|  |  |
| --- | --- |
| Protocol Name: |  |
| Protocol Reference: | [Reference number or label] |
| Protocol Type: | [Not always relevant] |
| Protocol Description: |  |
| References: | [Give reference in book or paper] |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | [e.g., P1,P2 or P1,…,Pn, or Committer/Receiver etc. |
| Parties’ Inputs: | [Put each party’s input on a separate line] |
| Parties’ Outputs: | [Put each party’s input on a separate line] |

The ***protocol specification*** describes the instructions of each party as part of the *interaction*:

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Party 1): |  |
| Step 2 (Party 2): |  |
| Step 3 (Party 3): |  |

The party’s specification describes the instructions of the party from its own point of view (and thus WAIT-FOR-MESSAGE is an often-used instruction here):

|  |  |
| --- | --- |
| Party 1’s Specification | |
| Step 1: |  |
| Step 2: |  |
| Step 3: |  |

|  |  |
| --- | --- |
| Party 2’s Specification | |
| Step 1: |  |
| Step 2: |  |
| Step 3: |  |

Please use the following conventions:

1. All math variables are in bold and italics
2. All math symbols are bold (but not italicized)
3. All descriptions should use the same language and style.
4. A preliminary section should appear explaining what a DLOG description is, and other general remarks.
5. Make sure to use the same notation for sampling values etc. as below
6. Make sure to use consistent, formal expressions. For example, when calling a subprotocol, state: “RUN subprotocol SIGMA\_DLOG with input (g,h) as Verifier”.

The code comments must minimally include the specification.

# Conventions

Include conventions that repeat themselves. For example:

1. A DLOG description (G,q,g) is … We stress that all operations are in the group, unless otherwise explicitly stated.
2. WAIT is not a blocking operation. Other operations may be run in parallel. For sampling a random variable that does not depend on the received message.

If there nothing to do in parallel the operation WAIT will be put in different step. Otherwise, the WAIT operation may be put with the next to come operations. For example:

|  |  |
| --- | --- |
| Step 2: | WAIT for message *e* from V |
| Step 3: | COMPUTE ***z* = *r* + *ew* mod *q***  SEND ***z*** to V |

The party cannot do anything before receiving ***e***, thus this WAIT operation is put in a different step.

|  |  |
| --- | --- |
| Step 2: | WAIT for message *a* from P  SAMPLE a random challenge *e* ∈{0, 1}*t*  SEND *e* to P |

On the other hand, the above party can sample ***e* ∈{0, 1}*t*** while waiting for ***a*** since the two variables are not depended.

# Sigma protocols

|  |  |
| --- | --- |
| Protocol Name: | Schnorr’s Protocol for DLOG |
| Protocol Reference: | SIGMA\_DLOG |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a prover to convince a verifier that it knows the discrete log of a given value in a given group |
| References: | Protocol 6.1.1, page 148 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Parties’ Inputs: | * Common input: (***G,q,g,h***) where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * P’s private input: a value ***w*∈ *Zq*** such that ***h*=*gw*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (both): | V: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q*** * ***h*** ∈ ***G***   OUTPUT REJ  P: SAMPLE a random ***r*∈ *Zq*** and COMPUTE ***a* = *gr***  SEND ***a*** to V |
| Step 2 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3 (P): | COMPUTE ***z* = *r* + *ew* mod *q***  SEND ***z*** to V |
| Step 4 (both): | V: IF ***gz* = *ahe* mod *p***  OUTPUT ACC  ELSE  OUTPUT REJ  P: output nothing |

|  |  |
| --- | --- |
| Prover (P) Specification | |
| Step 1: | SAMPLE a random ***r* ∈ *Zq*** and COMPUTE ***a* = *gr***  SEND ***a*** to V |
| Step 2: | WAIT for message ***e*** from V |
| Step 3: | COMPUTE ***z* = *r* + *ew* mod *q***  SEND ***z*** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| Verifier (V) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q*** * ***h*** ∈ ***G***   OUTPUT REJ |
| Step 2: | WAIT for message ***a*** from P  SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3: | WAIT for a message ***z*** from P  IF ***gz* = *ahe* mod *p***  OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| Protocol Name: | *Σ* Protocol for Diffie-Hellman Tuples |
| Protocol Reference: | SIGMA\_DH |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a prover to convince a verifier that a given tuple is a Diffie-Helman tuple. |
| References: | Protocol 6.2.4, page 152 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Parties’ Inputs: | * Common input: (***G,q,g,h,u,v***) where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * P’s private input: a value ***w*∈ *Zq*** such that ***u*=*gw*** and ***v*=*hw*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (both): | V: IF **NOT**   * ***q*** is prime * ***g***,***h*** are not of order ***q*** * ***u,v*** ∈ ***G***   OUTPUT REJ  P: SAMPLE a random ***r*∈ *Zq*** and COMPUTE ***a* = *gr*** and ***b* = *hr***  SEND ***(a,b)*** to V |
| Step 2 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3 (P): | COMPUTE ***z* = *r* + *ew* mod *q***  SEND ***z*** to V |
| Step 4 (both): | V: IF ***gz* = *aue* mod *p*** and ***hz* = *ave***  OUTPUT ACC  ELSE  OUTPUT REJ  P: output nothing |

|  |  |
| --- | --- |
| Prover (P) Specification | |
| Step 1: | SAMPLE a random ***r*∈ *Zq*** and COMPUTE ***a* = *gr*** and ***b* = *hr***  SEND ***(a,b)*** to V |
| Step 2: | WAIT for message ***e*** from V |
| Step 3: | COMPUTE ***z* = *r* + *ew* mod *q***  SEND ***z*** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| Verifier (V) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q*** * ***h*** ∈ ***G***   OUTPUT REJ |
| Step 2: | WAIT for message ***(a,b)*** from P  SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3: | WAIT for a message ***z*** from P  IF ***gz* = *aue* mod *p*** and ***hz* = *ave***  OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| Protocol Name: | *Σ Protocol that committed value is as given - Pedersen* |
| Protocol Reference: | SIGMA\_COMMITTED\_VALUE\_PEDERSEN |
| Protocol Type: | Sigma Protocol |
| Protocol Description: |  |
| References: |  |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Parties’ Inputs: | * Common input: ***(G, q, g, α, c, x)*** where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** , ***α* = *ga*** for some ***a ←* Z*q*** and there exist a value ***r*** such that ***c=gr · αx*** * P’s private input: a value ***r*** such that ***c=gr · αx*.** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (both): | V: IF **NOT**   * ***q*** is prime * ***g*** is not of order ***q*** * ***α,c*** ∈ ***G***   OUTPUT REJ  P: SAMPLE a random ***s*∈ *Zq*** and COMPUTE ***d=gs***  SEND ***d*** to V |
| Step 2 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3 (P): | COMPUTE ***u* = *s* + *er*****mod *q***  SEND ***u*** to V |
| Step 4 (both): | V: COMPUTE ***b=c/ αx***  IF ***be=gu/gs* mod *p***  OUTPUT ACC  ELSE  OUTPUT REJ  P: output nothing |

|  |  |
| --- | --- |
| Prover (P) Specification | |
| Step 1: | SAMPLE a random ***s*∈ *Zq*** and COMPUTE ***d=gs***  SEND ***d*** to V |
| Step 2: | WAIT for message ***e*** from V |
| Step 3: | COMPUTE ***u* = *s* + *er*****mod *q***  SEND ***u*** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| Verifier (V) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is not of order ***q*** * ***α,c*** ∈ ***G***   OUTPUT REJ |
| Step 2: | WAIT for message ***d*** from P  SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3: | WAIT for a message ***u*** from P  COMPUTE ***b=c/ αx***  IF ***be=gu/gs* mod *p***  OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol for Pedersen Commitments |
| Protocol Reference: | SIGMA\_FOR\_PEDERSEN |
| Protocol Type: | Sigma Protocol |
| Protocol Description: |  |
| References: |  |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Parties’ Inputs: | * Common input: ***(G, q, g, α, c)*** where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** , ***α* = *ga*** for some ***a ←* Z*q***. * P’s private input: values ***x*** and ***r***such that ***c=gr · αx*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (both): | V: IF **NOT**   * ***q*** is prime * ***g*** is not of order ***q*** * ***α,c*** ∈ ***G***   OUTPUT REJ  P: SAMPLE random values ***t* ∈ *Zq*** and ***s*∈ *Zq*** and COMPUTE ***d=******αtgs***  SEND ***d*** to V |
| Step 2 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3 (P): | COMPUTE ***u* = *t* + *ex*****mod *q***and ***v* = *s* + *er*****mod *q***  SEND ***u,v*** to V |
| Step 4 (both): | V: IF ***αugv=dce***  OUTPUT ACC  ELSE  OUTPUT REJ  P: output nothing |

|  |  |
| --- | --- |
| Prover (P) Specification | |
| Step 1: | SAMPLE random values ***t* ∈ *Zq*** and ***s*∈ *Zq*** and COMPUTE ***d=******αtgs***  SEND ***d*** to V |
| Step 2: | WAIT for message ***e*** from V |
| Step 3: | COMPUTE ***u* = *t* + *ex*****mod *q***and ***v* = *s* + *er*****mod *q***  SEND ***u,v*** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| Verifier (V) Specification | |
| Step 1: | : IF **NOT**   * ***q*** is prime * ***g*** is not of order ***q*** * ***α,c*** ∈ ***G***   OUTPUT REJ |
| Step 2: | WAIT for message ***d*** from P  SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3: | WAIT for a message ***u,v*** from P  IF ***αugv=dce***  OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| Protocol Name: | Σ protocol for El Gamal Commitments |
| Protocol Reference: | SIGMA\_FOR\_ELGAMAL |
| Protocol Type: | Sigma Protocol |
| Protocol Description: |  |
| References: |  |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Parties’ Inputs: | * Common input: ***(G, q, g, h, c=(c1,c2))*** where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** , ***h* = *ga*** for some ***a ←* Z*q***. * P’s private input: values ***m*** and ***r***such that ***c=(gr,hrm)*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (both): | V: IF **NOT**   * *g* is a generator of *G* * ***c*1*,c*2*,h*∈ *G***   OUTPUT REJ  P: SAMPLE a random value ***s*∈ *Zq*** and COMPUTE ***d=******gs***  SEND ***d*** to V |
| Step 2 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3 (P): | COMPUTE ***u* = *s* + *er*****mod *q***  SEND ***u*** to V |
| Step 4 (both): | V: IF ***c1e= gu/d***  OUTPUT ACC  ELSE  OUTPUT REJ  P: output nothing |

|  |  |
| --- | --- |
| Prover (P) Specification | |
| Step 1: | SAMPLE a random value ***s*∈ *Zq*** and COMPUTE ***d=******gs***  SEND ***d*** to V |
| Step 2: | WAIT for message ***e*** from V |
| Step 3: | COMPUTE ***u* = *s* + *er*****mod *q***  SEND ***u*** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| Verifier (V) Specification | |
| Step 1: | IF **NOT**   * *g* is a generator of *G* * ***c*1*,c*2*,h*∈ *G***   OUTPUT REJ |
| Step 2: | WAIT for message ***d*** from P  SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3: | WAIT for a message ***u***from P  IF ***c1e= gu/d***  OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| Protocol Name: | AND Protocol for Relations *R0*, *R1* Based on *π0 and π1* |
| Protocol Reference: | AND\_SIGMA |
| Protocol Type: | AND of any number of Sigma protocols |
| Protocol Description: | This protocol is used for a prover to convince a verifier that the AND of two statements are true. |
| References: | page 158 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Parties’ Inputs: | * Common input: pair ***(x0, x1)*** * P’s private input: a pair ***(w0, w1)*** such that ***(x0, w0) ∈ R0*** *and*  ***(x1, w1) ∈ R1***. (it might be that ***R0***= ***R1***) |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (P): | P: SEND ***(a0,a1)*** to V |
| Step 2 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3 (P): | SEND ***(z0,z1)*** to V |
| Step 4 (both): | V: IF   * transcript ***(a0, e, z0)*** is accepting in **π0**, on inputs ***x0*** * transcript***(a1, e, z1)*** is accepting in ***π1***, on inputs ***x1***.   OUTPUT ACC  ELSE  OUTPUT REJ  P: output nothing |
|  |  |

|  |  |
| --- | --- |
| Prover (P) Specification | |
| Step 1: | SEND ***(a0,a1)*** to V |
| Step 2: | WAIT for message ***e*** from V |
| Step 3: | SEND ***(z0,z1)*** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| Verifier (V) Specification | |
| Step 1: |  |
| Step 2: | WAIT for message ***(a0,a1)*** from P  SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3: | WAIT for a message ***(z0,z1)*** from P  IF   * transcripts ***(a0, e, z0)*** is accepting in **π0**, on inputs ***x0*** * transcript***(a1, e, z1)*** is accepting in ***π1***, on inputs ***x1***.   OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| Protocol Name: | OR Protocol for Relation *R* Based on *π* |
| Protocol Reference: | OR\_SIGMA |
| Protocol Type: | OR of two Sigma protocols |
| Protocol Description: | This protocol is used for a prover to convince a verifier that it knows a witness to one of the relations without revealing which one. |
| References: | Protocol 6.4.1, page 159 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Parties’ Inputs: | * Common input: pair ***(x0, x1)*** * P’s private input: a value ***w***and a bit ***b***such that **(*xb,w*) *∈ R*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (P): | P: COMPUTE the first message *ab* in *π*, using (*xb,w*) as input  SAMPLE a random challenge ***e*1*−b* ∈{0, 1}*t***  RUN simulator ***M***on input ***(x1−b, e1−b)***  The output of ***M*** is computed as follows:   * 1. Choose***z1−b*** at random from group ***G***. (e.g. in DL it is ***z ←R Zp\****)   2. Choose***e1−b ←R {0, 1}t***   3. Calculate ***a1−b*** as a function of***(e*1*−b ,z*1*−b* )**. (e.g. in DL it is   ***a1−b = gzh− e1−b***mod *p*)  SEND ***(a0,a1)*** to V |
| Step 2 (V): | SAMPLE a random challenge ***s* ∈{0, 1}*t***  SEND ***s*** to P |
| Step 3 (P): | SET ***eb = s⊕e1−b***  COMPUTE the answer ***zb***in *π* to challenge ***eb***using **(*xb, ab, eb,w*)** as input  SEND ***(e0, z0, e1, z1)*** to V |
| Step 4 (both): | V: IF   * ***e*0 *⊕ e*1 = *s*** * transcripts ***(a0, e, z0)*** is accepting in **π0**, on inputs ***x0*** * transcript***(a1, e, z1)*** is accepting in ***π1***, on inputs ***x1***.   OUTPUT ACC  ELSE  OUTPUT REJ  P: output nothing |
|  |  |

|  |  |
| --- | --- |
| Prover (P) Specification | |
| Step 1: | COMPUTE the first message *ab* in *π*, using (*xb,w*) as input  SAMPLE a random challenge ***e*1*−b* ∈{0, 1}*t***  RUN simulator ***M***on input ***(x1−b, e1−b)***  The output of ***M*** is computed as follows:   * 1. Choose***z1−b*** at random from group ***G***. (e.g. in DL it is ***z ←R Zp\****)   2. Choose***e1−b ←R {0, 1}t***   3. Calculate ***a1−b*** as a function of***(e*1*−b ,z*1*−b* )**. (e.g. in DL it is   ***a1−b = gzh− e1−b***mod *p*)  SEND ***(a0,a1)*** to V |
| Step 2: | WAIT for message ***s*** from V |
| Step 3: | SET ***eb = s⊕e1−b***  COMPUTE the answer ***zb***in *π* to challenge ***eb***using **(*xb, ab, eb,w*)** as input  SEND ***(e0, z0, e1, z1)*** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| Verifier (V) Specification | |
| Step 1: | Validate parameters according to ***π*** |
| Step 2: | WAIT for message ***(a0,a1)*** from P  SAMPLE a random challenge ***s* ∈{0, 1}*t***  SEND ***s*** to P |
| Step 3: | WAIT for a message ***(e0, z0, e1, z1)***from P  V: IF   * ***e*0 *⊕ e*1 = *s*** * transcript ***(a0, e, z0)*** is accepting in **π0**, on inputs ***x0*** * transcript***(a1, e, z1)*** is accepting in ***π1***, on inputs ***x1***.   OUTPUT ACC  ELSE  OUTPUT REJ |

# Zero-knowledge

|  |  |
| --- | --- |
| Protocol Name: | Zero-knowledge for every Sigma-protocol using any commitment |
| Protocol Reference: | ZERO\_KNOWLEDGE\_FOR\_SIGMA\_AND\_COMMIT |
| Protocol Type: | Zero-knowledge Protocol |
| Protocol Description: | This protocol is used for a prover to convince a verifier that it knows a witness without revealing anything about witness |
| References: | Protocol 6.5.1, page 161 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Parties’ Inputs: | * Common input: ***x*** * P’s private input: a value ***w***such that **(*x,w*) *∈ R.*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (V): | V: SAMPLE a random challenge ***e* ∈{0, 1}*t***  RUN subprotocol COMMIT.commit(***e***) |
| Step 2 (P): | COMPUTE the first message ***a***in ***π***, using **(*x,w*)** as input  SEND ***a*** to V |
| Step 3 (V): | RUN subprotocol COMMIT.decommit(***e***) |
| Step 4 (P): | IF COMMIT.decommit(***e***)==VALID  COMPUTE the answer ***z***to challenge ***e***according to the  instructions in ***π***  SEND ***z*** to V  ELSE  REPORT ERROR |
| Step 5 (Both): | V: IF transcript **(*a, e, z*)** is accepting in ***π***on input ***x***  OUTPUT ACC  ELSE  OUTPUT REJ  P: output nothing |

|  |  |
| --- | --- |
| Prover (P) Specification | |
| Step 1: | WAIT for COMMIT.commit(***e***) from V |
| Step 2: | COMPUTE the first message ***a***in ***π***, using **(*x,w*)** as input  SEND ***a*** to V |
| Step 3: | WAIT for COMMIT.decommit(***e***) from V |
| Step 4: | IF COMMIT.decommit(***e***)==VALID  COMPUTE the answer ***z***to challenge ***e***according to the  instructions in ***π***  SEND ***z*** to V  ELSE  REPORT ERROR |
| Step 5: | output nothing |

|  |  |
| --- | --- |
| Verifier (V) Specification | |
| Step 1: | SAMPLE a random challenge ***e* ∈{0, 1}*t***  RUN subprotocol COMMIT.commit(***e***) |
| Step 2: | WAIT for message ***a*** from P |
| Step 3: | RUN subprotocol COMMIT.decommit(***e***) |
| Step 4: | WAIT for message ***z*** from P |
| Step 5: | V: IF transcript **(*a, e, z*)** is accepting in ***π***on input ***x***  OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| Protocol Name: | Zero-knowledge proof of knowledge for every Sigma-protocol using any commitment |
| Protocol Reference: | ZKPOK\_FOR\_SIGMA\_AND\_COMMIT |
| Protocol Type: | Zero-knowledge proof of knowledge Protocol |
| Protocol Description: | This protocol is used for a prover to convince a verifier that it knows a witness without revealing anything about witness. It is also a proof of knowledge protocol. |
| References: | Protocol 6.5.4, page 165 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Parties’ Inputs: | * Common input: ***x*** * P’s private input: a value ***w***such that **(*x,w*) *∈ R.*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (V): | V: SAMPLE a random challenge ***e* ∈{0, 1}*t***  RUN subprotocol COMMIT\_TRAP.commit(***e***) |
| Step 2 (P): | COMPUTE the first message ***a***in ***π***, using **(*x,w*)** as input  SEND ***a*** to V |
| Step 3 (V): | RUN subprotocol COMMIT.decommit(***e***) |
| Step 4 (P): | IF COMMIT\_TRAP.decommit(***e***)==VALID  COMPUTE the answer ***z***to challenge ***e***according to the  instructions in ***π***  SEND ***z*** and trapdoor **trap** to V  ELSE  REPORT ERROR |
| Step 5 (Both): | V: IF   * transcript **(*a, e, z*)** is accepting in ***π***on input ***x*** * trapdoor **trap** is valid   OUTPUT ACC  ELSE  OUTPUT REJ  P: output nothing |

|  |  |
| --- | --- |
| Prover (P) Specification | |
| Step 1: | WAIT for COMMIT.commit(***e***) from V |
| Step 2: | COMPUTE the first message ***a***in ***π***, using **(*x,w*)** as input  SEND ***a*** to V |
| Step 3: | WAIT for COMMIT.decommit(***e***) from V |
| Step 4: | IF COMMIT.decommit(***e***)==VALID  COMPUTE the answer ***z***to challenge ***e***according to the  instructions in ***π***  SEND ***z*** and trapdoor **trap** to V  ELSE  REPORT ERROR |
| Step 5: | output nothing |

|  |  |
| --- | --- |
| Verifier (V) Specification | |
| Step 1: | SAMPLE a random challenge ***e* ∈{0, 1}*t***  RUN subprotocol COMMIT.commit(***e***) |
| Step 2: | WAIT for message ***a*** from P |
| Step 3: | RUN subprotocol COMMIT.decommit(***e***) |
| Step 4: | WAIT for message ***z*** and trapdoor **trap** from P |
| Step 5: | IF   * transcript **(*a, e, z*)** is accepting in ***π***on input ***x*** * trapdoor **trap** is valid   OUTPUT ACC  ELSE  OUTPUT REJ |

# Commitment schemes

|  |  |
| --- | --- |
| Protocol Name: | Pedersen commitment |
| Protocol Reference: | COMMIT\_TRAP\_DLOG |
| Protocol Type: | Trapdoor Commitment |
| Protocol Description: | The committer C commits to a value to the receiver V while keeping it hidden, with the ability to reveal the committed value later. |
| References: | Protocol 6.5.3, page 164 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Parties’ Inputs: | * Common input: : (***G,q,g***) where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * C’s private input: a value ***x ∈ {*0*,* 1*}t*** |
| Parties’ Outputs: | * C: nothing * R: ACC or REJ and **trap** *a* |

|  |  |
| --- | --- |
| Protocol Specification | |
| Commit phase | |
| Step 1 (Both): | C: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR  R: SAMPLE a random challenge ***a ∈ Zq***  COMPUTE ***α* = *ga***  SEND ***α*** to C |
| Step 2 (Both): | C: SAMPLE a random challenge ***r ∈ Zq***  COMPUTE ***c* = *gr · αx***  SEND ***c***  R: IF **NOT** ***α*** ∈ ***G***  REPORT ERROR |
| Decommit phase | |
| Step 3 (C): | SEND **(*r, x*)** to R |
| Step 4 (Both): | R: IF ***c* = *gr · αx***  OUTPUT ACC and trap ***a***  ELSE  OUTPUT REJ  C: output nothing |
|  |  |

|  |  |
| --- | --- |
| Committer (C) Specification | |
| Commit phase | |
| Step 1: | WAIT for ***α*** from R  IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR  SAMPLE a random challenge ***r ∈ Zq*** |
| Step 2: | COMPUTE ***c* = *gr · αx***  SEND ***c*** |
| Decommit phase | |
| Step 3: | SEND **(*r, x*)** to R |
| Step 4: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Commit phase | |
| Step 1: | SAMPLE a random challenge ***a ∈ Zq***  COMPUTE ***α* = *ga***  SEND ***α*** to C |
| Step 2: | WAIT for message ***c*** from C |
| Decommit phase | |
| Step 3: | WAIT for **(*r, x*)** from C |
| Step 4: | IF ***c* = *gr · αx***  OUTPUT ACC and trap ***a***  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| Protocol Name: | Trapdoor (equivocal) commitment schemes - Commitment from Σ-Protocol |
| Protocol Reference: | COMMIT\_TRAP\_FROM\_SIGMA |
| Protocol Type: | Trapdoor Commitment |
| Protocol Description: | The committer C commits to a value to the receiver V while keeping it hidden, with the ability to reveal the committed value later. |
| References: | Protocol 6.6.2, page 174 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Parties’ Inputs: | * Common input: 1*t* * C’s private input: a value ***e ∈ {*0*,* 1*}t*** |
| Parties’ Outputs: | * C: nothing * R: ACC or REJ and **trap *w*** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Commit phase | |
| Step 1 (R): | RUN the generator *G* on input **1*t***to obtain **(*x,w*) *∈ R***  SEND ***x***to *C*. |
| Step 2 (C): | IF ***x ∉ LR***  REPORT ERROR  RUN subprotocol SIGMA.simulator with input (***x, e***) and obtains a transcript **(*a, e, z*)**.  The output of *M* is computed as follows:   1. SAMPLE at random ***z ∈ G***. (e.g in DLOG it is ***z ∈* Zp*\****) 2. COMPUTE ***a*** as a function of***( e ,z)***. (e.g in DLOG it is   ***a* = *gzh− e* mod *p***)  SEND ***a*** to R |
| Decommit phase | |
| Step 3 (C): | SEND **(*e, z*)** to R |
| Step 4 (Both): | R: IF **(*a, e, z*)** is an accepting transcript in *π* with respect to input ***x***  OUTPUT ACC and trap ***w***  ELSE  OUTPUT REJ  C: output nothing |
|  |  |

|  |  |
| --- | --- |
| Committer (C) Specification | |
| Commit phase | |
| Step 1: | WAIT for ***x*** from R |
| Step 2: | IF ***x ∉ LR***  REPORT ERROR  RUN subprotocol SIGMA.simulator with input (***x, e***) and obtains a transcript **(*a, e, z*)**.  The output of *M* is computed as follows:   1. SAMPLE at random ***z ∈ G***. (e.g in DLOG it is ***z ∈* Zp*\****) 2. COMPUTE ***a*** as a function of***( e ,z)***. (e.g in DLOG it is   ***a* = *gzh− e* mod *p***)  SEND ***a*** to R |
| Decommit phase | |
| Step 3: | SEND **(*e, z*)** to R |
| Step 4: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Commit phase | |
| Step 1: | RUN the generator *G* on input **1*t***to obtain **(*x,w*) *∈ R***  SEND ***x***to *C*. |
| Step 2: | WAIT for message ***a*** from C |
| Decommit phase | |
| Step 3: | WAIT for **(*e, z*)** from C |
| Step 4: | IF **(*a, e, z*)** is an accepting transcript in *π* with respect to input ***x***  OUTPUT ACC and trap ***w***  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| Protocol Name: | Trapdoor (equivocal) commitment schemes - Commitment from Schnorr |
| Protocol Reference: | COMMIT\_TRAP\_FROM\_SIGMA\_DLOG |
| Protocol Type: | Trapdoor Commitment |
| Protocol Description: | The committer C commits to a value to the receiver V while keeping it hidden, with the ability to reveal the committed value later. The underlying sigma protocol here is Schnorr's |
| References: | Protocol 6.6.2, page 174 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * C’s private input: a value ***e ∈ {*0*,* 1*}t*** |
| Parties’ Outputs: | * C: nothing * R: ACC or REJ and **trap *w*** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Commit phase | |
| Step 1 (Both): | C: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR  R: RUN the generator *G* on input **1*t***to obtain **(*gw,w*) *∈ R***  SEND ***gw***to *C*. |
| Step 2 (C): | IF ***gw ∉ LR***  REPORT ERROR  RUN subprotocol SIGMA\_DLOG.simulator with input (***gw, e***) and obtains a transcript **(*a, e, z*)**.  The output of *M* is computed as follows:   1. SAMPLE at random ***z ∈* Zp*\**** 2. COMPUTE ***a* = *gzh− e* mod *p***   SEND ***a*** to R |
| Decommit phase | |
| Step 3 (C): | SEND **(*e, z*)** to R |
| Step 4 (Both): | R: IF **(*a, e, z*)** is an accepting transcript in SIGMA\_DLOGwith respect to input ***gw***  OUTPUT ACC and trap ***w***  ELSE  OUTPUT REJ  C: output nothing |

|  |  |
| --- | --- |
| Committer (C) Specification | |
| Commit phase | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2: | WAIT for message ***gw*** from C  IF ***gw ∉ LR***  REPORT ERROR  RUN subprotocol SIGMA\_DLOG.simulator with input (***gw, e***) and obtains a transcript **(*a, e, z*)**.  The output of *M* is computed as follows:   1. SAMPLE at random ***z ∈* Zp*\**** 2. COMPUTE ***a* = *gzh− e* mod *p***   SEND ***a*** to R |
| Decommit phase | |
| Step 3: | SEND **(*e, z*)** to R |
| Step 4: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Commit phase | |
| Step 1: | RUN the generator *G* on input **1*t***to obtain **(*gw,w*) *∈ R***  SEND ***gw***to *C*. |
| Step 2: | WAIT for message ***a*** from C |
| Decommit phase | |
| Step 3: | WAIT for **(*e, z*)** from C |
| Step 4: | IF **(*a, e, z*)** is an accepting transcript in SIGMA\_DLOGwith respect to input ***gw***  OUTPUT ACC and trap ***w***  ELSE  OUTPUT REJ |

# Oblivious Transfers

|  |  |
| --- | --- |
| Protocol Name: | Naor-Pinkas |
| Protocol Reference: | OT\_USING\_DH |
| Protocol Type: | Oblivious Transfer Protocol |
| Protocol Description: | The sender S sends some information to the receiver R, but remains oblivious as to what is received. |
| References: | Protocol 7.2.1 page 179 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * S’s private input: ***x*0*, x*1** of the same (arbitrary) length (The calling protocol has to pad if they may not be the same length) * R's private input: a bit ***σ ∈ {*0*,* 1*}***. |
| Parties’ Outputs: | * S: nothing * R: ***xσ* = *cσ XOR KDF*(*kσ*).** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | S: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR  R: SAMPLE at random ***α, β, γ ∈ {*1*, . . . , q}***  COMPUTE **¯*a***as follows:   1. If ***σ* = 0** then **¯*a* = (*gα, gβ, gαβ, gγ*)** 2. If ***σ* = 1** then **¯*a* = (*gα, gβ, gγ, gαβ*)**   SEND **¯*a***to S. |
| Step 2 (S): | DENOTE the tuple **¯*a***received by Sby **(*x, y, z*0*, z*1)**  IF **NOT**   * ***z*0 *= z*1** * ***x, y, z*0*, z*1**∈ ***G***   REPORT ERROR  SAMPLE at random ***u0,u1,v0,v1*** ***∈ {*1*, . . . , q}***  COMPUTE 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**   COMPUTE   * ***X0 XOR KDF(k0)*** * ***X1 XOR KDF(k1)***   SEND the pairs **(*w*0*, c*0)** and **(*w*1*, c*1)** to R |
| Step 3 (R): | R: IF NOT   * ***w*0*, w*1**∈ ***G*** * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***xσ* = *cσ XOR KDF*(*kσ*).**  S: output nothing |
|  |  |

|  |  |
| --- | --- |
| Sender (S) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2: | WAIT for message **¯*a*** from R |
| Step 3: | DENOTE the tuple **¯*a***received by Sby **(*x, y, z*0*, z*1)**  IF **NOT**   * ***z*0 *= z*1** * ***x, y, z*0*, z*1**∈ ***G***   REPORT ERROR  SAMPLE at random ***u0,u1,v0,v1*** ***∈ {*1*, . . . , q}***  COMPUTE 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**   COMPUTE   * ***X0 XOR KDF(k0)*** * ***X1 XOR KDF(k1)***   SEND the pairs **(*w*0*, c*0)** and **(*w*1*, c*1)** to R |
| Step 4: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Step 1: | SAMPLE at random ***α, β, γ ∈ {*1*, . . . , q}***  COMPUTE **¯*a***as follows:   1. If ***σ* = 0** then **¯*a* = (*gα, gβ, gαβ, gγ*)** 2. If ***σ* = 1** then **¯*a* = (*gα, gβ, gγ, gαβ*)**   SEND **¯*a***to S. |
| Step 2: | WAIT for message pairs **(*w*0*, c*0)** and **(*w*1*, c*1)** from S |
| Step 3: | IF NOT   * ***w*0*, w*1**∈ ***G*** * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***xσ* = *cσ XOR KDF*(*kσ*)** |

|  |  |
| --- | --- |
| Protocol Name: | Oblivious Transfer with one-sided simulation |
| Protocol Reference: | OT\_ONE\_SIDED\_USING\_DH |
| Protocol Type: | Oblivious Transfer Protocol |
| Protocol Description: | The sender S sends some information to the receiver R, but remains oblivious as to what is received. |
| References: | Protocol 7.3 page 185 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * S’s private input: ***x*0*, x*1** of the same (arbitrary) length (The calling protocol has to pad if they may not be the same length) * R's private input: a bit ***σ ∈ {*0*,* 1*}***. |
| Parties’ Outputs: | * S: nothing * R:  ***xσ* = *cσ XOR KDF*(*kσ*)** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | S: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR  R: SAMPLE at random ***α, β, γ ∈ {*1*, . . . , q}***  COMPUTE **¯*a***as follows:   1. If ***σ* = 0** then **¯*a* = (*gα, gβ, gαβ, gγ*)** 2. If ***σ* = 1** then **¯*a* = (*gα, gβ, gγ, gαβ*)**   SEND **¯*a***to S. |
| Step 2 (R): | RUN subprotocol ZERO\_KNOWLEDGE of α |
| Step 3 (S): | IF proof of knowledge does not work  REPORT ERROR  ELSE  DENOTE the tuple **¯*a***received by Sby **(*x, y, z*0*, z*1)**  IF **NOT**   * ***z*0 *= z*1** * ***x, y, z*0*, z*1**∈ ***G***   REPORT ERROR  SAMPLE at random ***u0,u1,v0,v1*** ***∈ {*1*, . . . , q}***  COMPUTE 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**   COMPUTE   * ***X0 XOR KDF(k0)*** * ***X1 XOR KDF(k1)***   SEND the pairs **(*w*0*, c*0)** and **(*w*1*, c*1)** to R |
| Step 4 (R): | R: IF NOT   * ***w*0*, w*1**∈ ***G*** * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***xσ* = *cσ XOR KDF*(*kσ*)**.  S: output nothing |
|  |  |

|  |  |
| --- | --- |
| Sender (S) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2: | WAIT for message **¯*a*** from R |
| Step 3: | IF proof of knowledge does not work  REPORT ERROR  ELSE  DENOTE the tuple **¯*a***received by Sby **(*x, y, z*0*, z*1)**  IF **NOT**   * ***z*0 *= z*1** * ***x, y, z*0*, z*1**∈ ***G***   REPORT ERROR  SAMPLE at random ***u0,u1,v0,v1*** ***∈ {*1*, . . . , q}***  COMPUTE 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**   COMPUTE   * ***X0 XOR KDF(k0)*** * ***X1 XOR KDF(k1)***   SEND the pairs **(*w*0*, c*0)** and **(*w*1*, c*1)** to R |
| Step 4: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Step 1: | SAMPLE at random ***α, β, γ ∈ {*1*, . . . , q}***  COMPUTE **¯*a***as follows:   1. If ***σ* = 0** then **¯*a* = (*gα, gβ, gαβ, gγ*)** 2. If ***σ* = 1** then **¯*a* = (*gα, gβ, gγ, gαβ*)**   SEND **¯*a***to S. |
| Step 2: | WAIT for message pairs **(*w*0*, c*0)** and **(*w*1*, c*1)** from S |
| Step 3: | IF NOT   * ***w*0*, w*1**∈ ***G*** * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***xσ* = *cσ XOR KDF*(*kσ*)** |

|  |  |
| --- | --- |
| Protocol Name: | Oblivious Transfer with full simulation |
| Protocol Reference: | OT\_FULL\_USING\_DH |
| Protocol Type: | Oblivious Transfer Protocol |
| Protocol Description: | The sender S sends some information to the receiver R, but remains oblivious as to what is received. |
| References: | Protocol 7.4.1 page 190 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * S’s private input: ***x*0*, x*1** of the same (arbitrary) length (The calling protocol has to pad if they may not be the same length) * R's private input: a bit ***σ ∈ {*0*,* 1*}***. |
| Parties’ Outputs: | * S: nothing * R**:  *zσ XORKDF(wσασ)*** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | S: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR  R: SAMPLE at random ***α0, α1, r ∈ {*1*, . . . , q}***  COMPUTE the following:   1. ***h*0 = *gα*0** 2. ***h*1 = *gα*1** 3. ***a* = *gr***   SEND **(*h*0*, h*1*, a, b*0*, b*1)** to S. |
| Step 2 (Both): | S: IF **NOT** ***h*0*, h*1*, a, b*0*, b*1 *∈* G**  REPORT ERROR  R: SET ***h* = *h*0*/h*1** and ***b* = *b*0*/b*1**  RUN subprotocol ZERO\_KNOWLEDGE that (G*, q, g, h, a, b*) is a  DH tuple  (Formally, Rproves the relation:  ***R*DH = { ((G*, q, g, h, a, b*)*, r*) *| a* = *gr* & *b* = *hr* }**) |
| Step 3 (S): | IF proof of knowledge does not work  REPORT ERROR  ELSE  SAMPLE ***u*0*, v*0*, u*1*, v*1 *∈ {*1*, . . . , q}***  COMPUTE   1. ***e*0 = (*w*0*, z*0)** where   ***w*0 = *au*0 *· gv*0** and ***z*0 = KDF(*b0u*0  *· h0v*0 *) XOR x*0**   1. ***e1 = (w1, z1)*** where   ***w1 = au1 · gv1*** *and* ***z1 = KDF(( b1 /g )u1 · h1v1) XOR x1***  SEND ***(e0,e1)*** to R |
| Step 4 (Both): | R: IF NOT ***w*0*, w*1**∈ ***G***  REPORT ERROR  ELSE  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***zσ XORKDF(wσασ)***  S: output nothing |

|  |  |
| --- | --- |
| Sender (S) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2: | WAIT for message **(*h*0*, h*1*, a, b*0*, b*1)** from R  IF **NOT** ***h*0*, h*1*, a, b*0*, b*1 *∈* G**  REPORT ERROR |
| Step 3: | IF proof of knowledge does not work  REPORT ERROR  ELSE  SAMPLE ***u*0*, v*0*, u*1*, v*1 *∈ {*1*, . . . , q}***  COMPUTE   1. ***e*0 = (*w*0*, z*0)** where   ***w*0 = *au*0 *· gv*0** and ***z*0 = KDF(*b0u*0  *· h0v*0 *) XOR x*0**   1. ***e1 = (w1, z1)*** where   ***w1 = au1 · gv1*** *and* ***z1 = KDF(( b1 /g )u1 · h1v1) XOR x1***  SEND ***(e0,e1)*** to R |
| Step 4: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Step 1: | SAMPLE at random ***α0, α1, r ∈ {*1*, . . . , q}***  COMPUTE the following:   1. ***h*0 = *gα*0** 2. ***h*1 = *gα*1** 3. ***a* = *gr*** 4. ***b*0 = *h0r* *· gσ*** 5. ***b*1 = *h1r* *· gσ***   SEND **(*h*0*, h*1*, a, b*0*, b*1)** to S. |
| Step 2: | SET ***h* = *h*0*/h*1** and ***b* = *b*0*/b*1**  RUN subprotocol ZERO\_KNOWLEDGE that (G*, q, g, h, a, b*) is a  DH tuple  (Formally, Rproves the relation:  ***R*DH = { ((G*, q, g, h, a, b*)*, r*) *| a* = *gr* & *b* = *hr* }**) |
| Step 3: | WAIT for message ***(e0,e1)*** From S |
| Step 4: | IF NOT ***w*0*, w*1**∈ ***G***  REPORT ERROR  ELSE  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***zσ XORKDF(wσασ)*** |

|  |  |
| --- | --- |
| Protocol Name: | PVW\_plain |
| Protocol Reference: | OT\_FULL\_USING\_DH\_OR\_N\_RESIDUOSITY\_PVW |
| Protocol Type: | Oblivious Transfer Protocol |
| Protocol Description: | The sender S sends some information to the receiver R, but remains oblivious as to what is received. |
| References: | Protocol 7.5.1 page 201 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g0***) where (***G,q, g0*)** is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * S’s private input: ***x*0*, x*1** of the same (arbitrary) length (The calling protocol has to pad if they may not be the same length) * R's private input: a bit ***σ ∈ {*0*,* 1*}***. |
| Parties’ Outputs: | * S: nothing * R**:  *zσ XORKDF(wσασ)*** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | S: IF **NOT**   * ***q*** is prime * ***g0***  is of order ***q***   REPORT ERROR  R: SAMPLE at random ***α0, y ∈ {*1*, . . . , q}***  SET ***α1* = *α0* + 1**  COMPUTE the following:   1. ***g*1 = *g0y*** 2. ***h*0 = *g0α*0** 3. ***h*1= *g1α*1**   SEND **(*g*1*, h*0*, h*1)** to S. |
| Step 2 (R): | RUN subprotocol ZERO\_KNOWLEDGE\_FOR\_SIGMA\_DH\_AND\_COMMIT for **w= *α0*** and **x =( *g*0*, g*1*, h*0*, h*1/ *g*1)** that ***x*** is DH tuple  (Formally, Rproves the relation:  ***R*DH = { ((G*, q, g*0*, g*1*, h*0*, h*1/ *g*1)*, α0*) *| h*0= *g*0 *α0* & *h*1/ *g*1= *g*1 *α0* }**) |
| Step 3 (Both): | S: IF proof of knowledge does not work  REPORT ERROR  R: SAMPLE at random ***r ∈ {*1*, . . . , q}***  COMPUTE   1. ***g* = (*gσ*)*r*** 2. ***h* = (*hσ*)*r***,   SEND **(*g, h*)** to S |
| Step 4 (S): | DEFINE ***RAND*(*w, x, y, z*) = (*u, v*)**, where ***u* = (*w*)*s·*(*y*)*t***and  ***v* = (*x*)*s·*(*z*)*t***, and ***s, t ∈ {*1*, . . . , q}*** are SAMPLED at random.  COMPUTE   * 1. **(*u*0*, v*0) = *RAND*(*g*0*, g, h*0*, h*)**   2. **(*u*1*, v*1) = *RAND*(*g*1*, g, h*1*, h*)**   SET   * 1. ***w*0 = *v*0*·x*0**   2. ***w*1 = *v*1*·x*1**   SEND **(*u*0*,w*0)** and **(*u*1*,w*1)** |
| Step 4 (Both): | R: OUTPUT ***xσ* = *wσ/*(*uσ*)*r***  S: output nothing |

|  |  |
| --- | --- |
| Sender (S) Specification | |
| Step 1: | : IF **NOT**   * ***q*** is prime * ***g0***  is of order ***q***   REPORT ERROR |
| Step 2: | WAIT for message **(*g*1*, h*0*, h*1)** form S |
| Step 2: | IF proof of knowledge does not work  REPORT ERROR |
| Step 3: | DEFINE ***RAND*(*w, x, y, z*) = (*u, v*)**, where ***u* = (*w*)*s·*(*y*)*t***and  ***v* = (*x*)*s·*(*z*)*t***, and ***s, t ∈ {*1*, . . . , q}*** are SAMPLED at random.  COMPUTE   * 1. **(*u*0*, v*0) = *RAND*(*g*0*, g, h*0*, h*)**   2. **(*u*1*, v*1) = *RAND*(*g*1*, g, h*1*, h*)**   SET   * 1. ***w*0 = *v*0*·x*0**   2. ***w*1 = *v*1*·x*1**   SEND **(*u*0*,w*0)** and **(*u*1*,w*1)** |
| Step 4: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Step 1: | SAMPLE at random ***α0, y ∈ {*1*, . . . , q}***  SET ***α1* = *α0* + 1**  COMPUTE the following:   1. ***g*1 = *g0y*** 2. ***h*0 = *g0α*0** 3. ***h*1= *g1α*1**   SEND **(*g*1*, h*0*, h*1)** to S. |
| Step 2: | RUN subprotocol ZERO\_KNOWLEDGE\_FOR\_SIGMA\_DH\_AND\_COMMIT for **w= *α0*** and **x =( *g*0*, g*1*, h*0*, h*1/ *g*1)** that ***x*** is DH tuple  (Formally, Rproves the relation:  ***R*DH = { ((G*, q, g*0*, g*1*, h*0*, h*1/ *g*1)*, α0*) *| h*0= *g*0 *α0* & *h*1/ *g*1= *g*1 *α0* }**) |
| Step 3: | SAMPLE at random ***r ∈ {*1*, . . . , q}***  COMPUTE   1. ***g* = (*gσ*)*r*** 2. ***h* = (*hσ*)*r***,   SEND **(*g, h*)** to S |
| Step 4: | WAIT for message **(*u*0*,w*0)** from S |
| Step 5: | OUTPUT ***xσ* = *wσ/*(*uσ*)*r*** |
|  |  |

# Batch Oblivious Transfer

|  |  |
| --- | --- |
| Protocol Name: | Naor-Pinkas Batch Oblivious Transfer |
| Protocol Reference: | BATCH\_OT\_USING\_DH |
| Protocol Type: | Batch Oblivious Transfer Protocol |
| Protocol Description: |  |
| References: |  |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q, g*)** is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * S’s private input: a list of m pairs of strings **(*x01 , x11* ), . . . , (*x0m, x1m*)** of the same (arbitrary) length (The calling protocol has to pad if they may not be the same length) * R's private input: ***m*** bits string **(*σ1, . . . , σm*)** |
| Parties’ Outputs: | * S: nothing * R**:  *xσ i* = *cσ i XOR KDF*(*kσi i*)** for every ***i=1,…,m*** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | S: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR  R: SAMPLE at random ***α, βi,… , βm , γi,…, γm******∈ {*1*, . . . , q}***  COMPUTE **¯*a***as follows:   1. If ***σi* = 0** then **¯*ai* = (*gβi, gαβi, gγi*)** 2. If ***σi* = 1** then **¯*ai* = (*gβi, gγi, gαβi*)**   SEND ***gα*** and **¯*a1,...,* ¯*am***to S. |
| Step 2 (S): | DENOTE   1. the tuple **¯*ai***received by Sby **( *yi, z*0*i, z*1*i*)** 2. ***x* = *gα***   IF **NOT**   * ***z*0 *i = z*1 *i*** * ***x, yi, z*0 *i, z*1 *i*** ∈ ***G***   REPORT ERROR  ELSE  SAMPLE at random ***u*0*i, u*1 *i, v*0 *i, v*1 *i*** ***∈ {*1*, . . . , q}*** for every ***i=1,…,m***  COMPUTE 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* *· yv*0 *i*** * ***w*1 *i* = *xu*1 *i* *· gv*1 *i* *k*1 *i* = (*z*1 *i*)*u*1 *i* *· yv*1 *i***   COMPUTE   * ***X0i XOR KDF(k0i)*** * ***X1i XOR KDF(k1i)***   SEND ***m*** pairs **(*w*0 *i, c*0 *i*)** and **(*w*1 *i, c*1 *i*)** to R |
| Step 3 (Both): | R: IF NOT   * ***w*0 *i, w*1 *i*** ∈ ***G*** * ***c*0 *i, c*1 *i*** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσi i* = (*wσi i*)*βi***  OUTPUT ***xσ i* = *cσ i XOR KDF*(*kσi i*)** for every ***i***  S: output nothing |
|  |  |

|  |  |
| --- | --- |
| Sender (S) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2: | WAIT for message ***gα*** and **¯*a1,...,* ¯*am*** from R |
| Step 3: | DENOTE   1. the tuple **¯*ai***received by Sby **( *yi, z*0*i, z*1*i*)** 2. ***x* = *gα***   IF **NOT**   * ***z*0 *i = z*1 *i*** * ***x, yi, z*0 *i, z*1 *i*** ∈ ***G***   REPORT ERROR  ELSE  SAMPLE at random ***u*0*i, u*1 *i, v*0 *i, v*1 *i*** ***∈ {*1*, . . . , q}*** for every ***i=1,…,m***  COMPUTE 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* *· yv*0 *i*** * ***w*1 *i* = *xu*1 *i* *· gv*1 *i* *k*1 *i* = (*z*1 *i*)*u*1 *i* *· yv*1 *i***   COMPUTE   * ***X0i XOR KDF(k0i)*** * ***X1i XOR KDF(k1i)***   SEND ***m*** pairs **(*w*0 *i, c*0 *i*)** and **(*w*1 *i, c*1 *i*)** to R |
| Step 4: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Step 1: | SAMPLE at random ***α, βi,… , βm , γi,…, γm******∈ {*1*, . . . , q}***  COMPUTE **¯*a***as follows:   1. If ***σi* = 0** then **¯*ai* = (*gβi, gαβi, gγi*)** 2. If ***σi* = 1** then **¯*ai* = (*gβi, gγi, gαβi*)**   SEND ***gα*** and **¯*a1,...,* ¯*am***to S. |
| Step 2: | WAIT for ***m*** pairs **(*w*0 *i, c*0 *i*)** and **(*w*1 *i, c*1 *i*)** from S |
| Step 3: | IF NOT   * ***w*0 *i, w*1 *i*** ∈ ***G*** * ***c*0 *i, c*1 *i*** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσi i* = (*wσi i*)*βi***  OUTPUT ***xσ i* = *cσ i XOR KDF*(*kσi i*)** for every ***i*** |

|  |  |
| --- | --- |
| Protocol Name: | Batch Oblivious Transfer Full Simulation |
| Protocol Reference: | BATCH\_OT\_FULL |
| Protocol Type: | Batch Oblivious Transfer Protocol |
| Protocol Description: | A series of oblivious transfers |
| References: | Protocol 7.4.3 page 198 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q, g*)** is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * S’s private input: a list of m pairs of strings **(*x01 , x11* ), . . . , (*x0m, x1m*)** of the same (arbitrary) length (The calling protocol has to pad if they may not be the same length) * R's private input: ***m*** bits string **(*σ1, . . . , σm*)** |
| Parties’ Outputs: | * S: nothing * R**: *zσj / KDF(wσjασj)***for every **j *=1,…,m*** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | S: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR  R: SAMPLE at random ***α0, α1, r ∈ {*1*, . . . , q}***  COMPUTE the following:   1. ***h*0 = *gα*0** 2. ***h*1 = *gα*1**   SEND **(*h*0*, h*1)** to S. ????? Not in the book. |
| Step 2 (Both): | S: IF **NOT *h*0*, h*1,**  REPORT ERROR  R: RUN subprotocol ZERO\_KNOWLEDGE\_FOR\_SIGMA\_DLOG\_AND\_COMMIT with input ***w= α0*** and ***x = h*0 *= gα0*** (the descrete log of ***h*0**) |
| Step 3 (Both): | S: IF proof of knowledge does not work  REPORT ERROR  R: SAMPLE at random ***r1,…, rm******∈ {*1*, . . . , q}***  COMPUTE   1. ***aj = grj*** 2. ***b0j = h0rj  · gσj*** 3. ***b1j  = h1rj · gσj***   SEND ***aj***, ***b0j***, ***b1j*** for every **j** to S |
| Step 4 (S): | IF **NOT *aj***, ***b0j***, ***b1j∈*** G  REPORT ERROR  ELSE  SAMPLE at random ***ρ1, . . . , ρm ∈ {1, . . . , q}***  SEND ***ρ1, . . . , ρm*** to R |
| Step 5 (Both): | S and R both:  COMPUTE |
| Step 6 (Both): | S: IF **NOT**  ***a, b*0*, b*1 *∈* G**  REPORT ERROR  R: RUN subprotocol ZERO\_KNOWLEDGE that (G*, q, g, h, a, b*) is a  DH tuple  (Formally, Rproves the relation:  ***RDH = { ((G, q, g, h0/ h1, a, b), ) | a =***  ***& b = }*** |
| Step 7 (S): | IF proof of knowledge does not work  REPORT ERROR  ELSE  SAMPLE ***u*0j*, v*0j*, u*1j*, v*1j** ***∈ {*1*, . . . , q}***  COMPUTE for every **j** (superscript **j** omitted)   1. ***e*0 = (*w*0*, z*0)** where   ***w*0 = *au*0 *· gv*0** and ***z*0 = KDF(*b0u*0  *· h0v*0 *) XOR x*0**   1. ***e1 = (w1, z1)*** where   ***w1 = au1 · gv1*** *and* ***z1 = KDF(( b1 /g )u1 · h1v1) XOR x1***  SEND ***(e0j,e1j)*** for every ***j*** to R |
| Step 8 (Both): | R: IF NOT ***w*0*j, w*1j**∈ ***G*** for every ***j***  REPORT ERROR  ELSE  COMPUTE ***kσ* = (*wσ*)*β*** for every ***j*** (***j*** omitted)  OUTPUT ***zσ XORKDF(wσασ)*** for every ***j*** (***j*** omitted)  S: output nothing |

|  |  |
| --- | --- |
| Sender (S) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2: | WAIT for message **(*h*0*, h*1)** from R  IF **NOT** ***h*0*, h*1 *∈* G**  REPORT ERROR |
| Step 3: | IF proof of knowledge does not work  REPORT ERROR |
| Step 4: | WAIT for messages ***aj***, ***b0j***, ***b1j*** from R  IF **NOT *aj***, ***b0j***, ***b1j∈*** G  REPORT ERROR |
| Step 5: | WAIT for messages ***ρ1, . . . , ρm*** from R  COMPUTE |
| Step 6: | IF **NOT**  ***a, b*0*, b*1 *∈* G**  REPORT ERROR |
| Step 7: | IF proof of knowledge does not work  REPORT ERROR  ELSE  SAMPLE ***u*0j*, v*0j*, u*1j*, v*1j** ***∈ {*1*, . . . , q}***  COMPUTE for every **j** (superscript **j** omitted)   1. ***e*0 = (*w*0*, z*0)** where   ***w*0 = *au*0 *· gv*0** and ***z*0 = KDF(*b0u*0  *· h0v*0 *) XOR x*0**   1. ***e1 = (w1, z1)*** where   ***w1 = au1 · gv1*** *and* ***z1 = KDF(( b1 /g )u1 · h1v1) XOR x1***  SEND ***(e0j,e1j)*** for every ***j*** to R |
| Step 8: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Step 1: | SAMPLE at random ***α0, α1, r ∈ {*1*, . . . , q}***  COMPUTE the following:   1. ***h*0 = *gα*0** 2. ***h*1 = *gα*1**   SEND **(*h*0*, h*1)** to S. ????? Not in the book. |
| Step 2: | RUN subprotocol ZERO\_KNOWLEDGE\_FOR\_SIGMA\_DLOG\_AND\_COMMIT with input ***w= α0*** and ***x = h*0 *= gα0*** (the descrete log of ***h*0**) |
| Step 3: | SAMPLE at random ***r1,…, rm******∈ {*1*, . . . , q}***  COMPUTE   1. ***aj = grj*** 2. ***b0j = h0rj  · gσj*** 3. ***b1j  = h1rj · gσj***   SEND ***aj***, ***b0j***, ***b1j*** for every **j** to S |
| Step 4: | WAIT for message ***ρ1, . . . , ρm*** From S |
| Step 5: | COMPUTE |
| Step 6: | RUN subprotocol ZERO\_KNOWLEDGE that (G*, q, g, h, a, b*) is a  DH tuple  (Formally, Rproves the relation:  ***RDH = { ((G, q, g, h0/ h1, a, b), ) | a =***  ***& b = }*** |
| Step 7: | WAIT for message ***(e0j,e1j)*** for every ***j*** From S |
| Step 8: | IF NOT ***w*0*j, w*1j**∈ ***G*** for every ***j***  REPORT ERROR  ELSE  COMPUTE ***kσ* = (*wσ*)*β*** for every ***j*** (***j*** omitted)  OUTPUT ***zσ XORKDF(wσασ)*** for every ***j*** (***j*** omitted) |

|  |  |
| --- | --- |
| Protocol Name: | PVW\_plain |
| Protocol Reference: | BATCH\_OT\_FULL\_USING\_DH\_OR\_N\_RESIDUOSITY\_PVW |
| Protocol Type: | Batch Oblivious Transfer Protocol |
| Protocol Description: | A series of oblivious transfers |
| References: | Protocol 7.5.2 page 202 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g0***) where (***G,q, g0*)** is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * S’s private input: a list of m pairs of strings **(*x01 , x11* ), . . . , (*x0m, x1m*)** of the same (arbitrary) length (The calling protocol has to pad if they may not be the same length) * R's private input: ***m*** bits string **(*σ1, . . . , σm*)** |
| Parties’ Outputs: | * S: nothing * R**:  *xσJ* = *wσJ/*(*uσJ*)*rJ*** for every **j** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | S: IF **NOT**   * ***q*** is prime * ***g0***  is of order ***q***   REPORT ERROR  R: SAMPLE at random ***α0, y ∈ {*1*, . . . , q}***  SET ***α1* = *α0* + 1**  COMPUTE the following:   1. ***g*1 = *g0y*** 2. ***h*0 = *g0α*0** 3. ***h*1= *g1α*1**   SEND **(*g*1*, h*0*, h*1)** to S. |
| Step 2 (R): | RUN subprotocol ZERO\_KNOWLEDGE\_FOR\_SIGMA\_DH\_AND\_COMMIT with input **w= *α0*** and **x =( *g*0*, g*1*, h*0*, h*1/ *g*1)** that ***x*** is DH tuple  (Formally, Rproves the relation:  ***R*DH = { ((G*, q, g*0*, g*1*, h*0*, h*1/ *g*1)*, α0*) *| h*0= *g*0 *α0* & *h*1/ *g*1= *g*1 *α0* }**) |
| Step 3 (Both): | S: IF proof of knowledge does not work  REPORT ERROR  R: For every **j** (superscript j omitted)  SAMPLE at random ***r ∈ {*1*, . . . , q}***  COMPUTE   1. ***g* = (*gσ*)*r*** 2. ***h* = (*hσ*)*r***,   SEND ***m*** pairs of **(*g, h*)** to S |
| Step 4 (S): | DEFINE ***RAND*(*w, x, y, z*) = (*u, v*)**, where ***u* = (*w*)*s·*(*y*)*t***and  ***v* = (*x*)*s·*(*z*)*t***, and ***s, t ∈ {*1*, . . . , q}*** are SAMPLED at random.  For every **j** (superscript **j** omitted)  COMPUTE   * 1. **(*u*0*, v*0) = *RAND*(*g*0*, g, h*0*, h*)**   2. **(*u*1*, v*1) = *RAND*(*g*1*, g, h*1*, h*)**   SET   * 1. ***w*0 = *v*0*·x*0**   2. ***w*1 = *v*1*·x*1**   SEND for every **j (*u*0*,w*0)** and **(*u*1*,w*1)** to R (superscript **j** omitted) |
| Step 5 (Both): | R: OUTPUT ***xσJ* = *wσJ/*(*uσJ*)*rJ*** for every **j**  S: output nothing |

|  |  |
| --- | --- |
| Sender (S) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g0***  is of order ***q***   REPORT ERROR |
| Step 2: | WAIT for message **(*g*1*, h*0*, h*1)** form S |
| Step 2: | IF proof of knowledge does not work  REPORT ERROR |
| Step 3: | WAIT for ***m*** pairs of **(*g, h*)**  DEFINE ***RAND*(*w, x, y, z*) = (*u, v*)**, where ***u* = (*w*)*s·*(*y*)*t***and  ***v* = (*x*)*s·*(*z*)*t***, and ***s, t ∈ {*1*, . . . , q}*** are SAMPLED at random.  For every **j** (superscript **j** omitted)  COMPUTE   * 1. **(*u*0*, v*0) = *RAND*(*g*0*, g, h*0*, h*)**   2. **(*u*1*, v*1) = *RAND*(*g*1*, g, h*1*, h*)**   SET   * 1. ***w*0 = *v*0*·x*0**   2. ***w*1 = *v*1*·x*1**   SEND for every **j (*u*0j*,w*0 j)** and **(*u*1 j*,w*1 j)** to R |
| Step 4: | output nothing |

|  |  |
| --- | --- |
| Receiver (R) Specification | |
| Step 1: | SAMPLE at random ***α0, y ∈ {*1*, . . . , q}***  SET ***α1* = *α0* + 1**  COMPUTE the following:   1. ***g*1 = *g0y*** 2. ***h*0 = *g0α*0** 3. ***h*1= *g1α*1**   SEND **(*g*1*, h*0*, h*1)** to S. |
| Step 2: | RUN subprotocol ZERO\_KNOWLEDGE\_FOR\_SIGMA\_DH\_AND\_COMMIT for **w= *α0*** and **x =( *g*0*, g*1*, h*0*, h*1/ *g*1)** that ***x*** is DH tuple  (Formally, Rproves the relation:  ***R*DH = { ((G*, q, g*0*, g*1*, h*0*, h*1/ *g*1)*, α0*) *| h*0= *g*0 *α0* & *h*1/ *g*1= *g*1 *α0* }**) |
| Step 3: | For every **j** (superscript j omitted)  SAMPLE at random ***r ∈ {*1*, . . . , q}***  COMPUTE   1. ***g* = (*gσ*)*r*** 2. ***h* = (*hσ*)*r***,   SEND ***m*** pairs of **(*g, h*)** to S |
| Step 4: | WAIT for **(*u*0j*,w*0 j)** and **(*u*1 j*,w*1 j)** for every ***j*** from S |
| Step 5: | OUTPUT ***xσ* = *wσ/*(*uσ*)*r*** |
|  |  |

# Coin Tossing

|  |  |
| --- | --- |
| Protocol Name: | Blum single-coin tossing using any commitment scheme |
| Protocol Reference: | COIN\_TOSSING\_BLUM |
| Protocol Type: | Coin tossing Protocol |
| Protocol Description: |  |
| References: |  |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Party P1 (P1) and Party P2 (P2) |
| Parties’ Inputs: | * Common input: a bit ***b*** |
| Parties’ Outputs: | * P1: ***b1*** XOR***b2*** * P2: ***b1*** XOR***b2*** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (both): | P1: SAMPLE a random bit ***b1∈{0,1}***  P2: SAMPLE a random bit ***b2∈{0,1}*** |
| Step 2 (P1): | RUN subprotocol COMMIT.commit on ***b1*** |
| Step 3 (P2): | SEND ***b2*** to P1 |
| Step 4 (P1): | RUN subprotocol COMMIT.decommit on ***b1*** |
| Step 5 (Both) | P1: OUTPUT ***b1*** XOR***b2***  P2: IF decommit did not work  REPORT ERROR  ELSE  OUTPUT ***b1*** XOR***b2*** |

|  |  |
| --- | --- |
| Party P1 (P1) Specification | |
| Step 1: | SAMPLE a random bit ***b1∈{0,1}*** |
| Step 2: | RUN subprotocol COMMIT.commit on ***b1*** |
| Step 3: | WAIT for message ***b2*** from P2 |
| Step 4: | RUN subprotocol COMMIT.decommit on ***b1*** |
| Step 5 | OUTPUT ***b1*** XOR***b2*** |

|  |  |
| --- | --- |
| Party P2 (P2) Specification | |
| Step 1: | SAMPLE a random bit ***b1∈{0,1}*** |
| Step 3: | WAIT for COMMIT.commit on ***b1***  SEND ***b2*** to P1 |
| Step 4: | RUN subprotocol COMMIT.decommit on ***b1*** |
| Step 5 | IF decommit did not work  REPORT ERROR  ELSE  OUTPUT ***b1*** XOR***b2*** |

|  |  |
| --- | --- |
| Protocol Name: | Coin tossing using Pedersen commitments and DLOG-ZK |
| Protocol Reference: | COIN\_TOSSING\_BLUM |
| Protocol Type: | Coin tossing Protocol |
| Protocol Description: |  |
| References: | [Lindell01] |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Party P1 (P1) and Party P2 (P2) |
| Parties’ Inputs: | * Common input: Common input: : (***G,q,g***) where (***G,q,g***) is a DLOG description and a parameter ***t*** such that ***2t* < *q*** |
| Parties’ Outputs: | * P1: ***KDF(rs)*** * P2: ***KDF(rs)*** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (both): | P1: SAMPLE a random bit ***r∈G***  P2: SAMPLE a random bit ***s∈G*** |
| Step 2 (P1): | RUN subprotocol COMMIT.commit on ***r*** |
| Step 3 (P1): | RUN subprotocol ZERO\_KNOWLEDGE\_FOR\_ SIGMA\_FOR\_PEDERSEN |
| Step 4 (P2): | IF proof of knowledge does not work  REPORT ERROR  ELSE  SEND ***s*** to P1 |
| Step 5 (P1): | SEND ***r*** to P2 |
| Step 4 (P1): | RUN subprotocol ZERO\_KNOWLEDGE\_FOR SIGMA\_COMMITTED\_VALUE\_PEDERSEN with input ***r*** |
| Step 5 (Both) | P1: OUTPUT ***KDF(rs)***  P2: IF proof of knowledge does not work  REPORT ERROR  ELSE  OUTPUT ***KDF(rs)*** |

|  |  |
| --- | --- |
| Party P1 (P1) Specification | |
| Step 1: | SAMPLE a random bit ***r∈G*** |
| Step 2: | RUN subprotocol COMMIT.commit on ***r*** |
| Step 3: | RUN subprotocol ZERO\_KNOWLEDGE\_FOR\_ SIGMA\_FOR\_PEDERSEN |
| Step 4: | WAIT for message ***s*** from P2  SEND ***r*** to P2 |
| Step 5: | RUN subprotocol ZERO\_KNOWLEDGE\_FOR SIGMA\_COMMITTED\_VALUE\_PEDERSEN with input ***r*** |
| Step 6: | OUTPUT ***KDF(rs)*** |

|  |  |
| --- | --- |
| Party P2 (P2) Specification | |
| Step 1: | SAMPLE a random bit ***s∈G*** |
| Step 2: | WAIT for COMMIT.commit on ***b1***  IF proof of knowledge does not work  REPORT ERROR  ELSE  SEND ***s*** to P1 |
| Step 4: | WAIT for message ***r*** from P1 |
| Step 5 | IF proof of knowledge does not work  REPORT ERROR  ELSE  OUTPUT ***KDF(rs)*** |

|  |  |
| --- | --- |
| Protocol Name: | Semi -simulatable coin-tossing |
| Protocol Reference: | COIN\_TOSSING\_SEMI\_SIMULATABLE |
| Protocol Type: | Coin tossing Protocol |
| Protocol Description: |  |
| References: |  |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Party P1 (P1) and Party P2 (P2) |
| Parties’ Inputs: | * Common input: |
| Parties’ Outputs: | * P1: ***r XOR s*** * P2: ***r XOR s*** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (both): | P1: SAMPLE a random bit ***r∈G***  P2: SAMPLE a random bit ***s∈G*** |
| Step 2 (P1): | RUN subprotocol COMMIT\_PERFECTLY\_HIDING.commit on ***r*** |
| Step 3 (P2): | RUN subprotocol COMMIT\_PERFECTLY\_BINDING.commit on ***s*** |
| Step 4 (P1): | RUN subprotocol COMMIT\_PERFECTLY\_HIDING.decommit on ***r*** |
| Step 5 (P2): | RUN subprotocol COMMIT\_PERFECTLY\_BINDING.decommit on ***s*** |
| Step 6 (Both) | P1: IF decommit did not work  REPORT ERROR  ELSE  OUTPUT ***r XOR s***  P2: IF decommit did not work  REPORT ERROR  ELSE  OUTPUT ***r XOR s*** |

|  |  |
| --- | --- |
| Party P1 (P1) Specification | |
| Step 1: | SAMPLE a random bit ***r∈G*** |
| Step 2: | RUN subprotocol COMMIT\_PERFECTLY\_HIDING.commit on ***r*** |
| Step 3: | WAIT for COMMIT.commit on ***s*** from P2  RUN subprotocol COMMIT\_PERFECTLY\_HIDING.decommit on ***r*** |
| Step 4: | IF decommit did not work  REPORT ERROR  ELSE  OUTPUT ***r XOR s*** |

|  |  |
| --- | --- |
| Party P2 (P2) Specification | |
| Step 1: | SAMPLE a random bit ***s∈G*** |
| Step 2: | WAIT for COMMIT.commit on ***r*** from P1  RUN subprotocol COMMIT\_PERFECTLY\_BINDING.commit on ***s*** |
| Step 3: | RUN subprotocol COMMIT\_PERFECTLY\_BINDING.decommit on ***s*** |
| Step 4: | IF decommit did not work  REPORT ERROR  ELSE  OUTPUT ***r XOR s*** |

# 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 Name: | Private Pseudorandom Function Evaluation |
| Protocol Reference: | PRIVATE\_PSEUDORANDOM\_FUNCTION\_EVALUATION |
| Protocol Type: | Pseudorandom Function Evaluation Protocol |
| Protocol Description: |  |
| References: | Protocol 7.6.3 page 206 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Party P1 (P1) and Party P2 (P2) |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q, g*)** is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * P1 ’s private input: *k = (ga0 , a1, . . . , am)* where   *a0, a1, . . . , am* ***∈ {*1*, . . . , q}***   * P2's private input: ***x = x1, . . . , xm*** of length *m* |
| Parties’ Outputs: | * P1: nothing * P2**:** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | P1: SAMPLE at random ***r1, . . . , rm ∈ {*1*, . . . , q}***  P2: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2 (Both): | RUN subprotocol BATCH\_OT\_USING\_DH (private – Naor-Pinkas)  Where  P1: The Sender (S) input:  list of ***m*** pairs of strings **(*y01 ,y11* ), . . . , (*y0m, y1m*) *y0i = ri*** and ***y1i= ri · ai*** (with multiplication in ***Zq\**** )  P2: The Receiver (R) input:  *m* bits string **(*σ1, . . . , σm*)** where ***σi = xi*** where ***x = x1, . . . , xm*** |
| Step 3 (Both): | IF the output of any of the oblivious transfers is ⊥  REPORT ERROR |
| Step 4 (P2): | P2:LET ***y1x1, . . . , ymxm***be the output of the BOT  FOR EVERY***i****:*  *IF* ***yixi* *∉ {*1*, . . . , q}***  SET*yixi* **= 1** |
| Step 5 (P1): | COMPUTE  SEND ˜g to P2 |
| Step 6 (Both): | P1: output nothing  P2: COMPUTE  OUTPUT ***y*** |

|  |  |
| --- | --- |
| Party P1 (P1) Specification | |
| Step 1: | SAMPLE at random ***r1, . . . , rm ∈ {*1*, . . . , q}*** |
| Step 2: | RUN subprotocol BATCH\_OT\_USING\_DH (private – Naor-Pinkas)  Where  As the Sender (S) input:  list of ***m*** pairs of strings **(*y01 ,y11* ), . . . , (*y0m, y1m*) *y0i = ri*** and ***y1i= ri · ai*** (with multiplication in ***Zq\**** ) |
| Step 3: | IF the output of any of the oblivious transfers is ⊥  REPORT ERROR |
| Step 4: | COMPUTE  SEND ˜g to P2 |
| Step 5: | output nothing |

|  |  |
| --- | --- |
| Party P2 (P2) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2: | RUN subprotocol BATCH\_OT\_USING\_DH (private – Naor-Pinkas)  Where  As the Receiver (R) input:  *m* bits string **(*σ1, . . . , σm*)** where ***σi = xi*** where ***x = x1, . . . , xm*** |
| Step 3: | IF the output of any of the oblivious transfers is ⊥  REPORT ERROR |
| Step 4: | P2:LET ***y1x1, . . . , ymxm***be the output of the BOT  FOR EVERY***i****:*  *IF* ***yixi* *∉ {*1*, . . . , q}***  SET*yixi* **= 1** |
| Step 5: | WAIT for message ˜g from P1 |
| Step 6: | COMPUTE  OUTPUT ***y*** |

|  |  |
| --- | --- |
| Protocol Name: | Fully-Simulatable Pseudorandom Function Evaluation |
| Protocol Reference: | FULL\_PSEUDORANDOM\_FUNCTION\_EVALUATION |
| Protocol Type: | Pseudorandom Function Evaluation Protocol |
| Protocol Description: |  |
| References: | Protocol 7.6.5 page 209 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Party P1 (P1) and Party P2 (P2) |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q, g*)** is a DLOG description and a parameter ***t*** such that ***2t* < *q*** * P1 ’s private input: *k = (ga0 , a1, . . . , am)* where   *a0, a1, . . . , am* ***∈ {*1*, . . . , q}***   * P2's private input: ***x = x1, . . . , xm*** of length *m* |
| Parties’ Outputs: | * P1: nothing * P2**:** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | P1: SAMPLE at random ***r1, . . . , rm ∈ {*1*, . . . , q}***  P2: IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2 (Both): | RUN subprotocol BATCH\_FULL\_OT (full simulation)  Where  P1: The Sender (S) input:  list of ***m*** pairs of strings **(*y01 ,y11* ), . . . , (*y0m, y1m*) *y0i = ri*** and ***y1i= ri · ai*** (with multiplication in ***Zq\**** )  P2: The Receiver (R) input:  *m* bits string **(*σ1, . . . , σm*)** where ***σi = xi*** where ***x = x1, . . . , xm*** |
| Step 3 (Both): | IF the output of any of the oblivious transfers is ⊥  REPORT ERROR |
| Step 4 (P2): | P2:LET ***y1x1, . . . , ymxm***be the output of the BOT  FOR EVERY***i****:*  *IF* ***yixi* *∉ {*1*, . . . , q}***  SET*yixi* **= 1** |
| Step 5 (P1): | COMPUTE  SEND **˜g** to P2 |
| Step 6 (Both): | P1: output nothing  P2: IF **NOT** **˜g** is of order ***q***  REPORT ERROR  ELSE  COMPUTE  OUTPUT ***y*** |

|  |  |
| --- | --- |
| Party P1 (P1) Specification | |
| Step 1: | SAMPLE at random ***r1, . . . , rm ∈ {*1*, . . . , q}*** |
| Step 2: | RUN subprotocol BATCH\_FULL\_OT (full simulation)  Where  As the Sender (S) input:  list of ***m*** pairs of strings **(*y01 ,y11* ), . . . , (*y0m, y1m*) *y0i = ri*** and ***y1i= ri · ai*** (with multiplication in ***Zq\**** ) |
| Step 3: | IF the output of any of the oblivious transfers is ⊥  REPORT ERROR |
| Step 4: | COMPUTE  SEND ˜g to P2 |
| Step 5: | output nothing |

|  |  |
| --- | --- |
| Party P2 (P2) Specification | |
| Step 1: | IF **NOT**   * ***q*** is prime * ***g*** is of order ***q***   REPORT ERROR |
| Step 2: | RUN subprotocol BATCH\_FULL\_OT (full simulation)  Where  As the Receiver (R) input:  *m* bits string **(*σ1, . . . , σm*)** where ***σi = xi*** where ***x = x1, . . . , xm*** |
| Step 3: | IF the output of any of the oblivious transfers is ⊥  REPORT ERROR |
| Step 4: | P2:LET ***y1x1, . . . , ymxm***be the output of the BOT  FOR EVERY***i****:*  *IF* ***yixi* *∉ {*1*, . . . , q}***  SET*yixi* **= 1** |
| Step 5: | WAIT for message ˜g from P1 |
| Step 6: | IF **NOT** **˜g** is of order ***q***  REPORT ERROR  ELSE  COMPUTE  OUTPUT ***y*** |