z* to *V* .
2. *V* accepts if and only if transcript (*a, e, z*) is accepting in *π* on input *x*.

## ZKPOK for every Sigma-protocol using any trapdoor commitment

**PROTOCOL 6.5.4 (ZK Proof of Knowledge for** *R* **Based on** *π***)**

* **Common input:** The prover *P* and verifier *V* both have *x*
* **Private input:** *P* has a value *w* such that (*x,w*) *∈ R*
* **Default behavior on wrong input:**
* **Verifier’s Output:**Accept or reject
* **Prover’s output:** nothing
* **The protocol:**

1. *V* chooses a random *t*-bit challenge *e* and interacts with *P* via the trapdoor commitment protocol **com** in order to commit to *e*.
2. *P* computes the first message *a* in *π*, using (*x,w*) as input, and sends it to *V* .
3. *V* reveals *e* to *P* by decommitting.
4. *P* verifies the decommitment and aborts if it is not valid. Otherwise, it

computes the answer *z* to challenge *e* according to the instructions in *π*

1. P sends *z* and the trapdoor **trap** to *V*.
2. *V* accepts if and only if the trapdoor **trap** is valid (For ex: for DLOG sigma-protocol, given **trap** check that h = gtrap) and the transcript (*a, e, z*) is accepting in *π* on input *x*.

# Commitment schemes

## Trapdoor DLOG commitment scheme

**PROTOCOL 6.6.2 (Commitment from** *DL Σ***-Protocol)**

* **Input:** The committer *C* and receiver *R* both hold 1*n* and also (p, q, g), and the committer *C*

has a value *e ∈ {*0*,* 1*}t*.

* **Default behavior on wrong input:**
* **receiver’s Output:**Accept or rejectand **trap *w***
* **committer’s output:** nothing
* **The commit phase:**

1. The receiver *R* runs (in private) the generator *G* on input 1*n* to obtain

(gw,w) *∈ R*. How does n relate to (gw,w)?

1. R sends gwto *C*.
2. *C* verifies that gw *∈ LR*; that is,gw *∈ Zp\**
3. if not it aborts.
4. If yes, in order to commit to *e ∈*

*{*0*,* 1*}t*, the committer *C* runs the *DL (6.1.1) Σ*-protocol simulator *M* on input (gw*,e*) and obtains a transcript (*a, e, z*).

The output of *M* is computed as follows:

* 1. *z ←R* Zp*\**
  2. *a* = *gzh− e* mod *p*

1. *C* then sends *a* = *gzh− e* mod *p* to *R*.

* **The decommit phase:** In order to decommit, the committer *C* sends the

remainder of the transcript (*e, z*) to *R*, who accepts *e* as the decommitted value

if and only if (*a, e, z*) is an accepting transcript in *DL Σ***-Protocol**with respect to input gw.

## Trapdoor (equivocal) commitment schemes

**PROTOCOL 6.6.2 (Commitment from** *Σ***-Protocol)**

* **Input:** The committer *C* and receiver *R* both hold 1*n*, and the committer *C*

has a value *e ∈ {*0*,* 1*}t*.

* **Default behavior on wrong input:**
* **receiver’s Output:**Accept or rejectand **trap *w***
* **committer’s output:** nothing
* **The commit phase:**

1. The receiver *R* runs (in private) the generator *G* on input 1*n* to obtain

(*x,w*) *∈ R*, and sends *x* to *C*.

1. *C* verifies that *x ∈ LR*;
2. if not it aborts.
3. If yes, in order to commit to *e ∈*

*{*0*,* 1*}t*, the committer *C* runs the *Σ*-protocol simulator *M* on input (*x, e*)

and obtains a transcript (*a, e, z*).

The output of *M* is computed as follows:

* 1. Choose *z* at random from group G. (e.g in DL it is *z ←R* Zp*\**)
  2. Calculate *a* as a function of *( e ,z)*. (e.g in DL it is

*a* = *gzh− e* mod *p*)

1. *C* then sends *a* to *R*.

* **The decommit phase:** In order to decommit, the committer *C* sends the

remainder of the transcript (*e, z*) to *R*, who accepts *e* as the decommitted value

if and only if (*a, e, z*) is an accepting transcript in *π* with respect to input *x*.

## Pedersen commitments

**PROTOCOL 6.5.3 (The Pedersen Commitment Protocol)**

* **Input:** The committer *C* and receiver *R* both hold 1*n*, and the committer *C*

has a value *x ∈ {*0*,* 1*}n* interpreted as an integer between 0 and 2*n*.

* **Default behavior on wrong input:**
* **receiver’s Output:**Accept or rejectand **trap** *a*
* **committer’s output:** nothing
* **The commit phase:**

1. The receiver *R* chooses (G*, q, g*) where G is a group of order *q* with generator

*g* and *q >* 2*n*.

1. *R* then chooses a random *a ←* Z*q*, computes *α* = *ga*
2. R sends (G*, q, g, α*) to *C*.
3. The committer *C* verifies that
   1. G is a group of order *q*,
   2. *g* is a generator
   3. *α ∈* G. Then
   4. If not all the above statements are true. Abort with error. Otherwise continue
4. C chooses a random *r ←* Z*q*, computes *c* = *gr · αx*
5. C sends *c* to *R*.

* **The decommit phase:**

The committer *C* sends (*r, x*) to *R*, who verifies that *c* = *gr · αx*.

# Oblivious Transfers

# Naor-Pinkas (using any DH group)

**PROTOCOL 7.2.1 (Private Oblivious Transfer *π*P**

**OT)**

*•* **Inputs:** The sender has a pair of strings *x*0*, x*1 *of the same (arbitrary) length* and the receiver has a bit

*σ ∈ {*0*,* 1*}*. If actual inputs are not of the same length, abort with error. The calling protocol has to pad if they may not be the same length.

*•* **Auxiliary inputs:**

* Both parties have the security parameter 1*n*
* the description of a group G of *prime order*,
* a generator *g* for the group
* The order of the group, *q*.
* Both parties have a probabilistic polynomial-time algorithm *V*

that checks membership in G (i.e., for every *h*, *V* (*h*) = 1 if and only if *h ∈* G). This is part of the dlog library.

Note: The group can be chosen by *R* (receiver) if not given as auxiliary input. If R chooses the group, then it sends it to S in the first message. S must then check that it receives the description of a group of order q, where q is some prime. (If this is given by the dlog library then this can be an option. Otherwise, always use a fixed dlog group.)

* **Default behavior on wrong input:**
* **Receiver’s Output:**
* **Sender’s output:** nothing

*•* **The protocol:**

1. The receiver *R* chooses *α, β, γ ←R {*1*, . . . , q}* and computes ¯*a* as follows:

a. If *σ* = 0 then ¯*a* = (*gα, gβ, gαβ, gγ*).

b. If *σ* = 1 then ¯*a* = (*gα, gβ, gγ, gαβ*).

1. *R* sends ¯*a* to *S*.
2. Denote the tuple ¯*a* received by *S* by (*x, y, z*0*, z*1).
3. *S* checks that all four values are in the group and that *z*0 *̸*= *z*1.
4. If the elements are not all in the group of if z0=z1, it aborts outputting *error*. (What does abort mean exactly? Do we send an abort message {to the other party? To the higher level protocol?} but the socket stays open or do we also close the connection?) This occurs when the other party cheated. So, can close connection. Need to announce to higher level protocol. If for engineering purposes, you wish to tell the other party, then this is fine too.
5. Otherwise, *S* chooses random *u*0*, u*1*, v*0*, v*1 *←R {*1*, . . . , q}* and computes the following four values (all following operations in the group):

*w*0 = *xu*0 *· gv*0 *k*0 = (*z*0)*u*0 *· yv*0

*w*1 = *xu*1 *· gv*1 *k*1 = (*z*1)*u*1 *· yv*1

1. *S* then encrypts *x*0 under *k*0 and *x*1 under *k*1. In order to do this, a KDF (as defined in the library) is applied to k0 in order to obtain a symmetric key. Any symmetric encryption scheme that is secure for eavesdropping adversaries can then be used. Likewise for k1. We recommend using a simple one-time pad. For this, obtain the appropriate output length from KDF(k0) and XOR the result with x0; likewise for k1.
2. *S* sends *R* the pairs (*w*0*, c*0) and (*w*1*, c*1).
3. *R* check that w0,w1 are in the group and the c0,c1 are binary strings of the same length. If not, sends error as in step 5. If yes, *R* computes *kσ* = (*wσ*)*β* and outputs *xσ* = *cσ XOR KDF*(*kσ*).

# AIR (using any homomorphic encryption) LEAVE TO VERSION 2 IF AT ALL (EXPLAIN IN DOCUMENTATION THAT MORE EXPENSIVE; NEED TO PROVE THAT ENCRYPTED 0/1 AND GENERATE KEYS)

**PROTOCOL 7.2.4 (Private Oblivious Transfer *π′*P OT)**

*•* **Inputs:** The sender has a pair of strings *x*0*, x*1 *∈ {*0*,* 1*}n* and the receiver has

a bit *σ ∈ {*0*,* 1*}*.

*•* **Auxiliary inputs:** Both parties have the security parameter 1*n*.

* **Default behavior on wrong input:**
* **Receiver’s Output:** *sr* = *Dsk*(*c′*)
* **Sender’s output:** nothing

*•* **The protocol:**

1. The receiver *R chooses* a pair of keys (*pk, sk*) *← G*(1*n*) of length greater than n.
2. R computes *c* = *Epk*(*σ*) and sends *c* and *pk* to *S*.
3. The sender *S verifies* that *pk* is a valid public-key and that *c* encrypts either 0 or a value with a multiplicative inverse in the plaintext group *M*.
4. If both checks pass, then *S maps* *x*0 and *x*1 into *M* and then

*Else, what?Abort and send abort to receiver?*

1. uses the homomorphic property of the encryption scheme (can compute *Epk*(*m*1 + *m*2) given *pk*, *c*1 = *Epk*(*m*1) and *c*2 = *Epk*(*m*2) without knowing *m1* and *m2 and scalar multiplication*), and its knowledge of *x*0 and *x*1, to compute two random encryptions *c*0 = *Epk*((1 *− σ*) *· x*0 + *r*0 *· σ*) and *c*1 = *Epk*(*σ · x*1 + *r*1 *·* (1 *− σ*)) where *r*0*, r*1 *←R M* are random elements in the plaintext group.
2. Do we need to check validity of c0 and c1 ? What if *r*0*, r*1 are not from M?
3. *R* computes and outputs *sr* = *Dsk*(*c′*).

We assume that there is a sigma protocol for proving that an encrypted value equals 0 and a sigma protocol that an encrypted value equals 1. (Such protocols exist for Paillier; reference!) Given this, the transformations in the sigma-protocol chapter of the book can be used to prove in zero-knowledge that the encrypted value is either 0 or 1.

# HL-one-sided (using any DH group)

Is any DH group of prime order? No.

Why do we only membership for the simulation protocols and not for protocol 7.2.1? Proof of security; problems of cryptographers…

**PROTOCOL 7.3 (Oblivious Transfer with one-sided simulation)**

*•* **Inputs:** The sender has a pair of strings *x*0*, x*1 *of the same (arbitrary) length* and the receiver has a bit

*σ ∈ {*0*,* 1*}*.

*•* **Auxiliary inputs:**

* Both parties have the security parameter 1*n*
* the description of a group G of *prime order*,
* a generator *g* for the group
* The order of the group, *q*.
* Both parties have a probabilistic polynomial-time algorithm *V*

that checks membership in G (i.e., for every *h*, *V* (*h*) = 1 if and only if *h ∈* G). This is part of the dlog library.

If the group is not given as an auxiliary input, can it be chosen by *P*2?

* **Default behavior on wrong input:**
* **Receiver’s Output:**
* **Sender’s output:** nothing

*•* **The protocol:**

1. The receiver *R* chooses *α, β, γ ←R {*1*, . . . , q}* and computes ¯*a* as follows:

a. If *σ* = 0 then ¯*a* = (*gα, gβ, gαβ, gγ*).

b. If *σ* = 1 then ¯*a* = (*gα, gβ, gγ, gαβ*).

1. *R* sends ¯*a* to *S*
2. Denote the tuple ¯*a* received by *S* by (*x, y, z*0*, z*1).
3. *S* checks that all four values are in the group and that *z*0 *̸*= *z*1.
4. If the elements are not all in the group of if z0=z1, it aborts outputting *error*. Otherwise, continue.
5. R sends zero-knowledge proof of knowledge of *α to S.* For optimization reas*ons t*he first round of zero-knowledge can be sent together with ¯a
6. If R malicious (and proof of knowledge doesn’t work) then S aborts with error.
7. Else continue
8. *S* chooses random *u*0*, u*1*, v*0*, v*1 *←R {*1*, . . . , q}* and computes the following four values:

*w*0 = *xu*0 *· gv*0 *k*0 = (*z*0)*u*0 *· yv*0

*w*1 = *xu*1 *· gv*1 *k*1 = (*z*1)*u*1 *· yv*1

1. *S* then encrypts *x*0 under *k*0 and *x*1 under *k*1 in the same way as protocol 7.2.1 using a KDF.
2. *S* sends *R* the pairs (*w*0*, c*0) and (*w*1*, c*1).
3. *R* check that w0,w1 are in the group and the c0,c1 are binary strings of the same length. If not, sends error as in step 5. Otherwise, *R* computes *kσ* = (*wσ*)*β* and outputs *xσ* = *cσ XOR KDF*(*kσ*).

# HL-full simulation (using any DH group)

**PROTOCOL 7.4.1 (Fully Simulatable Oblivious Transfer *π*OT)**

*•* **Inputs:** The sender has a pair of strings *x*0*, x*1 arbitrary same length and the receiver has a bit

*σ ∈ {*0*,* 1*}*. In this protocol *x*0*, x*1 *∈* G? Or we need to check their lengths?

*•* **Auxiliary inputs:**

* Both parties have the security parameter 1*n*
* The description of a group G of *prime order*, including a generator *g* for the group and its order *q*. (should we check that the given group is of prime order?) If given as auxiliary input then no need. Otherwise, this is as when chosen by P2.
* Both parties have a probabilistic polynomial-time algorithm *V*

that checks membership in G (i.e., for every *h*, *V* (*h*) = 1 if and only if *h ∈* G).

If the group is not given as an auxiliary input, can it be chosen by *P*2? Yes.

* **Default behavior on wrong input:**
* **Receiver’s Output:** *zσ /wσασ*
* **Sender’s output:** nothing

*•* **The protocol:**

1. *R* chooses *α*0*, α*1*, r ←R {*1*, . . . , q}* and computes *h*0 = *gα*0 , *h*1 = *gα*1 and

*a* = *gr*. It also computes *b*0 = *h0r* *· gσ* and *b*1 = *h1r* *· gσ*.

1. *R* sends (*h*0*, h*1*, a, b*0*, b*1) to *S*.
2. *S* checks that all of *h*0*, h*1*, a, b*0*, b*1 *∈* G and if not it aborts.
3. Let *h* = *h*0*/h*1 and *b* = *b*0*/b*1. Then, *R* proves to *S* that (G*, q, g, h, a, b*) is a

Diffie-Hellman tuple, using a zero-knowledge proof of knowledge. Formally,

*R* proves the relation:

*R*DH = { ((G*, q, g, h, a, b*)*, r*) *| a* = *gr* & *b* = *hr* }

1. If *S* accepted the proof in the previous step, it chooses *u*0*, v*0*, u*1*, v*1 *←R*

*{*1*, . . . , q}* and sends (e0,e1) computed as follows:

a. *e*0 = (*w*0*, z*0) where *w*0 = *au*0 *· gv*0 and *z*0 = KDF(*b0u*0  *· h0v*0 *) XOR x*0

b. *e*1 = (*w*1*, z*1) where *w*1 = *au*1 *· gv*1 and *z*1 = KDF(( *b*1 /*g* )*u*1 *· h1v*1) XOR *x*1

1. *R* checks that w0,w1 are in the group. If not, error. If yes, outputs *zσ XORKDF(wσασ)*and *S* outputs nothing.

# PVW\_plain (using any DH group or N-residuosity)

**PROTOCOL 7.5.1 (Another Fully Simulatable Oblivious Transfer)**

*•* **Inputs:** The sender’s input is a pair (*x*0*, x*1) and the receiver’s input is a bit *σ*

*•* **Auxiliary input:** Both parties hold a security parameter 1*n* and (G*, q, g*0),

where G is an efficient representation of a group of order *q* with a generator *g*0,

and *q* is of length *n*. (what does it mean efficient representation? Is q prime?)

* **Default behavior on wrong input:**
* **Receiver’s Output:**
* **Sender’s output:** nothing

check the length of q? abort if not?

*•* **The protocol:**

1. The receiver *R* chooses random values *y, α*0 *←* Z*q* and sets *α*1 = *α*0 + 1.
2. *R* then computes *g*1 = (*g*0)*y*, *h*0 = *g0α*0 and *h*1 = *g1α*1 and
3. *R* sends (*g*1*, h*0*, h*1) to the sender *S*.
4. *R* proves, using a zero-knowledge proof of knowledge, that (*g*0*, g*1*, h*0*, h*1/*g*1 ) is a DH tuple; see Protocol 6.2.4. (6.2.4 is a sigma protocol, is this interchangeable with zero-knowledge protocol?) (How do the input parameters here correspond to the input parameters in 6.2.4?)

*What happens if R doesn’t convince S? Abort?*

1. R chooses a random value *r* and computes *g* = (*gσ*)*r* and *h* = (*hσ*)*r*,
2. R sends (*g, h*) to *S*.
3. The sender operates in the following way:

– Define the function *RAND*(*w, x, y, z*) = (*u, v*), where *u* = (*w*)*s·*(*y*)*t* and

*v* = (*x*)*s·*(*z*)*t*, and the values *s, t ←* Z*q* are random.

– *S* computes (*u*0*, v*0) = *RAND*(*g*0*, g, h*0*, h*), and (*u*1*, v*1) =

*RAND*(*g*1*, g, h*1*, h*).

– *S* sends the receiver the values (*u*0*,w*0) where *w*0 = *v*0*·x*0, and (*u*1*,w*1)

where *w*1 = *v*1*·x*1.

1. The receiver computes *xσ* = *wσ/*(*uσ*)*r*.

# Batch OT HL-full-sim

There are *m* OTs we need to perform.

**PROTOCOL 7.4.3 (Batch Oblivious Transfer πBOT)**

*•* **Inputs:** The sender has a list of m pairs of strings (*x01 , x11* ), . . . , (*x0m, x1m*) and the receiver has an m bits string (*σ1, . . . , σm*).

*•* **Auxiliary inputs:**

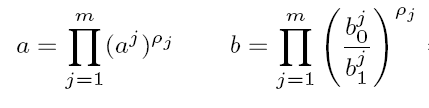
* Both parties have the security parameter 1*n*
* The description of a group G of *prime order*, including a generator *g* for the group and its order *q*.
* Both parties have a probabilistic polynomial-time algorithm *V*

that checks membership in G (i.e., for every *h*, *V* (*h*) = 1 if and only if *h ∈* G).

* **Default behavior on wrong input:**
* **Receiver’s Output:** *zσj /wσjασj* for every j
* **Sender’s output:** nothing

*•* **The protocol:**

1. *R* chooses *α*0*, α*1*, r ←R {*1*, . . . , q}* and computes *h*0 = *gα*0 , *h*1 = *gα*1
2. R proves that it knows the discrete log of h0, using a zero-knowledge proof of knowledge for RDL.
3. For every j = 1, . . . ,m, the receiver R chooses a random rj and computes aj = grj , b0j = h0rj  · gσj and b1j  = h1rj · gσj
4. R sends all these values to S.
5. *S* checks that all the received values are in G and if not it aborts.? (not in the book)
6. S chooses random ρ1, . . . , ρm ←R {1, . . . , q} and sends them to R.
7. Both parties locally compute



1. Then, *R* proves to *S* that (G*, q, g, h, a, b*) is a

Diffie-Hellman tuple, using a zero-knowledge proof of knowledge. Formally,

*R* proves the relation:

*R*DH = { ((G*, q, g, h, a, b*)*,* ) *| a* = & *b* = } (please check)

1. If *S* accepted the proof in the previous step, it chooses *u*0j*, v*0j*, u*1j*, v*1j *←R*

*{*1*, . . . , q}* and computes the following for every j (superscript j is omitted below):

a. *e*0 = (*w*0*, z*0) where *w*0 = *au*0 *· gv*0 and *z*0 = *b0u*0  *· h0v*0 *· x*0

b. *e*1 = (*w*1*, z*1) where *w*1 = *au*1 *· gv*1 and *z*1 = ( *b*1 /*g* )*u*1 *· h1v*1 *· x*1

1. *R* outputs *zσj /wσjασj* for every j.

**PROTOCOL 7.5.2 (Another Batch Fully Simulatable Oblivious Transfer)**

*•* **Inputs:** The sender’s input is a pair (*x*0*, x*1) and the receiver’s input is a bit *σ*

*•* **Auxiliary input: Both** parties hold a security parameter 1*n* and (G*, q, g*0),

where G is an efficient representation of a group of order *q* with a generator *g*0,

and *q* is of length *n*. (what does it mean efficient representation? Is q prime?)

* **Default behavior on wrong input:**
* **Receiver’s Output:**
* **Sender’s output:** nothing

check the length of q? abort if not?

*•* **The protocol:**

1. The receiver *R* chooses random values *y, α*0 *←* Z*q* and sets *α*1 = *α*0 + 1.
2. *R* then computes *g*1 = (*g*0)*y*, *h*0 = *g0α*0 and *h*1 = *g1α*1 and
3. *R* sends (*g*1*, h*0*, h*1) to the sender *S*.
4. *R* proves, using a zero-knowledge proof of knowledge, that (*g*0*, g*1*, h*0*, h*1/*g*1 ) is a DH tuple; see Protocol 6.2.4. (6.2.4 is a sigma protocol, is this interchangeable with zero-knowledge protocol?) (How do the input parameters here correspond to the input parameters in 6.2.4?)

*What happens if R doesn’t convince S? Abort?*

1. For every j: (index j is omitted)
   1. R chooses a random value *r* and computes *g* = (*gσ*)*r* and *h* = (*hσ*)*r*,
   2. R sends (*g, h*) to *S*.
   3. The sender operates in the following way:

– Define the function *RAND*(*w, x, y, z*) = (*u, v*), where *u* = (*w*)*s·*(*y*)*t* and

*v* = (*x*)*s·*(*z*)*t*, and the values *s, t ←* Z*q* are random.

– *S* computes (*u*0*, v*0) = *RAND*(*g*0*, g, h*0*, h*), and (*u*1*, v*1) =

*RAND*(*g*1*, g, h*1*, h*).

– *S* sends the receiver the values (*u*0*,w*0) where *w*0 = *v*0*·x*0, and (*u*1*,w*1)

where *w*1 = *v*1*·x*1.

* 1. The receiver computes *xσ* = *wσ/*(*uσ*)*r*.

# Coin Tossing

## Basic Blum single-coin tossing using any commitment scheme

* **Input:** none
* **Default behavior on wrong input:**
* **Common output:** a bit *b*
* **The protocol:**

1. P1 and P2 choose random bits *b1* and *b2*, respectively
2. P1 commits to the single random bit *b1* using any commitment scheme
3. P2 sends the random bit *b2*  to P1
4. P1 decommits
5. Both parties output XOR of the bits *b1*  and *b2*

**[Lindell01] coin tossing, using Pedersen commitments and DLOG-ZK**

* **Input:** none
* **Default behavior on wrong input:**
* **Common output:** a random string of a given length
* **The protocol:**

|  |
| --- |
| 1. P1 commits to a random element *r* of the group using Pedersen |
| 1. P1 proves in ZKPOK that it knows the committed value (item 4 in sigma) |
| 1. P2 sends a random element *s* of the group |
| 1. P1 sends *r* (without decommitting) |
| 1. P1 proves in ZKPOK that *r* is the committed value (item 6 in sigma) |
| 1. Both parties output KDF(*rs*) of length given. Where is KDF defined? |

**Semi-simulatable coin-tossing**

* **Input:** none
* **Default behavior on wrong input:**
* **Common output:** a random string of a given length
* **The protocol:**

|  |
| --- |
| 1. P1 sends a perfectly-hiding commitment to a random element r of the group (if Pedersen) or a random string of appropriate length (if random-oracle) |
| 1. P2 sends a perfectly-binding commitment to s (e.g., Public-key commit or random-oracle; again string or element appropriately) |
| 1. P1 opens |
| 1. P2 opens |
| 1. Both parties output group operation/XOR of r and s (operation depending) |
|  |

# Secure Pseudorandom Function Evaluation

### Definition of secure pseudorandom function evaluation:

1) P1 has a key *k* to a PRF

2) P2 has an input *x* to PRF

3) They together run a protocol that at the end of it P2 learns the output of PRF

(let’s say *y* = PRF(*k,x*)) but P2 doesn't learn the key *k*  and P1 doesn't learn *y* (the output of PRF) (P1 doesn’t learn *x* either).

The PRF function is defined by:



PROTOCOL 7.6.3 (Private Pseudorandom Function Evaluation πP PRF)

* **Inputs**: The input of P1 is a key *k = (ga0 , a1, . . . , am)* where

*a0, a1, . . . , am* ← *R Zq\**.

Input of P2 is a value *x* of length *m*.

* **Auxiliary inputs**: Both parties have the security parameter 1n and are given

*G* – cyclic group, *q* prime and *g* generator.

* **P1’s output:** nothing
* **P2’s output:** 
* **The protocol:**

1. P1 chooses *m* random values *r1, . . . , rm ←R Zq\** .
2. The parties engage in a 1-out-2 private batch oblivious transfer protocol

πPBOT.

1. In the ith iteration, P1 inputs *y0i = ri* and *y1i= ri · ai* (with multiplication

in Zq\* )

1. P2 enters the bit *σi = xi* where *x = x1, . . . , xm.*
2. If the output of any of the oblivious transfers is ⊥, then both parties output

⊥ and halt.

1. Otherwise, P2’s output from the *m* executions is a series of values

*y1x1, . . . , ymxm.*

1. If any value *yixi* is not in *Zq\** , then P2 redefines it to equal 1.
2. P1 computes 
3. P1 sends ˜g it to P2.
4. P2 computes  and outputs y.

|  |
| --- |
|  |

PROTOCOL 7.6.5 (Fully-Simulatable PRF Evaluation πPRF)

* **Inputs**: The input of P1 is *k = (ga0 , a1, . . . , am)* and the input of P2 is a value

*x* of length *m*.

* **Auxiliary inputs**: Both parties have the security parameter 1n and are given

*G* – cyclic group, *q* prime and *g* generator.

* **The protocol:**

1. P1 chooses *m* random values *r1, . . . , rm ←R Zq\** .
2. The parties engage in a 1-out-2 private batch oblivious transfer protocol

πBOT (fully simulatable).

1. In the ith iteration, P1 inputs *y0i = ri* and *y1i= ri · ai* (with multiplication

in Zq\* )

1. P2 enters the bit *σi = xi* where *x = x1, . . . , xm.*
2. If the output of any of the oblivious transfers is ⊥, then both parties output

⊥ and halt.

1. Otherwise, P2’s output from the *m* executions is a series of values

*y1x1, . . . , ymxm.*

1. If any value *yixi* is not in *Zq\** , then P2 redefines it to equal 1.
2. P1 computes 
3. P1 sends ˜g it to P2.
4. P2 checks if the order of ˜g is *q*. Otherwise aborts with error.
5. P2 computes  and outputs y.