## Content

## Pseudo codes and running examples of the pseudo code already written for each protocol.The aim of this analysis is to find weaknesses and important issues to take into account when designing.

Contents

[Content 1](#_Toc268078717)

[Pseudo codes and running examples of the pseudo code already written for each protocol.The aim of this analysis is to find weaknesses and important issues to take into account when designing. 1](#_Toc268078718)

[1. coin tossing 1](#_Toc268078719)

[Basic Blum single-coin tossing using any commitment scheme 1](#_Toc268078720)

[Basic Blum single-coin tossing using Pedersen commitment scheme 2](#_Toc268078721)

[2. Commitments 4](#_Toc268078722)

[Extractable Pedersen commitment 4](#_Toc268078723)

[3. Sigma protocols 6](#_Toc268078724)

[*Σ* Protocol for Pedersen Commitments 6](#_Toc268078725)

[OR of any two Sigma protocols 6](#_Toc268078726)

[4. ZKPOK 8](#_Toc268078727)

[ZKPOK for every Sigma-protocol using any trapdoor commitment 8](#_Toc268078728)

[ZKPOK for DLOG Sigma-protocol using Pederson commitment 9](#_Toc268078729)

[5. Oblivious Transfers 11](#_Toc268078730)

[Naor-Pinkas (using any DH group) 11](#_Toc268078731)

[6. Batch Oblivious Transfers 12](#_Toc268078732)

[Naor-Pinkas (using any DH group) 12](#_Toc268078733)

[7. Secure Pseudorandom Function Evaluation 14](#_Toc268078734)

[Secure Pseudorandom Function Evaluation 14](#_Toc268078735)

# coin tossing

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

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | | P2 | |
| Input | None | Input | None |
| 1.Protocol | Choose random bits *b1* | 1.Protocol | Choose random bits *b2* |
|  | Commits to the single random bit *b1* using any commitment scheme |  | Participate in commitment |
|  |  |  | Send the random bit *b2* |
|  | Receive random bit |  |  |
|  | Decommit to *b1* |  |  |
|  |  |  | Receive decommit |
| Output | XOR of the bits *b1*  and *b2* | Output | XOR of the bits *b1*  and *b2* |

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

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | | P2 | |
| Input | 1*n* | Input | 1*n* |
| 1.Protocol | Choose random bits *b1*  Note that *b1* =*x* of commit | 1. Protocol | Choose random bits *b2* |
| 1.Commitment | Commits to the single random bit *b1* using any commitment scheme | 1.Commitment | Choose (G*, q, g*) where G is a group of order *q* with generator  *g* and *q >* 2*n* |
| Commitment |  | 2.Commitment | Choose a random *a ←* Z*q*, computes *α* = *ga* |
| Commitment |  | 3.Commitment | Send (G*, q, g, α*) |
| 2.Commitment | Receive (G*, q, g, α*) | Commitment |  |
| 3.Commitment | Verify 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. Report error. | Commitment |  |
| 4.Commitment | choose a random *r ←* Z*q*, compute *c* = *gr · α b1* | Commitment |  |
|  | sends *c* |  |  |
|  |  | 4.Commitment | Receive *c* |
|  |  | 2.protocol | send the random bit *b2* |
| 2.protocol | Receive random bit |  |  |
| 5.Commitment | Decommit to *b1* :  sends (*r, b1*) | Commitment |  |
|  |  | 5.Commitment | Receive decommit |
|  |  | 6.Commitment | verifies that *c* = *gr · α b1* |
| Output | XOR of the bits *b1*  and *b2* | Output | XOR of the bits *b1*  and *b2* |

# Commitments

## Extractable Pedersen commitment

|  |  |  |  |
| --- | --- | --- | --- |
| Comitter | | Receiver | |
| Input | *x ∈ {*0*,* 1*}n* | Input | (*x*0*, x*1) |
| A.Input | 1*n* | | |
| Note |  | | |
| Protocol:  Commit phase |  | 1.Protocol Commit phase | choose (G*, q, g*) where G is a group of order *q* with generator  *g* and *q >* 2*n*. |
|  |  | 2 | Choose a random *a ←* Z*q*, computes *α* = *ga* |
|  |  | 3 | Send (G*, q, g, α*) |
| 1 | Receive (G*, q, g, α*) |  |  |
| 2 | Verify that   1. G is a group of order *q*, 2. *g* is a generator 3. *α ∈* G. Then   If not all the above statements are true. Report error. |  |  |
| 3 | choose a random *r ←* Z*q*, computes *c* = *gr · αx* |  |  |
| 4 | sends *c* |  |  |
| Input to ZKPOK | *w=* (r , x) **from the commitment protocol** |  |  |
| A.Input to ZKPOK | ( G*, q, g, α, c*) | | |
| Relation | **R =** {<(G’*, q’, g’, α’, c’*), (r’,x’)> : *c’* = *g’r’ · α’ x’*} (no need to verify 1. G is a group of order *q*, 2. *g* is a generator and 3.*α ∈* G.) | | |
|  |  | 1.ZKPOK | Commit to *σ ← R*Z*q* |
| 1.ZKPOK | Choose *t1, t2 ←R* Z*q* |  |  |
| 2.ZKPOK | Computes *T*= *gt1 · αt2* |  |  |
| 3.ZKPOK | Send *T* |  |  |
|  |  | 2.ZKPOK | Receive *T* |
|  |  | 3.ZKPOK | Decommit: Send *σ* |
| 4.ZKPOK | Receive *σ*. |  |  |
| 5.ZKPOK | Verify the decommitment and reports error if it is not valid |  |  |
| 6.ZKPOK | Compute:  *s1*= *r · σ +t1* ,  *s2*= *x · σ +t2* |  |  |
| 7.ZKPOK | Send *s1, s2* |  |  |
|  |  | 4.ZKPOK | Receive *s1, s2* |
|  |  | 5.ZKPOK | Accept iff received *s1, s2* such that *gs1 · αs2 =c σ · T* |
| 5 | Send (*r, x*) |  |  |
|  |  | 4 | *c* = *gr · αx* |

# Sigma protocols

## *Σ* Protocol for Pedersen Commitments

|  |  |  |  |
| --- | --- | --- | --- |
| Prover | | Verifier | |
| Input | values *x* and *r* such that *c=gr · αx***.** | Input |  |
| A.Input | (G*, q, g, α*, c) where G is a group of order *q* with generator *g* and *q >* 2*n* and *α* = *ga* for some *a ←* Z*q*. | | |
| Protocol |  | 1.Protocol | check that:   1. G is a group of order *q* 2. *g* is a generator of *G.* 3. *α∈* G 4. *c ∈* G |
| 1 | Choose a random *t ←R* Z*q* and *s ←R* Z*q* and |  |  |
| 2 | Compute *d=αtgs*. |  |  |
| 3 | Send *d* |  |  |
|  |  | 2 | Receive *d* |
|  |  | 3 | Choose a random challenge  *e ← R* Z*q* |
|  |  | 4 | Send *e* |
| 4 | Receive *e* |  |  |
| 5 | Compute the values  *u* = *t* + *ex* mod *q* ,  *v* = *s* + *er*  mod *q* |  |  |
| 6 | Sends *u,v* |  |  |
|  |  | 5 | Receive *u,v* |
|  |  | 6 | Check that *αugv=dce*.  Accepts if and only if the above statement is true. |

## 

## *Σ* Protocol that committed value is as given - Pedersen

|  |  |  |  |
| --- | --- | --- | --- |
| Prover | | Verifier | |
| Input | *r* such that *c=gr · αx***.** | Input |  |
| A.Input | *(G, q, g, α, c, x)* where G is a group of order *q* with generator *g* and *q >* 2*n* , *α* = *ga* for some *a ←* Z*q* and there exist a value *r* such that *c=gr · αx* | | |
| Protocol |  | 1.Protocol | check that:   1. G is a group of order *q* 2. *g* is a generator of *G.* 3. *α∈* G 4. *c ∈* G |
| 1 | Choose a random *s ←R* Z*q* and |  |  |
| 2 | Compute *d=gs*. |  |  |
| 3 | Send *d* |  |  |
|  |  | 2 | Receive *d* |
|  |  | 3 | Choose a random challenge  *e ← R* Z*q* |
|  |  | 4 | Send *e* |
| 4 | Receive *e* |  |  |
| 5 | Compute the values  *u* = *s* + *er* mod |  |  |
| 6 | Sends *u* |  |  |
|  |  | 5 | Receive *u* |
|  |  | 6 | Compute *b=c/ αx* |
|  |  | 7 | Check that *be=gu/gs*.  Accepts if and only if the above statement is true. |

## *Σ* Protocol for El Gamal Commitments

|  |  |  |  |
| --- | --- | --- | --- |
| Prover | | Verifier | |
| Input | *m* and *r* such that *c=(gr,hrm)* | Input |  |
| A.Input | (G*, q, g, h*, c=(c1,c2)) where G is a group of order *q* with generator *g* and *q >* 2*n* and *h* = *ga* for some *a ←* Z*q*. | | |
| Protocol |  | 1.Protocol | check that:   1. G is a group of order *q* 2. *g* is a generator of *G.* 3. *h∈* G 4. *c1 ∈* G 5. *c2 ∈* G |
| 1 | Choose a random *s ←R* Z*q* |  |  |
| 2 | Compute *d=gs* |  |  |
| 3 | Send *d* |  |  |
|  |  | 2 | Receive *d* |
|  |  | 3 | Choose a random challenge  *e ← R* Z*q* |
|  |  | 4 | Send *e* |
| 4 | Receive *e* |  |  |
| 5 | Compute the values  *u* = *s* + *er* mod *q* |  |  |
| 6 | Sends *u* |  |  |
|  |  | 5 | Receive *u,v* |
|  |  | 6 | Check that *c1e= gu/d*.  Accepts if and only if the above statement is true. |

## OR of any two Sigma protocols

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

|  |  |  |  |
| --- | --- | --- | --- |
| Prover | | Verifier | |
| Input | (*x*0*, x*1), a bit *b* such that (*xb,w*) *∈ R* | Input | (*x*0*, x*1) |
| Note |  | | |
| 1.Protocol | Computes the first message *ab* in *π* | Protocol |  |
| 2 | chooses *e*1*−b* at random |  |  |
| 3 | runs the simulator *M* on input (*x*1*−b, e*1*−b*),  Let (*a*1*−b, e*1*−b, z*1*−b*) be the output of *M* |  |  |
| 4 | Send (*a*0*, a*1) |  |  |
|  |  | 1 | Receive (*a*0*, a*1) |
|  |  | 2 | chooses a random *t*-bit string *s* |
|  |  | 3 | sends *s* |
| 5 | Receive *s* |  |  |
| 6 | *eb* = *s⊕e*1*−b* |  |  |
| 7 | *zb* = *π*(*xb, ab, eb,w*) |  |  |
| 8 | Send (*e*0*, z*0*, e*1*, z*1) |  |  |
|  |  | 4 | Receive (*e*0*, z*0*, e*1*, z*1) |
|  |  | 5 | 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. |
|  |  | 6 | If all the above statements are true accept. Otherwise reject. |

# ZKPOK

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

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prover | | Verifier | | |
| Input | (*x,w*) *∈ R* | Input | *x* | |
| Note |  | | | |
| Protocol |  | 1.Protocol | | Chooses a random *t*-bit challenge *e* and interacts with *P* via the trapdoor commitment protocol **com** in order to commit to *e* |
|  |  | 2 | | Commit to *e* |
| 1 | Computes the first message *a* in *π*, using (*x,w*) as input |  | |  |
| 2 | Send *a* |  | |  |
|  |  | 3 | | Decommit to *e* |
| 3 | Verifies the decommitment and reports error if it is not valid |  | |  |
| 4 | Computes the answer *z* to challenge *e* according to the instructions in *π* |  | |  |
| 5 | Sends *z* and the trapdoor **trap** |  | |  |
|  |  | 4 | | Receive trap |
|  |  | 5 | | *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* |

## ZKPOK for DLOG Sigma-protocol using Pederson commitment

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prover | | Verifier | | |
| Input | *x*, (*p, q, g, h*),  *w* such that (*x,w*) *∈ R*  *h=x=gw mod p* | Input | *x*, (*p, q, g, h*), *h=x* | |
| Note | *t* and *n* are known paramets.  *t*- statistical security parameter (for example: 40 bit)  *n*- computational security parameter (for example 1024 bit)  There exists a padding scheme from *t* to *n* | | | |
| 1. Protocol =1.pedersen | Choose *(G',q',g')* | 1.Protocol | | Choose a random *challenge e ←R {*0*,* 1*}t* |
|  |  | 2. | | Pad *e* with a padding scheme  *x = Pad(e)* |
| 2.pedersen | *trap a ←* Z*q*,  computes *α* = *g' trap a* |  | |  |
| 3.pedersen | Send *(G',q',g', α)* |  | |  |
|  |  | 1.pedersen | | Receive *(G',q',g', α)* |
|  |  | 2. pedersen | | Verify *(G',q',g', α)*, if not true report error |
|  |  | 3. pedersen | | *r' ←* Z*q*,  computes *c* = *gr' · αx* |
|  |  | 4. pedersen | | Send *c* (Commit) |
| 4.pedersen | Receive commitment *c* |  | |  |
| 2. Protocol | Computes the first message *a* in *π* as follows | 1. Schnorr | | Checks that:   1. p, q are prime 2. g, h have order q, if not report error |
| 1. Schnorr | *r ←R* Z*q*,  *a* = *gr* mod *p* |  | |  |
| 2. Schnorr | Send *a* |  | |  |
|  |  | 2. Schnorr | | Receive *a* |
|  |  | 3.protocol | | Decommit challeng *e* |
|  |  | 3. Schnorr | | Send *(r',e)* |
| 3. Schnorr | Receive *(r',e)* |  | |  |
| 4. Schnorr + 5.pedersen | Verify decommitment of *e* by checking that *c* = *gr' · αx*. If not report error |  | |  |
| 5. Schnorr | Compute *z* = *r* + *ew* mod *q* |  | |  |
|  | Send *z* and *trap a* |  | |  |
|  |  | 4. Schnorr | | Receive z, trap *a* |
|  |  | 5. Schnorr | | Check that trap *a* is valid by *α* = *g' trap a* |
|  |  |  | | Check that transcript (a,e,z) is acceping in Schnorr with *x*  by checking *gz* = *ahe* mod *p* |
|  |  |  | | If all true, accept. Otherwise report specific error and reject. |

# Oblivious Transfers

## Naor-Pinkas (using any DH group)

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

**OT)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Receiver | | Sender | | |
| Input | a bit *σ ∈ {*0*,* 1*}* | Input | *x*0*, x*1 of the same (arbitrary) length | |
| Note |  |  | If actual inputs are not of the same length, report error. The calling protocol has to pad if they may not be the same length. | |
| A. Input | * Security parameter 1*n* * Description of a group G of *prime order*, * A generator *g* for the group * The order of the group, *q*. * Probabilistic polynomial-time algorithm *V* | | | |
| 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.) | | | |
| 1.Protocol | choose *α, β, γ ←R {*1*, . . . , q}* and computes ¯*a* as follows:   1. If *σ* = 0 then   ¯*a* = (*gα, gβ, gαβ, gγ*).   1. If *σ* = 1 then   ¯*a* = (*gα, gβ, gγ, gαβ*). | Protocol | |  |
| 2 | Send ¯*a* |  | |  |
|  |  | 1 | | Receive ¯*a* |
|  |  | 2 | | Denote the tuple ¯*a* received by (*x, y, z*0*, z*1). |
|  |  | 3 | | 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 or if z0=z1, report error |
|  |  | 5 | | choose 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 |
|  |  | 6 | | Encrypts *x*0 under *k*0 and *x*1 under *k*1.  C0 = KDF(k0) XOR x0;  C1 = KDF(k1) XOR x1; |
|  |  | 7 | | Sendthe pairs (*w*0*, c*0) and (*w*1*, c*1). |
| 3 | Receivethe pairs (*w*0*, c*0) and (*w*1*, c*1). |  | |  |
| 4 | Check that w0,w1 are in the group and the c0,c1 are binary strings of the same length.  If not, report error.  Else, computes *kσ* = (*wσ*)*β* |  | |  |
| Output | *xσ* = *cσ XOR KDF*(*kσ*). | Output | | Nothing |

# Batch Oblivious Transfers

# Naor-Pinkas (using any DH group)

**PROTOCOL (Private Batch Oblivious Transfer *π*PBOT)**

|  |  |  |  |
| --- | --- | --- | --- |
| Receiver | | Sender | |
| Input | *m* bits string (*σ1, . . . , σm*). | Input | list of m pairs of strings (*x01 , x11* ), . . . , (*x0m, x1m*) |
| Note |  |  | If actual inputs are not of the same length, report error. The calling protocol has to pad if they may not be the same length. |
| A. Input | * Security parameter 1*n* * Description of a group G of *prime order*, * A generator *g* for the group * The order of the group, *q*. * Probabilistic polynomial-time algorithm *V* | | |
| 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.) | | |
| 1.Protocol | choose *α, β1,… , βm , γ1,…, γm ←R {*1*, . . . , q}* and computes ¯*a* as follows:  a. If *σi* = 0 then ¯*ai* = (*gβi, gαβi, gγi*).  b. If *σi* = 1 then ¯*ai* = (*gβi, gγi, gαβi*). | Protocol |  |
| 2 | *gα* and ¯*a1,...,* ¯*am* |  |  |
|  |  | 1 | Receive ¯*a* and *gα* |
|  |  | 2 | Denote the tuple ¯*ai* received by *S* by ( *yi, z*0*i, z*1*i*) and *x* = *gα*. |
|  |  | 3 | checks that that all received values are in the group and that *z*0i *̸*= *z*1i for every *i* |
|  |  | 4 | If the elements are not all in the group or if *z*0i *̸*= *z*1i, report error |
|  |  | 5 | choose random *u*0*i, u*1 *i, v*0 *i, v*1 *i* *←R {*1*, . . . , q}* for every *i=1,…,m* and computes the following *4m* values (all following operations in the group):  *w*0 *i* = *xu*0 *i* *· gv*0 *i* *k*0 *i*= (*z*0 *i*)*u*0 *i* *· y i v*0 *i*  *w*1 *i* = *xu*1 *i* *· gv*1 *i* *k*1 *i* = (*z*1 *i*)*u*1 *i* *· y i v*1 *i* |
|  |  | 6 | Encrypts *x*0 *i* under *k*0 *i* and *x*1 *i* under *k*1 *i*.  C0 *i* = KDF(k0 *i*) XOR x0 *i*;  C1 *i* = KDF(k1 *i*) XOR x1 *i*; |
|  |  | 7 | Sendthe *m* pairs  (*w*0 *i, c*0 *i*) and (*w*1 *i, c*1 *i*). |
| 3 | Receivethe m pairs (*w*0 *i, c*0 *i*) and (*w*1 *i, c*1 *i*). |  |  |
| 4 | Check that *w0 i,w1 i* are in the group and the *c0 i,c1 i* are binary strings of the same length. If not, reports error.  Else, computes *kσi i* = (*wσi i*)*βi* |  |  |
| Output | *xσ i* = *cσ i XOR KDF*(*kσi i*) for every *i*. | Output | Nothing |

# Secure Pseudorandom Function Evaluation

## Secure Pseudorandom Function Evaluation

PROTOCOL 7.6.3 (Private Pseudorandom Function Evaluation πP PRF)

|  |  |  |  |
| --- | --- | --- | --- |
| P2 | | P1 | |
| Note | 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: | | |
| Input | *x* of length *m* | Input | key *k = (ga0 , a1, . . . , am)* where *a0, a1, . . . , am* ← *R Zq\**. |
| A Input | 1n and are given *G* – cyclic group, *q* prime and *g* generator. | | |
| Protocol |  | Protocol |  |
|  |  | 1 | choose *m* random values *r1, . . . , rm ←R Zq\** |
| Input to BOT | *m* bits string (*σ1, . . . , σm*).  *σi = xi* where *x = x1, . . . , xm* | Input to BOT | list of m pairs of strings (*y01 ,y11* ), . . . , (*y0m, y1m*)  *y0i = ri* and *y1i= ri · ai* (with multiplication in Zq\* ) |
| A. Input to BOT | * Security parameter 1*n*  - the same * Description of a group G of *prime order*, -the same * A generator *g* for the group – the same * The order of the group, *q*. * Probabilistic polynomial-time algorithm *V (from the library)* | | |
| 1.BOT | choose *α, β1,… , βm , γ1,…, γm ←R {*1*, . . . , q}* and computes ¯*a* as follows:   1. If *σi* = 0 then ¯*ai* = (*gβi, gαβi, gγi*). 2. If *σi* = 1 then ¯*ai* = (*gβi, gγi, gαβi*). | Protocol |  |
| 2.BOT | Send *gα* and ¯*a1,...,* ¯*am* |  |  |
|  |  | 1.BOT | Receive ¯*a* and *gα* |
|  |  | 2.BOT | Denote the tuple ¯*ai* received by *S* by ( *si, z*0*i, z*1*i*) and *t* = *gα*. |
|  |  | 3.BOT | checks that that all received values are in the group and that *z*0i *̸*= *z*1i for every *i* |
|  |  | 4.BOT | If the elements are not all in the group or if *z*0i *̸*= *z*1i, report error |
|  |  | 5.BOT | choose random *u*0*i, u*1 *i, v*0 *i, v*1 *i* *←R {*1*, . . . , q}* for every *i=1,…,m* and computes the following *4m* values (all following operations in the group):  *w*0 *i* = *tu*0 *i* *· gv*0 *i* *k*0 *i*= (*z*0 *i*)*u*0 *i* *· si v*0 *i*  *w*1 *i* = *tu*1 *i* *· gv*1 *i* *k*1 *i* = (*z*1 *i*)*u*1 *i* *· si v*1*i* |
|  |  | 6.BOT | Encrypts *y*0 *i* under *k*0 *i* and *y*1 *i* under *k*1 *i*.  *C0 i = KDF(k0 i) XOR y0 i;*  *C1 i = KDF(k1 i) XOR y1 i;* |
|  |  | 7.BOT | Sendthe *m* pairs  (*w*0 *i, c*0 *i*) and (*w*1 *i, c*1 *i*). |
| 3.BOT | Receivethe m pairs (*w*0 *i, c*0 *i*) and (*w*1 *i, c*1 *i*). |  |  |
| 4.BOT | Check that *w0 i,w1 i* are in the group and the *c0 i,c1 i* are binary strings of the same length. If not, reports error.  Else, computes *kσi i* = (*wσi i*)*βi* |  |  |
|  | *yσ i* = *cσ i XOR KDF*(*kσi i*) for every *i*. which is *y1x1, . . . , ymxm* |  |  |
|  | If any value *yixi* is not in *Zq\** , then P2 redefines it to equal 1 |  |  |
|  |  | 2 | CAN BE DONE BEFORE IN PARALLEL  compute |
|  |  |  | Send ˜g |
| Output |  | Output | Nothing |

1. **List of algorithms in one-sided pseudo-code**

In this list the highlighted protocols are written and those that are not highlighted can only be find in pseudo-code that is one-sided.

**Standard commitment schemes** [EREZ]

1. Hash-based (random-oracle) commitments: define Commit(x) = HASH(x||r) where r is 128-bits random
2. Hash-based commitments: <http://cs.nyu.edu/courses/fall01/G22.3033-003/lect/lecture14.ps> , Section 2.3
3. Public-key encryption based commitments: commit to x by choosing new (pk,sk) and sending (pk,E(pk,x))
4. UC-secure commitment <http://eprint.iacr.org/2001/091>

**Trapdoor (equivocal) commitment schemes**

1. General transformation from Sigma protocol where simulator instructions are provided.

**Extractable commitment schemes** [ERAN]

1. Any commitment with a ZKPOK of committed value (Pedersen)
2. Others?

**Homomorphic commitment schemes** [ERAN]

1. Take from LEGO: <http://eprint.iacr.org/2008/427.pdf>

**Sigma protocol** [HILA except for what is marked BENNY]

1. Sigma protocols for Damgard-Jurik [BENNY]
2. Sigma of committed value for Pedersen commitments [???]
3. Sigma of committed value for ElGamal commitments [???]
4. Sigma that committed value is as given - Pedersen [???]
5. Sigma that committed value is as given - El Gamal [???]
6. General compound statements [BENNY]

**Zero knowledge** [EREZ]

1. Fiat-Shamir transform for any Sigma protocol: just get verifier message by HASH(x,\alpha)

**Oblivious transfer** [HILA]

1. PVW\_UC (using any DH group or N-residuosity) [PVW], <http://www.cc.gatech.edu/~cpeikert/pubs/OTpaper.pdf>

**Oblivious polynomial evaluation** [BENNY]

1. Based on OT [BENNY]
2. Based on homomorphic encryption [BENNY]

**Information theoretic techniques** [GILAD] (take specification from VIFF)

1. Secret sharing (including general polynomial interpolation)
   1. Class that can work over any field
   2. Use Zp\* as basic field (here p can be small)
2. VSS
3. Arithmetic circuit protocols (addition, multiplication…)

**Standard commitment schemes Standard commitment schemes**

1. Pedersen commitments: <http://cs.nyu.edu/courses/fall01/G22.3033-003/lect/lecture14.ps> , Section 2.5

**Trapdoor (equivocal) commitment schemes**

1. Based on DLOG Sigma protocol [HL, Section 6.6, use DLOG sigma as basis]

**Extractable commitment schemes**

**Homomorphic commitment schemes**

**Sigma protocol**

1. Sigma protocol of DLOG [HL, Section 6.1]
2. Sigma protocol of DH tuple [HL, Section 6.2]
3. Template for Sigma protocol – programmer fills in procedures as below and Sigma protocol is built automatically:
   1. Prover compute 1st message
   2. Prover compute 2nd message
   3. Verifier check
   4. Verifier query length
4. AND of any number of Sigma protocols [HL, Section 6.4]
5. OR of any two Sigma protocols [HL, Section 6.4]

**Zero knowledge**

1. Zero-knowledge for every Sigma-protocol using any commitment [HL, Section 6.5.1]
2. ZKPOK for every Sigma-protocol using any trapdoor commitment [HL, Section 6.5.2] (Pederson)

**Coin tossing**

1. Basic Blum single-coin tossing using any commitment scheme
   1. P1 commits to a single random bit using any commitment scheme
   2. P2 sends a random bit to P1
   3. P1 decommits
   4. Both parties output XOR of bits
2. [Lindell01] coin tossing, using Pedersen commitments and DLOG-ZK
   1. P1 commits to a random r using Pedersen
   2. P1 proves in ZKPOK that it knows the committed value (item 4 in sigma)
   3. P2 sends a random s
   4. P1 sends r (without decommitting)
   5. P1 proves in ZKPOK that r is the committed value (item 6 in sigma)
   6. Both parties output XOR of r and s
3. Semi-simulatable coin-tossing
   1. P1 sends a perfectly-hiding commitment to r (e.g. Pedersen or random-oracle)
   2. P2 sends a perfectly-binding commitment to s (e.g., Public-key commit or random-oracle)
   3. P1 opens
   4. P2 opens
   5. Both parties output XOR of r and s

**Oblivious transfer**

1. Naor-Pinkas (using any DH group) [HL, Section 7.2.1]
2. AIR (using any homomorphic encryption) [HL, Section 7.2.2]
3. HL-one-sided (using any DH group) [HL, Section 7.3]
4. HL-full simulation (using any DH group) [HL, Section 7.4]
5. PVW\_plain (using any DH group or N-residuosity) [HL, Section 7.5] and [PVW]

**Batch OT**

1. HL-full-sim [HL, Section 7.4.2]
2. PVW-batch [HL, Section 7.5]
3. Naor-Pinkas

**Oblivious polynomial evaluation**

**Oblivious pseudorandom function evaluation**

1. Based on OT [HL, Section 7.6]

**Authenticated communication channels**: use openSSL

**Private communication channels**: use openSSL

**Authenticated broadcast channel**

As in document by Meital