## *Σ* Protocol

**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.

## Oblivious Transfer

**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).

*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*).

## 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*.

## Zero knowledge

**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*.