.

SCAPI Pseudocode Specification

**Cryptography and Computer Security Research Group**

**Department of Computer Science**

**Bar-Ilan University**

**10/3/2010**

This document contains the exact specification of all the protocols that are implemented in SCAPI – the Secure Computation API (Application Programming Interface). Each protocol is referenced to an academic source for ease of use.

|  |  |  |  |
| --- | --- | --- | --- |
| Version No. | Date | Changed by | Scope of Change |
| 1.0 | 15/09/10 | Meital Levy | Initial preparation |
| 1.1 | 19/09/10 | Yehuda Lindell | Overall review, additions and corrections |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[2 Template and Document Explanation 6](#_Toc276586160)

[3 Sigma protocols 9](#_Toc276586161)

[3.1 Schnorr’s Sigma-Protocol for DLOG (SIGMA\_DLOG) 9](#_Toc276586162)

[3.2 Sigma-Protocol for Diffie-Hellman Tuples (SIGMA\_DH) 11](#_Toc276586163)

[3.3 Sigma Protocol for Pedersen Commitment Knowledge (SIGMA\_PEDERSEN) 13](#_Toc276586164)

[3.4 Sigma-Protocol that a Pedersen-Committed Value is x (SIGMA\_COMMITTED\_VALUE\_PEDERSEN) 15](#_Toc276586165)

[3.5 Sigma Protocol for El Gamal Commitment Knowledge (SIGMA\_ELGAMAL\_COMMIT) 16](#_Toc276586166)

[3.6 Sigma Protocol that an ElGamal-Committed Value is x (SIGMA\_COMMITTED\_VALUE\_ELGAMAL) 17](#_Toc276586167)

[3.7 Sigma Protocol of ElGamal Secret Key (SIGMA\_SK\_ELGAMAL) 18](#_Toc276586168)

[3.8 Sigma Protocol that ElGamal-Encrypted Value is x (SIGMA\_ENCRYPTED\_VALUE\_ELGAMAL) 19](#_Toc276586169)

[3.8.1 Version 1 – using knowledge of the secret key 19](#_Toc276586170)

[3.8.2 Version 2 – using knowledge of the randomness used to encrypt 20](#_Toc276586171)

[3.9 Sigma Protocol – AND of Multiple Statements (AND\_SIGMA) 21](#_Toc276586172)

[3.10 Sigma Protocol – OR of 2 Statements (OR\_2\_SIGMA) 23](#_Toc276586173)

[3.11 Sigma Protocol – OR of Multiple Statements (OR\_MANY\_SIGMA) 25](#_Toc276586174)

[3.12 Sigma Protocols for Paillier (Damgard-Jurik Version) 26](#_Toc276586175)

[4 Zero-knowledge 27](#_Toc276586176)

[4.1 Zero-Knowledge from any Sigma Protocol (ZK\_FROM\_SIGMA) 27](#_Toc276586177)

[4.2 Zero-Knowledge Proof of Knowledge from any Sigma Protocol (ZKPOK\_FROM\_SIGMA) 29](#_Toc276586178)

[4.3 ZKPOK from any Sigma-Protocol – ROM (Fiat-Shamir) (ZKPOK\_FS\_SIGMA) 31](#_Toc276586179)

[4.4 UC-Secure ZKPOK from any Sigma-Protocol (UCZKPOK\_FROM\_SIGMA) 33](#_Toc276586180)

[5 Commitment schemes 35](#_Toc276586181)

[5.1 Pedersen Commitment (COMMIT\_PEDERSEN) 35](#_Toc276586182)

[5.2 Pedersen-Hash Commitment (COMMIT\_HASH\_PEDERSEN) 37](#_Toc276586183)

[5.4 ElGamal Commitment(COMMIT\_ELGAMAL) 38](#_Toc276586184)

[5.5 ElGamal-Hash Commitment (COMMIT\_HASH\_ELGAMAL) 40](#_Toc276586185)

[5.7 Hash-Based Commitment (Basic) (COMMIT\_HASH\_BASIC) 41](#_Toc276586186)

[5.8 Hash-Based Statistically-Hiding Commitment (COMMIT\_HASH) 43](#_Toc276586187)

[5.10 Equivocal Commitments (COMMIT\_EQUIVOCAL) 45](#_Toc276586188)

[5.11 Extractable Commitments(COMMIT\_EXTRACT) 46](#_Toc276586189)

[5.12 Fully Trapdoor Commitments 48](#_Toc276586190)

[5.12.1 Fully Trapdoor using Sigma Protocols(COMMIT\_DOUBLE\_TRAPDOOR) 48](#_Toc276586191)

[5.12.2 Fully Trapdoor Hash (ROM) (COMMIT\_DOUBLE\_TRAPDOOR\_HASH\_ROM) 50](#_Toc276586192)

[5.12.3 Fully Trapdoor in the CRS Model (COMMIT\_DOUBLE\_TRAPDOOR\_DDH\_CRS) 50](#_Toc276586193)

[5.13 Additive Homomorphic Operation on Pedersen Commitments (PEDERSEN\_ADD) 51](#_Toc276586194)

[5.14 Multiplicative Homomorphic Operation on ElGamal Commitments (ELGAMAL\_MULT) 52](#_Toc276586195)

[5.15 UC-Secure Commitments 53](#_Toc276586196)

[5.15.1 UC-Secure Commitments with a Random Oracle (COMMIT\_UC\_ROM) 53](#_Toc276586197)

[5.15.2 UC-Secure Commitments from DDH (COMMIT\_UC\_DDH) 53](#_Toc276586198)

[5.16 Non-Malleable Commitments 54](#_Toc276586199)

[5.16.1 Non-Malleable Hash Commitments (heuristic) (COMMIT\_NON\_MALLEABLE\_HASH\_HEUR) 54](#_Toc276586200)

[5.16.2 Non-Malleable Commitments from Encryption (CRS) (COMMIT\_NON\_MALLEABLE\_CRS) 54](#_Toc276586201)

[6 Oblivious Transfers 56](#_Toc276586202)

[7 Batch Oblivious Transfer 68](#_Toc276586203)

[8 Coin Tossing 78](#_Toc276586204)

[9 Secure Pseudorandom Function Evaluation 84](#_Toc276586205)

[10 Oblivious Polynomial Evaluation (OPE) 88](#_Toc276586206)

[10.1 OPE from OT 88](#_Toc276586207)

[10.2 OPE from Homomorphic Encryption 88](#_Toc276586208)

[10.3 Fully Secure OPE (???) 88](#_Toc276586209)

[11 Key Exchange Protocols 89](#_Toc276586210)

[11.1 InitKey Protocol (KEY\_EXCHANGE\_INITKEY) 89](#_Toc276586211)

[11.2 Passive Diffie-Hellman (KEY\_EXCHANGE\_DH) 90](#_Toc276586212)

[11.3 UC-Secure Key Exchange (KEY\_EXCHANGE\_UCDH) 92](#_Toc276586213)

[12 Secure Channel 94](#_Toc276586214)

[12.1 Basic Secure Channel 94](#_Toc276586215)

[12.2 Adaptive Secure Channel (with Erasures) 94](#_Toc276586216)

[13 Special Encryption Schemes 94](#_Toc276586217)

[13.1 Paillier Encryption and Homomorphic Operations 94](#_Toc276586218)

[13.2 Attribute-Based Encryption 94](#_Toc276586219)

[13.3 Format-Preserving Encryption 94](#_Toc276586220)

[13.4 Order-Preserving Encryption 94](#_Toc276586221)

[13.5 Identity-Based Encryption 94](#_Toc276586222)

[14 Non-Interactive Primitives 94](#_Toc276586223)

[14.1 GetRandom 94](#_Toc276586224)

[14.2 Pseudorandom Function with Arbitrary Input-Output Lengths 95](#_Toc276586225)

[14.3 Pseudorandom Permutation with Arbitrary Input-Output Lengths 95](#_Toc276586226)

[14.4 Universal One-Way Hashing 95](#_Toc276586227)

[14.5 Perfect Universal Hash Functions 95](#_Toc276586228)

[14.6 Information-Theoretic MAC 95](#_Toc276586229)

[14.7 Key Derivation (HKDF) 95](#_Toc276586230)

# Template and Document Explanation

The protocol template is divided into three parts. The first contains general information about the protocol (name, reference, etc.); the second which is called “protocol parameters” contains the parties’ identities in the protocol, their inputs and outputs, and any common parameters that they may have; the third part contains the protocol specification. This last part is itself comprised of a number of parts. First, the specification of the protocol as an interactive game is given; this describes the flow of the protocol and presents a global point of view. Next, a separate specification is given for each party in the protocol, from the party’s local perspective.

The template is as follows:

|  |  |
| --- | --- |
| Protocol Name: | [Meaningful 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. |
| Common parameters: | [E.g., description of a DLOG group or something of the type] |
| 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 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: |  |

We remark that the division into steps is sometimes arbitrary; in some cases, an instruction could have belonged to the previous or following step. There are two different types of WAIT instructions, depending on whether or not the party has any computation that can be carried out while waiting for the next message.

* If there are no operations that can be carried out while waiting, then the WAIT instruction appears as a separate step, as shown here:

|  |  |
| --- | --- |
| Step 2: | WAIT for message X from Y |
| Step 3: | COMPUTE …  SEND … |

* If there are operations that can be carried out while waiting, then the WAIT instruction appears together with those instructions.

|  |  |
| --- | --- |
| Step 2: | WAIT for message X from Y  COMPUTE …  SEND … |

**Conventions:**

1. All variables are in bold and italics
2. All math symbols are bold (but not italicized)
3. The following reserved words are used:
   1. SEND:
   2. WAIT:
   3. COMPUTE:
   4. SAMPLE:
   5. RUN SUBPROTOCOL:
   6. OUTPUT:
   7. ACC/REJ:

**Discrete log parameter verification**

A discrete log group is represented by a triple **(*G,q,g*)**, where ***G*** is a group of order ***q***, and ***g*** is a generator of ***G***. In many protocols described below, **(*G,q,g*)** are *common parameters* that are used many times. In addition, the security of many of the protocols depends on the validity of the parameters; specifically, that ***q*** is prime and ***g*** is an element of ***G*** of order ***q*** (i.e., ***g*** is a generator). We define VALID\_PARAMS(***G,q,g***)=TRUE if and only if the above validity holds. Although VALID\_PARAMS is called inside many of the subprotocols, we stress that once specific parameters have been checked once, there is no need to rerun the check.

**References**

We stress that references to protocols are not given for purposes of credit, but rather as a pointer for further details and proofs of security. In order to reduce the number of reference points, we have used Hazay-Lindell as a reference wherever possible.

# Sigma protocols

## Schnorr’s Sigma-Protocol for DLOG (SIGMA\_DLOG)

|  |  |
| --- | --- |
| Protocol Name: | Schnorr’s Σ 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 the value ***h*** in ***G*** |
| References: | Protocol 6.1.1, page 148 of Hazay-Lindell |

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

|  |  |
| --- | --- |
| SIGMA\_DLOG Protocol Specification | |
| Step 1 (both): | V: IF **NOT**   * VALID\_PARAMS(***G,q,g***), AND * ***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***  OUTPUT ACC  ELSE  OUTPUT REJ  P: OUTPUT nothing |

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

|  |  |
| --- | --- |
| SIGMA\_DLOG Verifier (V) Specification | |
| Step 1: | IF **NOT**   * VALID\_PARAMS(***G,q,g***), AND * ***h*** ∈ ***G***   OUTPUT REJ |
| Step 2: | WAIT for a message ***a*** from P |
| Step 3: | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 4: | WAIT for a message ***z*** from P  IF ***gz* = *ahe***  OUTPUT ACC  ELSE  OUTPUT REJ |

Observe that the WAIT here in step 2 is not together with step 3 because V must receive ***a*** before it sends ***e*** to P.

**Sigma-Protocol Simulator:** This specification is needed for the OR of Sigma protocols; see Section ‎3.10.

|  |  |
| --- | --- |
| SIGMA\_DLOG Simulator (*M*) Specification | |
| Input: | Parameters (***G,q,g***) and ***t***, input ***h*** and a challenge ***e* ∈{0, 1}*t*** |
| Step 1: | SAMPLE a random ***z* ∈ *Zq*** |
| Step 2: | COMPUTE ***a = gz⋅h-e*** |
| Step 3: | OUTPUT **(*a,e,z*)** |

## Sigma-Protocol for Diffie-Hellman Tuples (SIGMA\_DH)

|  |  |
| --- | --- |
| 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 the input tuple **(*g,h,u,v*)** is a Diffie-Helman tuple. |
| References: | Protocol 6.2.4, page 152 of Hazay-Lindell |

|  |  |
| --- | --- |
| SIGMA\_DH Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common parameters: | A DLOG group description (***G,q,g***) and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input: (***h,u,v***) 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 |

|  |  |
| --- | --- |
| SIGMA\_DH Protocol Specification | |
| Step 1 (both): | V: IF **NOT**   * VALID\_PARAMS(***G,q,g***), AND * ***h*** is of order ***q***, AND * ***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*** and ***hz* = *bve***  OUTPUT ACC  ELSE  OUTPUT REJ  P: OUTPUT nothing |

|  |  |
| --- | --- |
| SIGMA\_DH 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 a message ***e*** from V |
| Step 3: | COMPUTE ***z* = *r* + *ew* mod *q***  SEND ***z*** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| SIGMA\_DH Verifier (V) Specification | |
| Step 1: | IF **NOT**   * VALID\_PARAMS(***G,q,g***), AND * ***h***  is of order ***q***, AND * ***u,v*** ∈ ***G***   OUTPUT REJ |
| Step 2: | WAIT for a message **(*a,b*)** from P |
| Step 3: | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 4: | WAIT for a message ***z*** from P  IF ***gz* = *aue*** and ***hz* = *bve***  OUTPUT ACC  ELSE  OUTPUT REJ |

**Sigma-Protocol Simulator** (see Section ‎3.10):

|  |  |
| --- | --- |
| SIGMA\_DH Simulator (*M*) Specification | |
| Input: | Parameters (***G,q,g***) and ***t***, input **(*h,u,v*)** and a challenge ***e* ∈{0, 1}*t*** |
| Step 1: | SAMPLE a random ***z* ∈ *Zq*** |
| Step 2: | COMPUTE ***a = gz⋅u-e*** and ***b = hz⋅v-e*** |
| Step 3: | OUTPUT **(**(***a,b***)***,e,z*)** |

## Sigma Protocol for Pedersen Commitment Knowledge (SIGMA\_PEDERSEN)

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol for Pedersen Commitment Knowledge |
| Protocol Reference: | SIGMA\_PEDERSEN |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a committer to prove that it knows the value committed to in the commitment **(*h,c*)** |
| References: | ?????  See Section for the description of Pedersen commitments |

|  |  |
| --- | --- |
| SIGMA\_PEDERSEN Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common Parameters: | A DLOG group description (***G,q,g***) and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input: **(*h,c*)** * P’s private input: values ***x,r* ∈ *Zq*** such that ***c* = *gr · hx*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_PEDERSEN Protocol Specification | |
| Step 1 (both): | V: IF **NOT**   * VALID\_PARAMS(***G,q,g***), AND * ***h*** is of order ***q***, AND * ***c*** ∈ ***G***   OUTPUT REJ  P: SAMPLE random values ***α* ∈ *Zq*** and ***β* ∈ *Zq*** and COMPUTE ***a* =*****hα⋅ gβ***  SEND ***a*** to V |
| Step 2 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3 (P): | COMPUTE ***u* = *α* + *ex*****mod *q***and ***v* = *β* + *er*****mod *q***  SEND **(*u,v*)** to V |
| Step 4 (both): | V: IF ***hu⋅gv=a⋅ce***  OUTPUT ACC  ELSE  OUTPUT REJ  P: OUTPUT nothing |

|  |  |
| --- | --- |
| SIGMA\_PEDERSEN Prover (P) Specification | |
| Step 1: | SAMPLE random values ***α* ∈ *Zq*** and ***β* ∈ *Zq*** and COMPUTE ***a* =*****hα⋅ gβ***  SEND ***a*** to V |
| Step 2: | WAIT for a message ***e*** from V |
| Step 3: | COMPUTE ***u* = *α* + *ex*****mod *q***and ***v* = *β* + *er*****mod *q***  SEND **(*u,v*)** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| SIGMA\_PEDERSEN Verifier (V) Specification | |
| Step 1: | IF **NOT**   * VALID\_PARAMS(***G,q,g***), AND * ***h*** is of order ***q***, AND * ***c*** ∈ ***G***   OUTPUT REJ |
| Step 2: | WAIT for a message ***a*** from P |
| Step 3: | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 4: | WAIT for a message **(*u,v*)** from P  IF ***hu⋅gv=a⋅ce***  OUTPUT ACC  ELSE  OUTPUT REJ |

**Sigma-Protocol Simulator** (see Section ‎3.10):

|  |  |
| --- | --- |
| SIGMA\_ Simulator (*M*) Specification | |
| Input: | Parameters (***G,q,g***) and ***t***, input **(*h,c*)** and a challenge ***e* ∈{0, 1}*t*** |
| Step 1: | SAMPLE random values ***u* ∈ *Zq*** and ***v* ∈ *Zq*** |
| Step 2: | COMPUTE ***a* = *hu⋅gv*** |
| Step 3: | OUTPUT **(*a,e,***(***u,v***)**)** |

## Sigma-Protocol that a Pedersen-Committed Value is x (SIGMA\_COMMITTED\_VALUE\_PEDERSEN)

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol that a Pedersen-Committed Value is *x* |
| Protocol Reference: | SIGMA\_COMMITTED\_VALUE\_PEDERSEN |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a committer to prove that the value committed to in the commitment **(*h, c*)** is ***x*** |
| References: | Since ***c* = *gr⋅hx***, it suffices to prove knowledge of ***r*** s.t. ***gr* = *c⋅h-x***. This is just a DLOG Sigma protocol.  See Section for the description of Pedersen commitments |

|  |  |
| --- | --- |
| SIGMA\_COMMITTED\_VALUE\_PEDERSEN Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common Parameters: | A DLOG group description (***G,q,g***) and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input: **(*h, c*)** and ***x*** * P’s private input: the value ***r* ∈ *Zq*** such that ***c* = *gr · hx*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_ COMMITTED\_VALUE\_PEDERSEN Protocol Specification | |
| Step 1: | RUN SIGMA\_DLOG with:   * Common parameters **(*G,q,g*)** and **t** * Common input: ***h’* = *c⋅h-x*** * P’s private input: a value ***r*∈ *Zq*** such that ***h’* = *gr*** |

**Sigma-Protocol Simulator** (see Section ‎3.10): same as SIGMA\_DLOG with inputs appropriately defined.

## Sigma Protocol for El Gamal Commitment Knowledge (SIGMA\_ELGAMAL\_COMMIT)

Note that an ElGamal commitment to ***x*** is a tuple **(*h,c1,c2*)** where ***h*** is a public key, and **(*c1,c2*)** are an encryption of ***x***.

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol for El Gamal Commitment Knowledge |
| Protocol Reference: | SIGMA\_ELGAMAL\_COMMIT |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a committer to prove that it knows the value committed to in the commitment **(*h*,*c1, c2*)** |
| References: | None: this is just a DLOG Sigma Protocol on the 1st element |

|  |  |
| --- | --- |
| SIGMA\_ELGAMAL\_COMMIT Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common parameters: | A DLOG group description (***G,q,g***) and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input statement: **(*h*,*c1,c2*)** * P’s private input: a value ***w*∈ *Zq*** such that ***h* = *gw***   (given ***w*** can decrypt and so this proves knowledge of committed value) |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_ELGAMAL\_COMMIT Protocol Specification | |
| Step 1: | RUN SIGMA\_DLOG with:   * Common parameters **(*G,q,g*)** and **t** * Common input: ***h*** (1st element of commitment) * P’s private input: a value ***w*∈ *Zq*** such that ***h* = *gw*** |

**Sigma-Protocol Simulator** (see Section ‎3.10): same as SIGMA\_DLOG with inputs appropriately defined.

## Sigma Protocol that an ElGamal-Committed Value is x (SIGMA\_COMMITTED\_VALUE\_ELGAMAL)

Note that an ElGamal commitment to ***x*** is a tuple **(*h,c1,c2*)** where ***h*** is a public key, and **(*c1,c2*)** are an encryption of ***x***.

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol that an El Gamal Committed Value is x |
| Protocol Reference: | SIGMA\_COMMITTED\_VALUE\_ELGAMAL |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a committer to prove that the value committed to in the commitment **(*h*,*c1, c2*)** is ***x*** |
| References: | None: this is just a DH Sigma Protocol |

|  |  |
| --- | --- |
| SIGMA\_COMMITTED\_VALUE\_ELGAMAL Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common parameters: | A DLOG group description (***G,q,g***) and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input statement: **(*h*,*c1,c2*)** and ***x*** * P’s private input: a value ***r*∈ *Zq*** such that ***c1*=*gr*** and ***c2* =*hr⋅x*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_COMMITTED\_VALUE\_ELGAMAL Protocol Specification | |
| Step 1: | RUN SIGMA\_DH with:   * Common parameters **(*G,q,g*)** and **t** * Common input: **(*g,h,u,v*) = (*g,h,c1,c2/x*)** * P’s private input: a value ***r*∈ *Zq*** such that ***c1*=*gr*** and ***c2/x* =*hr*** |

We remark that the public key ***h*** may also be part of the common parameters. It is not necessary for the prover to know the discrete log of ***h***.

**Sigma-Protocol Simulator** (see Section ‎3.10): same as SIGMA\_DH with inputs appropriately defined.

## Sigma Protocol of ElGamal Secret Key (SIGMA\_SK\_ELGAMAL)

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol for El Gamal Secret Key |
| Protocol Reference: | SIGMA\_SK\_ELGAMAL |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a party to prove that it knows the secret key to an ElGamal public key |
| References: | None: this is just a DLOG Sigma Protocol |

|  |  |
| --- | --- |
| SIGMA\_SK\_ELGAMAL Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common parameters: | A DLOG group description (***G,q,g***) and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input statement: an ElGamal public-key ***h*** * P’s private input: a value ***w*∈ *Zq*** such that ***h*=*gw*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_SK\_ELGAMAL Protocol Specification | |
| Step 1: | RUN SIGMA\_DLOG with:   * Common parameters **(*G,q,g*)** * Common input: ***h*** (the public key) * P’s private input: a value ***w*∈ *Zq*** such that ***h*=*gw*** |

**Sigma-Protocol Simulator** (see Section ‎3.10): same as SIGMA\_DLOG with inputs appropriately defined.

## Sigma Protocol that ElGamal-Encrypted Value is x (SIGMA\_ENCRYPTED\_VALUE\_ELGAMAL)

There are two versions of this protocol, depending upon if the prover knows the secret key or it knows the randomness used to generate the ciphertext.

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol that an El Gamal Encrypted Value is x |
| Protocol Reference: | SIGMA\_ENCRYPTED\_VALUE\_ELGAMAL |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used to prove that the value encrypted under ElGamal in the ciphertext **(*c1, c2*)** with public-key ***h*** is ***x*** |
| References: | None: this is just a DH Sigma Protocol |

### Version 1 – using knowledge of the secret key

|  |  |
| --- | --- |
| SIGMA\_ENCRYPTED\_VALUE\_ELGAMAL\_1 Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common parameters: | A DLOG group description (***G,q,g***) and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input statement: **(*c1,c2*)**, ***h*** and ***x*** * P’s private input: a value ***w*∈ *Zq*** such that ***h*=*gw*** and ***c2/c1w* = *x***   (in this case, P knows the *secret key*) |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_ENCRYPTED\_VALUE\_ELGAMAL\_1 Protocol Specification | |
| Step 1: | RUN SIGMA\_DH with:   * Common parameters **(*G,q,g*)** * Common input: **(*g,h,u,v*) = (*g,c1,h,c2/x*)** * P’s private input: a value ***w*∈ *Zq*** such that ***h*=*gw*** and ***c2/x* =*c1w*** |

**Sigma-Protocol Simulator** (see Section ‎3.10): same as SIGMA\_DLOG with inputs appropriately defined.

### Version 2 – using knowledge of the randomness used to encrypt

|  |  |
| --- | --- |
| SIGMA\_ENCRYPTED\_VALUE\_ELGAMAL\_2 Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common parameters: | A DLOG group description (***G,q,g***) and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input statement: **(*c1,c2*)**, ***h*** and ***x*** * P’s private input: a value ***r*∈ *Zq*** such that ***c1*=*gr*** and ***c2/x* = *hr***   (in this case, P knows the *randomness used to encrypt*) |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_ENCRYPTED\_VALUE\_ELGAMAL\_2 Protocol Specification | |
| Step 1: | RUN SIGMA\_DH with:   * Common parameters **(*G,q,g*)** * Common input: **(*g,h,u,v*) = (*g,h,c1,c2/x*)** * P’s private input: a value ***r*∈ *Zq*** such that ***c1*=*gr*** and ***c2/x* =*hr*** |

**Sigma-Protocol Simulator** (see Section ‎3.10): same as SIGMA\_DLOG with inputs appropriately defined.

## Sigma Protocol – AND of Multiple Statements (AND\_SIGMA)

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol – AND of Σ Protocols |
| Protocol Reference: | AND\_SIGMA |
| Protocol Type: | Sigma protocol transformation |
| Protocol Description: | This protocol is used for a prover to convince a verifier that the AND of any number of statements are true, where each statement can be proven by an associated Σ protocol. |
| References: | None: trivial |

|  |  |
| --- | --- |
| AND\_SIGMA Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common parameters: | Common parameters of subprotocols being used and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input: a series of ***m*** statements {***xi***} * P’s private input: a series of witnesses {***wi***} such that for every ***i*** **(*xi, wi*)∈ *Ri*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| AND\_SIGMA Protocol Specification | |
| Let (*ai,ei,zi*) denote the steps of a Σ protocol π*i* for proving that *xi* ∈ *LRi* | |
| Step 1 (P): | COMPUTE the 1st message ***ai*** for each ***i***, according to π*i*.  SEND **(*a1,…,am*)** to V |
| Step 2 (V): | VALIDATE any parameters as specified in protocols **π*i***  SAMPLE a single random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3 (P): | For every ***i***, COMPUTE ***zi*** as the continuation of **(*ai,e*)** in **π*i***  SEND **(*z1,…,zm*)** to V |
| Step 4 (both): | V: IF transcript **(*ai, e, zi*)** is accepting in **π*i*** on inputs ***xi***for every ***i***  OUTPUT ACC  ELSE  OUTPUT REJ  P: OUTPUT nothing |

|  |  |
| --- | --- |
| AND\_SIGMA Prover (P) Specification | |
| Step 1: | COMPUTE the 1st message ***ai*** for each ***i***, according to π*i*.  SEND **(*a1,…,am*)** to V |
| Step 2: | WAIT for a message ***e*** from V |
| Step 3: | For every ***i***, COMPUTE ***zi*** as the continuation of **(*ai,e*)** in **π*i***  SEND **(*z1,…,zm*)** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| AND\_SIGMA Verifier (V) Specification | |
| Step 1: | VALIDATE any parameters as specified in protocols **π*i***  WAIT for a message **(*a1,…,am*)** from P |
| Step 2: | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3: | WAIT for a message **(*z1,…,zm*)** from P  transcript **(*ai, e, zi*)** is accepting in **π*i*** on inputs ***xi***for every ***i***  OUTPUT ACC  ELSE  OUTPUT REJ |

**Sigma-Protocol Simulator** (see Section ‎3.10): run each Sigma protocol simulator with the same ***e***.

## Sigma Protocol – OR of 2 Statements (OR\_2\_SIGMA)

This protocol can be used when the subprotocols to be run have a specified Sigma-protocol simulator ***M***.

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol – OR of 2 Σ Protocols |
| Protocol Reference: | OR\_2\_SIGMA |
| Protocol Type: | Sigma protocol transformation |
| Protocol Description: | This protocol is used for a prover to convince a verifier that at least one of two statements is true, where each statement can be proven by an associated Σ protocol |
| References: | Protocol 6.4.1, page 159 of Hazay-Lindell |

|  |  |
| --- | --- |
| OR\_2\_SIGMA Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common parameters: | Common parameters of subprotocols being used and a soundness parameter ***t*** such that ***2t* < *q*** |
| Parties’ Inputs: | * Common input: pair **(*x0, x1*)** * P’s private input: ***w***such that **(*xb,w*) ∈ *R*** for some bit ***b*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| OR\_2\_SIGMA Protocol Specification | |
| Let (*ai,ei,zi*) denote the steps of a Σ protocol π*i* for proving that *xi* ∈ *LRi* | |
| Step 1 (P): | COMPUTE the first message ***ab***in **π*b***, using **(*xb,w*)** as input  SAMPLE a random challenge ***e*1*−b* ∈{0, 1}*t***  RUN the simulator ***M*** for **π*i*** on input **(*x1−b, e1−b*)** to obtain **(*a1−b*, *e1−b*, *z1−b*)**  SEND **(*a0,a1*)** to V |
| Step 2 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3 (P): | SET ***eb = e* ⊕ *e1−b***  COMPUTE the response ***zb***to **(*ab, eb*)** in **π*b***using input **(*xb,w*)**  SEND **(*e0, z0, e1, z1*)** to V |
| Step 4 (both): | V: IF   * ***e*0 ⊕ *e*1 = *e***, AND * Transcript **(*a0, e0, z0*)** is accepting in **π0**, on input ***x0***, AND * transcript**(*a1, e1, z1*)** is accepting in **π*1***, on input ***x1***   OUTPUT ACC  ELSE  OUTPUT REJ  P: OUTPUT nothing |

|  |  |
| --- | --- |
| OR\_2\_SIGMA Prover (P) Specification | |
| Step 1: | COMPUTE the first message ***ab***in **π*b***, using **(*xb,w*)** as input  SAMPLE a random challenge ***e*1*−b* ∈{0, 1}*t***  RUN the simulator ***M*** for **π*i*** on input **(*x1−b, e1−b*)** to obtain **(*a1−b*, *e1−b*, *z1−b*)**  SEND **(*a0,a1*)** to V |
| Step 2: | WAIT for message ***e*** from V |
| Step 3: | SET ***eb = e* ⊕ *e1−b***  COMPUTE the response ***zb***to **(*ab, eb*)** in **π*b***using input **(*xb,w*)**  SEND **(*e0, z0, e1, z1*)** to V |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| OR\_2\_SIGMA Verifier (V) Specification | |
| Step 1: | Validate parameters according to **π0, π1**  WAIT for message **(*a0,a1*)** from P |
| Step 2: | SAMPLE a random challenge ***e* ∈{0, 1}*t***  SEND ***e*** to P |
| Step 3: | WAIT for a message **(*e0, z0, e1, z1*)**from P |
| Step 4: | IF   * ***e*0 ⊕ *e*1 = *e***, AND * Transcript **(*a0, e0, z0*)** is accepting in **π0**, on input ***x0***, AND * transcript**(*a1, e1, z1*)** is accepting in **π*1***, on input ***x1***   OUTPUT ACC  ELSE  OUTPUT REJ |

**Sigma-Protocol Simulator** (see Section ‎3.10): use a random ***eb*** and then run the Sigma protocol simulator for each protocol with the resulting ***e0,e1*** values.

## Sigma Protocol – OR of Multiple Statements (OR\_MANY\_SIGMA)

This protocol can be used when all the subprotocols to be run have a specified Sigma-protocol simulator ***M***.

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol – OR of Many Σ Protocols |
| Protocol Reference: | OR\_MANY\_SIGMA |
| Protocol Type: | Sigma protocol transformation |
| Protocol Description: | This protocol is used for a prover to convince a verifier that at least ***k*** out of ***n*** statements is true, where each statement can be proven by an associated Σ protocol |
| References: | [CDS] |

## Sigma Protocols for Paillier (Damgard-Jurik Version)

# Zero-knowledge

## Zero-Knowledge from any Sigma Protocol (ZK\_FROM\_SIGMA)

|  |  |
| --- | --- |
| Protocol Name: | Zero-knowledge from any Sigma-Protocol |
| Protocol Reference: | ZK\_FROM\_SIGMA |
| Protocol Type: | Zero-knowledge proof (Sigma-protocol transformation) |
| Protocol Description: | This is a transformation that takes any Sigma protocol **π** and any *perfectly hiding commitment scheme* and yields a zero-knowledge proof. |
| References: | Protocol 6.5.1, page 161 of Hazay-Lindell |

|  |  |
| --- | --- |
| ZK\_FROM\_SIGMA Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common Parameters: | As needed for the Sigma protocol **π** and the (perfectly hiding) commitment scheme COMMIT |
| 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 |

|  |  |
| --- | --- |
| ZK\_FROM\_SIGMA Protocol Specification | |
| Step 1 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t*** |
| Step 2 (both): | RUN COMMIT.commit with V as the committer with input ***e***, and with P as the receiver |
| Step 2 (P): | COMPUTE the first message ***a***in **π**, using **(*x,w*)** as input  SEND ***a*** to V |
| Step 3 (both): | RUN COMMIT.decommit with V as the decomitter and P as the receiver |
| Step 4 (P): | IF COMMIT.decommit returns some ***e***  COMPUTE the response ***z***to **(*a*,*e*)**according to **π**  SEND ***z*** to V  OUTPUT nothing  ELSE (IF COMMIT.decommit returns INVALID)  OUTPUT ERROR (CHEAT\_ATTEMPT\_BY\_V) and HALT |
| Step 5 (V): | IF transcript **(*a, e, z*)** is accepting in **π**on input ***x***  OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| ZK\_FROM\_SIGMA Prover (P) Specification | |
| Step 1: | RUN the receiver in COMMIT.commit with V as the committer |
| Step 2: | COMPUTE the first message ***a***in **π**, using **(*x,w*)** as input  SEND ***a*** to V |
| Step 3: | RUN the receiver in COMMIT.decommit with V as the committer |
| Step 4: | IF COMMIT.decommit returns some ***e***  COMPUTE the response ***z***to **(*a*,*e*)**according to **π**  SEND ***z*** to V  OUTPUT nothing  ELSE (IF COMMIT.decommit returns INVALID)  OUTPUT ERROR (CHEAT\_ATTEMPT\_BY\_V) |

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

The above is proven to work when using any perfectly hiding commitment scheme. As such, the hash-based commitment can be used when relying on the random-oracle model heuristic. We stress that perfectly-binding commitment schemes do not necessarily suffice.

## Zero-Knowledge Proof of Knowledge from any Sigma Protocol (ZKPOK\_FROM\_SIGMA)

|  |  |
| --- | --- |
| Protocol Name: | Zero-knowledge proof of knowledge from any Sigma-protocol |
| Protocol Reference: | ZKPOK\_FROM\_SIGMA |
| Protocol Type: | ZK proof of knowledge (Sigma-protocol transformation) |
| Protocol Description: | This is a transformation that takes any Sigma protocol **π** and any *perfectly hiding trapdoor (equivocal) commitment scheme* and yields a zero-knowledge proof of knowledge. |
| References: | Protocol 6.5.4, page 165 of Hazay-Lindell |

|  |  |
| --- | --- |
| ZKPOK\_FROM\_SIGMA Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common Parameters: | As needed for the Sigma protocol **π** and the perfectly hiding trapdoor commitment scheme TRAP\_COMMIT |
| 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 |

|  |  |
| --- | --- |
| ZKPOK\_FROM\_SIGMA Protocol Specification | |
| Step 1 (V): | SAMPLE a random challenge ***e* ∈{0, 1}*t*** |
| Step 2 (both): | RUN TRAP\_COMMIT.commit with V as the committer with input ***e***, and with P as the receiver; let **trap** be P’s output from this phase |
| Step 2 (P): | COMPUTE the first message ***a***in **π**, using **(*x,w*)** as input  SEND ***a*** to V |
| Step 3 (both): | RUN TRAP\_COMMIT.decommit with V as the decomitter and P as the receiver |
| Step 4 (P): | IF TRAP\_COMMIT.decommit returns some ***e***  COMPUTE the response ***z***to **(*a*,*e*)**according to **π**  SEND ***z*** and **trap** to V  OUTPUT nothing  ELSE (IF TRAP\_COMMIT.decommit returns INVALID)  OUTPUT ERROR (CHEAT\_ATTEMPT\_BY\_V) and HALT |
| Step 5 (V): | IF   * TRAP\_COMMIT.valid(***T***,**trap**) = 1, where ***T***  is the transcript from the commit phase, AND * Transcript **(*a, e, z*)** is accepting in **π**on input ***x***   OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| ZKPOK\_FROM\_SIGMA Prover (P) Specification | |
| Step 1: | RUN the receiver in TRAP\_COMMIT.commit with V as the committer; let **trap** be the output |
| Step 2: | COMPUTE the first message ***a***in **π**, using **(*x,w*)** as input  SEND ***a*** to V |
| Step 3: | RUN the receiver in TRAP\_COMMIT.decommit with V as the committer |
| Step 4: | IF TRAP\_COMMIT.decommit returns some ***e***  COMPUTE the response ***z***to **(*a*,*e*)**according to **π**  SEND ***z*** and **trap** to V  OUTPUT nothing  ELSE (IF COMMIT.decommit returns INVALID)  OUTPUT ERROR (CHEAT\_ATTEMPT\_BY\_V) |

|  |  |
| --- | --- |
| ZKPOK\_FROM\_SIGMA Verifier (V) Specification | |
| Step 1: | SAMPLE a random challenge ***e* ∈{0, 1}*t*** |
| Step 2: | RUN TRAP\_COMMIT.commit as the committer with input ***e***, and with P as the receiver |
| Step 3: | WAIT for a message ***a*** from P |
| Step 4: | RUN TRAP\_COMMIT.decommit as the decommitter, with P as the receiver |
| Step 5: | WAIT for a message **(*z*,trap)** from P  IF   * TRAP\_COMMIT.valid(***T***,**trap**) = 1, where ***T***  is the transcript from the commit phase, AND * Transcript **(*a, e, z*)** is accepting in **π**on input ***x***   OUTPUT ACC  ELSE  OUTPUT REJ |

The above protocol uses a trapdoor commitment scheme; see Section XXX. Note that the receiver’s output from the commit stage is a trapdoor **trap** that can be used to open a commitment to any value. In addition, there exists a function TRAP\_COMMIT.valid(***T***,**trap**) that returns 1 if and only if **trap** is the valid trapdoor when the transcript of the commit phase is ***T***.

## ZKPOK from any Sigma-Protocol – ROM (Fiat-Shamir) (ZKPOK\_FS\_SIGMA)

|  |  |
| --- | --- |
| Protocol Name: | ZKPOK from any Sigma-protocol (Fiat-Shamir) |
| Protocol Reference: | ZKPOK\_FS\_SIGMA |
| Protocol Type: | ZK proof of knowledge (Sigma-protocol transformation) |
| Protocol Description: | This is a transformation that takes any Sigma protocol **π** and a random oracle (instantiated with any hash function) **H** and yields a zero-knowledge proof of knowledge. |
| References: | [FS] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO 1986*, pages 186-194. |

|  |  |
| --- | --- |
| ZKPOK\_FS\_SIGMA Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common Parameters: | As needed for the Sigma protocol **π** |
| 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 |

|  |  |
| --- | --- |
| ZKPOK\_FS\_SIGMA Protocol Specification | |
| Step 1 (P): | COMPUTE the first message ***a***in **π**, using **(*x,w*)** as input  COMPUTE ***e*=H(*x*,*a*)**  COMPUTE the response ***z***to **(*a*,*e*)**according to **π**  SEND **(*a,e,z*)** to V  OUTPUT nothing |
| Step 2 (V): | IF   * ***e*=H(*x*,*a*)**, AND * Transcript **(*a, e, z*)** is accepting in **π**on input ***x***   OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| ZKPOK\_FS\_SIGMA Prover (P) Specification | |
| Step 1: | COMPUTE the first message ***a***in **π**, using **(*x,w*)** as input  COMPUTE ***e*=H(*x*,*a*)**  COMPUTE the response ***z***to **(*a*,*e*)**according to **π**  SEND **(*a,e,z*)** to V  OUTPUT nothing |

|  |  |
| --- | --- |
| ZKPOK\_FS\_SIGMA Verifier (V) Specification | |
| Step 1: | WAIT for a message **(*a,e,z*)** from P  IF   * ***e*=H(*x*,*a*)**, AND * Transcript **(*a, e, z*)** is accepting in **π**on input ***x***   OUTPUT ACC  ELSE  OUTPUT REJ |

## UC-Secure ZKPOK from any Sigma-Protocol (UCZKPOK\_FROM\_SIGMA)

|  |  |
| --- | --- |
| Protocol Name: | UC Secure ZKPOK from any Sigma-protocol |
| Protocol Reference: | UCZKPOK\_FROM\_SIGMA |
| Protocol Type: | Universally Composable ZKPOK (Sigma-protocol transformation) |
| Protocol Description: | This is a transformation that takes any Sigma protocol **π** and any *universally composable commitment* and yields a universally composable zero-knowledge proof of knowledge. |
| References: | [LP11] |

|  |  |
| --- | --- |
| UCZKPOK\_FROM\_SIGMA Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common Parameters: | A soundness parameter ***T*** and the parameters needed for the Sigma protocol **π** and the universally composable commitment scheme UC\_COMMIT. |
| 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 |

|  |  |
| --- | --- |
| UCZKPOK\_FROM\_SIGMA Protocol Specification | |
| Step 1 (P): | FOR ***i* = 1** to ***T***  COMPUTE the first message ***ai***in **π**, using **(*x,w*)** as input  (use fresh randomness each time)  COMPUTE the third message ***zi0***in **π**, using **(*x,w*)** as input, ***ai*** as the first message, and ***e* = 0*t*** as the challenge  COMPUTE the third message ***zi1***in **π**, using **(*x,w*)** as input, ***ai*** as the first message, and ***e* = 1*t*** as the challenge |
| Step 2 (both): | FOR ***i* = 1** to ***T***  RUN UC\_COMMIT.commit with P as the committer with input ***ai***, and with V as the receiver  RUN UC\_COMMIT.commit with P as the committer with input ***zi0***, and with V as the receiver  RUN UC\_COMMIT.commit with P as the committer with input ***zi1***, and with V as the receiver |
| Step 2 (V): | SAMPLE a random challenge ***E* ∈{0, 1}*T***  SEND ***E*** to P |
| Step 3 (both): | Let ***E = E1,…,ET***  FOR ***i* = 1** to ***T***  RUN UC\_COMMIT.decommit with P as the committer and with V as the receiver in order to reveal ***ziEi*** |
| Step 5 (V): | IF   * All decommit calls are valid, AND * FOR ***i* = 1** to ***T****,* transcript **(*ai, Ei, ziEi*)** is accepting in **π**on input ***x***   OUTPUT ACC  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| UCZKPOK\_FROM\_SIGMA Prover (P) Specification | |
| Step 1: | FOR ***i* = 1** to ***T***  COMPUTE the first message ***ai***in **π**, using **(*x,w*)** as input  (use fresh randomness each time)  COMPUTE the third message ***zi0***in **π**, using **(*x,w*)** as input, ***ai*** as the first message, and ***e* = 0*t*** as the challenge  COMPUTE the third message ***zi1***in **π**, using **(*x,w*)** as input, ***ai*** as the first message, and ***e* = 1*t*** as the challenge |
| Step 2: | FOR ***i* = 1** to ***T***  RUN UC\_COMMIT.commit as the committer with input ***ai***, and with V as the receiver  RUN UC\_COMMIT.commit as the committer with input ***zi0***, and with V as the receiver  RUN UC\_COMMIT.commit as the committer with input ***zi1***, and with V as the receiver |
| Step 3: | WAIT for a message ***E***  from V |
| Step 4: | Let ***E = E1,…,ET***  FOR ***i* = 1** to ***T***  RUN UC\_COMMIT.decommit as the committer and with V as the receiver in order to reveal ***ziEi*** |

|  |  |
| --- | --- |
| UCZKPOK\_FROM\_SIGMA Verifier (V) Specification | |
| Step 1: | FOR ***i* = 1** to ***T***  RUN UC\_COMMIT.commit as the receiver, with P as the committer, to obtain a commitment to ***ai***  RUN UC\_COMMIT.commit as the receiver, with P as the committer, to obtain a commitment to ***zi0***  RUN UC\_COMMIT.commit as the receiver, with P as the committer, to obtain a commitment to ***zi1*** |
| Step 2: | SAMPLE a random challenge ***E* ∈{0, 1}*T***  SEND ***E*** to P |
| Step 3: | FOR ***i* = 1** to ***T***  RUN UC\_COMMIT.decommit with P as the committer and with V as the receiver in order to reveal ***ziEi*** |
| Step 4: | IF   * All decommit calls are valid, AND * FOR ***i* = 1** to ***T****,* transcript **(*ai, Ei, ziEi*)** is accepting in **π**on input ***x***   OUTPUT ACC  ELSE  OUTPUT REJ |

# Commitment schemes

## Pedersen Commitment (COMMIT\_PEDERSEN)

|  |  |
| --- | --- |
| Protocol Name: | Pedersen commitment |
| Protocol Reference: | COMMIT\_PEDERSEN |
| Protocol Type: | Perfectly-Hiding Commitment |
| Protocol Description: | This commitment is also a trapdoor commitment in the sense that the receiver after the commitment phase has a trapdoor value, that if known by the committer would enable it to decommit to any value. |
| References: | Protocol 6.5.3, page 164 of Hazay-Lindell |

|  |  |
| --- | --- |
| COMMIT\_PEDERSEN Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | A DLOG group description (***G,q,g***) |
| Parties’ Inputs: | C’s private input: a value ***x* ∈ *Zq*** |
| Parties’ Outputs: | * C: nothing * R’s output from the COMMIT phase:   + A trapdoor **trap** (optional) * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x* ∈ *Zq*** |

|  |  |
| --- | --- |
| COMMIT\_PEDERSEN Protocol Specification | |
| Commit phase | |
| Step 1 (Both): | C: IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT  R: SAMPLE a random value ***a* ∈ *Zq***  COMPUTE ***h* = *ga***  SEND ***h*** to C |
| Step 2 (Both): | C: IF **NOT** ***h*∈ *G***  REPORT ERROR and HALT  SAMPLE a random value ***r* ∈ *Zq***  COMPUTE ***c* = *gr · hx***  SEND ***c***  R: OUTPUT trapdoor **trap = *a*** (and STORE value ***c***) |
| Decommit phase | |
| Step 1 (C): | SEND **(*r, x*)** to R |
| Step 2 (R): | IF ***c* = *gr · hx*** AND ***x* ∈ *Zq***  OUTPUT ACC and value ***x***  ELSE  OUTPUT REJ |

|  |  |
| --- | --- |
| COMMIT\_PEDERSEN Committer (C) Specification | |
| Commit phase | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT  WAIT for ***h*** from R |
| Step 2: | IF **NOT** ***h*∈ *G***  REPORT ERROR and HALT  SAMPLE a random value ***r* ∈ *Zq***  COMPUTE ***c* = *gr · hx***  SEND ***c*** |
| Decommit phase | |
| Step 1: | SEND **(*r, x*)** to R |
| Step 2: | OUTPUT nothing |

|  |  |
| --- | --- |
| COMMIT\_PEDERSEN Receiver (R) Specification | |
| Commit phase | |
| Step 1: | SAMPLE a random value ***a* ∈ *Zq***  COMPUTE ***h* = *ga***  SEND ***h*** to C |
| Step 2: | WAIT for message ***c*** from C  OUTPUT trapdoor **trap = *a*** (and STORE value ***c***) |
| Decommit phase | |
| Step 1: | WAIT for **(*r, x*)** from C |
| Step 2: | IF ***c* = *gr · hx*** AND ***x* ∈ *Zq***  OUTPUT ACC and value ***x***  ELSE  OUTPUT REJ |

## Pedersen-Hash Commitment (COMMIT\_HASH\_PEDERSEN)

|  |  |
| --- | --- |
| Protocol Name: | Pedersen-Hash commitment |
| Protocol Reference: | COMMIT\_HASH\_PEDERSEN |
| Protocol Type: | Perfectly-Hiding Commitment |
| Protocol Description: | This is a perfectly-hiding commitment that can be used to commit to a ***value of any length***. It is also a trapdoor commitment as is the basic Pedersen commitment. The only difference is that the proof that you know the committed value (SIGMA\_PEDERSEN) is not valid here. We stress that SIGMA\_COMMITTED\_VALUE\_PEDERSEN is still relevant, with the only difference that H(x) is used in place of x and the verifier receives x and computes H(x) itself. |
| References: | Protocol 6.5.3, page 164 of Hazay-Lindell |

|  |  |
| --- | --- |
| COMMIT\_HASH\_PEDERSEN Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | A DLOG group description (***G,q,g***) |
| Parties’ Inputs: | C’s private input: a value ***x*** of any length |
| Parties’ Outputs: | * C: nothing * R’s output from the COMMIT phase:   + A trapdoor **trap** (optional) * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x*** |

|  |
| --- |
| COMMIT\_HASH\_PEDERSEN Protocol Specification |
| Run COMMIT\_PEDERSEN to commit to value H(*x*). For decommitment, send *x* and the receiver verifies that the commitment was to H(*x*). |

## ElGamal Commitment(COMMIT\_ELGAMAL)

|  |  |
| --- | --- |
| Protocol Name: | ElGamal commitment |
| Protocol Reference: | COMMIT\_ELGAMAL |
| Protocol Type: | Perfectly-Binding Commitment |
| Protocol Description: |  |
| References: | None: this is a commitment using any public-key encryption scheme, adapted specifically to ElGamal. |

|  |  |
| --- | --- |
| COMMIT\_ELGAMAL Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | A DLOG group description (***G,q,g***) |
| Parties’ Inputs: | C’s private input: a value ***x* ∈ *G*** (Important: this assumes a mapping function to map strings into the group and group elements back to strings) |
| Parties’ Outputs: | * C: nothing * R’s output from the COMMIT phase:   + nothing * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x* ∈ *G*** |

|  |  |
| --- | --- |
| COMMIT\_ELGAMAL Protocol Specification | |
| Commit phase | |
| Step 1 (C): | IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT  SAMPLE random values ***a,r* ∈ *Zq***  COMPUTE ***h* = *ga***  COMPUTE ***u* = *gr*** and ***v* = *hr⋅ x***  SEND ***c* =(*h,u,v*)** to R |
| Step 2 (R): | WAIT for a value ***c***  STORE ***c*** |
| Decommit phase | |
| Step 1 (C): | SEND **(*r, x*)** to R |
| Step 2 (R): | Let ***c* = (*h,u,v*)**; if not of this format, output REJ  IF **NOT**   * VALID\_PARAMS(***G,q,g***), AND * ***h*∈*G***, AND * ***u=gr*** *,*AND * ***v = hr*** ⋅ ***x***, AND * ***x* ∈ *G***   OUTPUT REJ  ELSE  OUTPUT ACC and value ***x*** |

|  |  |
| --- | --- |
| COMMIT\_ELGAMAL Committer (C) Specification | |
| Commit phase | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT  SAMPLE random values ***a,r* ∈ *Zq***  COMPUTE ***h* = *ga***  COMPUTE ***u* = *gr*** and ***v* = *hr*** ⋅ ***x***  SEND ***c* =(*h,u,v*)** to R |
| Decommit phase | |
| Step 1: | SEND **(*r, x*)** to R |
| Step 2: | OUTPUT nothing |

|  |  |
| --- | --- |
| COMMIT\_ELGAMAL Receiver (R) Specification | |
| Commit phase | |
| Step 1: | WAIT for a value ***c***  STORE ***c*** |
| Decommit phase | |
| Step 1: | WAIT for **(*r, x*)** from C |
| Step 4: | Let ***c* = (*h,u,v*)**; if not of this format, output REJ  IF **NOT**   * VALID\_PARAMS(***G,q,g***), AND * ***h*∈*G***, AND * ***u=gr*** *,*AND * ***v = hr*** ⋅ ***x***, AND * ***x* ∈ *G***   OUTPUT REJ  ELSE  OUTPUT ACC and value ***x*** |

**Note 1:** if many commitments are sent, the same ***h*** can be used for all.

**Note 2:** This commitment scheme assumes that the string to be committed to can be efficiently mapped into the group, and that its inverse is also efficient.

## ElGamal-Hash Commitment (COMMIT\_HASH\_ELGAMAL)

|  |  |
| --- | --- |
| Protocol Name: | ElGamal-Hash commitment |
| Protocol Reference: | COMMIT\_HASH\_ELGAMAL |
| Protocol Type: | Computationally-Binding and Computationally-Hiding Commitment |
| Protocol Description: | This is a commitment that can be used to commit to a ***value of any length***. This cannot be used as an extractable commitment by applying a Sigma protocol, as is the basic ElGamal commitment. In particular, the proof that you know the committed value (SIGMA\_ELGAMAL) is not valid here. We stress that SIGMA\_COMMITTED\_VALUE\_ELGAMAL is still relevant, with the only difference that H(x) is used in place of x and the verifier receives x and computes H(x) itself. |
| References: | Protocol 6.5.3, page 164 of Hazay-Lindell |

|  |  |
| --- | --- |
| COMMIT\_HASH\_ELGAMAL Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | A DLOG group description (***G,q,g***) |
| Parties’ Inputs: | C’s private input: a value ***x*** of any length |
| Parties’ Outputs: | * C: nothing * R’s output from the COMMIT phase:   + A trapdoor **trap** (optional) * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x*** |

|  |
| --- |
| COMMIT\_HASH\_ELGAMAL Protocol Specification |
| Run COMMIT\_ELGAMAL to commit to value H(*x*). For decommitment, send *x* and the receiver verifies that the commitment was to H(*x*). |

## Hash-Based Commitment (Basic) (COMMIT\_HASH\_BASIC)

|  |  |
| --- | --- |
| Protocol Name: | Hash-based commitment (heuristic) |
| Protocol Reference: | COMMIT\_HASH\_BASIC |
| Protocol Type: | Computationally hiding and binding commitment |
| Protocol Description: | This is a commitment scheme based on hash functions. It can be viewed as a random-oracle scheme, but its security can also be viewed as a *standard assumption* on modern hash functions. Note that computational binding follows from the standard collision resistance assumption. (When viewing the hash as a *random oracle*, the scheme is actually a fully trapdoor commitment.) |
| References: | Folklore |

|  |  |
| --- | --- |
| COMMIT\_HASH\_BASIC Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | An agreed-upon hash function **H**, and a security parameter ***n*** |
| Parties’ Inputs: | C’s private input: a value ***x* ∈{0*,* 1}*t*** |
| Parties’ Outputs: | * C: nothing * R’s output from the COMMIT phase:   + nothing * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x* ∈ {0*,* 1}*t*** |

|  |  |
| --- | --- |
| COMMIT\_HASH\_BASIC Protocol Specification | |
| Commit phase | |
| Step 1 (C): | SAMPLE a random value ***r* ∈ {0*,* 1}*n***  COMPUTE ***c* = *H*(*r,x*)** (c concatenated with r)  SEND ***c*** to R |
| Step 2 (R): | WAIT for a value ***c***  STORE ***c*** |
| Decommit phase | |
| Step 1 (C): | SEND **(*r, x*)** to R |
| Step 2 (R): | IF **NOT**   * ***c* = *H*(*r,x*)**, AND * ***x* ∈{0*,* 1}*t***   OUTPUT REJ  ELSE  OUTPUT ACC and value ***x*** |

|  |  |
| --- | --- |
| COMMIT\_HASH\_BASIC Committer (C) Specification | |
| Commit phase | |
| Step 1: | SAMPLE a random value ***r* ∈ {0*,* 1}*n***  COMPUTE ***c* = *H*(*r,x*)** (c concatenated with r)  SEND ***c*** to R |
| Decommit phase | |
| Step 1: | SEND **(*r, x*)** to R |
| Step 2: | OUTPUT nothing |

|  |  |
| --- | --- |
| COMMIT\_HASH\_BASIC Receiver (R) Specification | |
| Commit phase | |
| Step 1: | WAIT for a value ***c***  STORE ***c*** |
| Decommit phase | |
| Step 1: | WAIT for **(*r, x*)** from C |
| Step 4: | IF **NOT**   * ***c* = *H*(*r,x*)**, AND * ***x* ∈{0*,* 1}*t***   OUTPUT REJ  ELSE  OUTPUT ACC and value ***x*** |

## Hash-Based Statistically-Hiding Commitment (COMMIT\_HASH)

|  |  |
| --- | --- |
| Protocol Name: | Hash-based commitment (rigorous) |
| Protocol Reference: | COMMIT\_HASH |
| Protocol Type: | Statistically-Hiding Commitment |
| Protocol Description: |  |
| References: | Dodis, lecture notes on commitments, Section 2.3 <http://cs.nyu.edu/courses/fall08/G22.3210-001/lect/lecture14.pdf> |

|  |  |
| --- | --- |
| COMMIT\_HASH Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | A security parameter ***n***, an agreed-upon collision-resistant hash function **H** with output length ***n***, and an agreed-upon perfect universal hash function with input length **3*n*** and output length ***n***. |
| Parties’ Inputs: | C’s private input: a value ***x* ∈{0*,* 1}*n*** |
| Parties’ Outputs: | * C: nothing * R’s output from the COMMIT phase:   + nothing * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x* ∈ {0*,* 1}*n*** |

|  |  |
| --- | --- |
| COMMIT\_HASH Protocol Specification | |
| Commit phase | |
| Step 1 (C): | SAMPLE a random value ***r* ∈ {0*,* 1}3*n***  SAMPLE a random universal hash function ***u*** subject to ***u*(*r*) = *x***  SET ***c* =(*u,H*(*r*))**  SEND ***c*** to R |
| Step 2 (R): | WAIT for a value ***c***  STORE ***c*** |
| Decommit phase | |
| Step 1 (C): | SEND **(*r, x*)** to R |
| Step 2 (R): | IF **NOT**   * ***c* = (*u,H*(*r*))**, AND * ***u*(*r*) = *x***, AND * ***x* ∈{0*,* 1}*n***   OUTPUT REJ  ELSE  OUTPUT ACC and value ***x*** |

|  |  |
| --- | --- |
| COMMIT\_HASH Committer (C) Specification | |
| Commit phase | |
| Step 1: | SAMPLE a random value ***r* ∈ {0*,* 1}3*n***  SAMPLE a random universal hash function ***u*** subject to ***u*(*r*) = *x***  SET ***c* =(*u,H*(*r*))**  SEND ***c*** to R |
| Decommit phase | |
| Step 1: | SEND **(*r, x*)** to R |
| Step 2: | OUTPUT nothing |

|  |  |
| --- | --- |
| COMMIT\_HASH Receiver (R) Specification | |
| Commit phase | |
| Step 1: | WAIT for a value ***c***  STORE ***c*** |
| Decommit phase | |
| Step 1: | WAIT for **(*r, x*)** from C |
| Step 4: | IF **NOT**   * ***c* = (*u,H*(*r*))**, AND * ***u*(*r*) = *x***, AND * ***x* ∈{0*,* 1}*n***   OUTPUT REJ  ELSE  OUTPUT ACC and value ***x*** |

## Equivocal Commitments (COMMIT\_EQUIVOCAL)

|  |  |
| --- | --- |
| Protocol Name: | Equivocal commitment |
| Protocol Reference: | COMMIT\_EQUIVOCAL |
| Protocol Type: | Equivocal commitment |
| Protocol Description: | This is a protocol to obtain an equivocal commitment from any commitment with a ZK-protocol of the commitment value.  The equivocality property means that a simulator can decommit to any value it needs (needed for proofs of security). |
| References: | None (but appears implicitly in [L01]) |

|  |  |
| --- | --- |
| COMMIT\_EQUIVOCAL Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | As needed for any commitment scheme |
| Parties’ Inputs: | C’s private input: a value ***x* ∈{0*,* 1}*t*** |
| Parties’ Outputs: | * C: nothing * R’s output from the COMMIT phase:   + nothing * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x* ∈{0*,* 1}*t*** |

|  |  |
| --- | --- |
| COMMIT\_EQUIVOCAL Protocol Specification | |
| Commit phase | |
| Step 1 (Both): | RUN any COMMIT protocol for C to commit to ***x*** |
| Decommit phase, using ZK protocol π of decommitment value | |
| Step 1 (C): | SEND ***x*** to R |
| Step 2 (Both): | Run **π** with C as the prover and R as the verifier, that ***x*** is the correct decommitment value  R: IF verifier-output of **π** is ACC  OUTPUT ACC and ***x***  ELSE  OUTPUT REJ |

This protocol has two instantiations currently available (with ZK transformation from the Sigma protocol):

1. Use COMMIT\_PEDERSEN and SIGMA\_COMMITTED\_VALUE\_PEDERSEN
2. Use COMMIT\_ELGAMAL and SIGMA\_COMMITTED\_VALUE\_ELGAMAL

## Extractable Commitments(COMMIT\_EXTRACT)

|  |  |
| --- | --- |
| Protocol Name: | Extractable commitment |
| Protocol Reference: | COMMIT\_EXTRACT |
| Protocol Type: | Extractable commitment |
| Protocol Description: | This is a protocol to obtain an extractable commitment from any commitment with a Sigma-protocol for the commitment (i.e., that the committed value is known). The extraction property means that a simulator can extract the committed value (needed for proofs of security). |
| References: | None: just commit and ZKPOK |

|  |  |
| --- | --- |
| COMMIT\_EXTRACT Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | As needed for any commitment scheme |
| Parties’ Inputs: | C’s private input: a value ***x* ∈{0*,* 1}*t*** |
| Parties’ Outputs: | * C: nothing * R’s output from the COMMIT phase:   + ACC or REJ * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x* ∈{0*,* 1}*t*** |

|  |  |
| --- | --- |
| COMMIT\_EXTRACT Protocol Specification | |
| Commit phase, using ZKPOK protocol π that know committed value | |
| Step 1 (Both): | RUN any COMMIT protocol for C to commit to ***x*** |
| Step 2 (Both): | Run **π** with C as the prover and R as the verifier, that the prover knows the committed value ***x***  R: IF verifier-output of **π** is ACC  OUTPUT ACC  ELSE  OUTPUT REJ and HALT |
| Decommit phase | |
| Step 1 (C): | RUN DECOMMIT for R to receive ***x*** |

This protocol has two instantiations currently available (with ZKPOK transformation from the Sigma protocol):

1. Use COMMIT\_PEDERSEN and SIGMA\_ PEDERSEN
2. Use COMMIT\_ELGAMAL and SIGMA\_ ELGAMAL\_COMMIT

Note that if many commitments are used, then in the case of ElGamal, the Sigma protocol can be run once only. This can be an important optimization and so this option must be available.

## Fully Trapdoor Commitments

### Fully Trapdoor using Sigma Protocols(COMMIT\_DOUBLE\_TRAPDOOR)

|  |  |
| --- | --- |
| Protocol Name: | Fully trapdoor (equivocal and extractable) commitment |
| Protocol Reference: | COMMIT\_DOUBLE\_TRAPDOOR |
| Protocol Type: | Equivocal and extractable commitment |
| Protocol Description: | This is a protocol to obtain an equivocal and extractable commitment from any commitment with a Sigma-protocol of knowledge of the commitment, and a Sigma-protocol of the commitment value. This commitment scheme has the property that a simulator can extract the committed value and decommit to any value it needs (needed for proofs of security) |
| References: | None: just both equivocal and extractable |

|  |  |
| --- | --- |
| COMMIT\_DOUBLE\_TRAPDOOR Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | As needed for any commitment scheme |
| Parties’ Inputs: | C’s private input: a value ***x* ∈{0*,* 1}*t*** |
| Parties’ Outputs: | * C: nothing * R’s output from the COMMIT phase:   + nothing * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x* ∈{0*,* 1}*t*** |

|  |  |
| --- | --- |
| COMMIT\_ DOUBLE\_TRAPDOOR Protocol Specification | |
| Commit phase, using ZKPOK protocol π that know committed value | |
| Step 1 (Both): | RUN any COMMIT protocol for C to commit to ***x*** |
| Step 2 (Both) | Run **π** with C as the prover and R as the verifier, that the prover knows the committed value ***x***  R: IF verifier-output of **π** is ACC  OUTPUT ACC  ELSE  OUTPUT REJ and HALT |
| Decommit phase, using ZK protocol π’ of decommitment value | |
| Step 1 (C): | SEND ***x*** to R |
| Step 2 (Both): | Run **π’** with C as the prover and R as the verifier, that ***x*** is the correct decommitment value  R: IF verifier-output of **π’** is ACC  OUTPUT ACC and ***x***  ELSE  OUTPUT REJ |

This protocol has two instantiations currently available:

1. Use COMMIT\_PEDERSEN, SIGMA\_PEDERSEN and SIGMA\_COMMITTED\_VALUE\_PEDERSEN
2. Use COMMIT\_ELGAMAL, SIGMA\_ELGAMAL\_COMMIT and SIGMA\_COMMITTED\_VALUE\_ELGAMAL

As in the case of extractable commitments, in the case of many commitments and ElGamal, it is possible to run the Sigma protocol in the commitment stage only once. (This is in contrast to the Sigma protocol of the decommitment stage that cannot be saved.)

### Fully Trapdoor Hash (ROM) (COMMIT\_DOUBLE\_TRAPDOOR\_HASH\_ROM)

|  |  |
| --- | --- |
| Protocol Name: | Full Trapdoor Commitment via Hash (ROM) |
| Protocol Reference: | COMMIT\_DOUBLE\_TRAPDOOR \_HASH\_ROM |
| Protocol Type: | A fully trapdoor (extractable and equivocal) commitment |
| Protocol Description: | This is a commitment scheme that is fully trapdoor when viewing the hash function as a random oracle. |
| References: | Folklore |
| The Protocol: | Run COMMIT\_HASH\_BASIC |

### Fully Trapdoor in the CRS Model (COMMIT\_DOUBLE\_TRAPDOOR\_DDH\_CRS)

Just like the UC protocol, but use ElGamal encryption instead. Commit is just an ElGamal encryption, and decommit is like equivocal…

## Additive Homomorphic Operation on Pedersen Commitments (PEDERSEN\_ADD)

The Pedersen commitment scheme is additively homomorphic in ***Zq***, as follows. Given ***c1* = *grhx*** and ***c2* = *gshy***, observe that ***c1⋅c2*** (with multiplication in the group) equals ***gr+shx+y***. In order to rerandomize, multiply again by ***gu*** for a random ***u***.

|  |  |
| --- | --- |
| Protocol Name: | Homomorphic addition for Pedersen |
| Protocol Reference: | PEDERSEN\_ADD |
| Protocol Type: | Homomorphic operation on commitments |
| Protocol Description: | A method for constructing a random Pedersen commitment to ***x*+*y* mod *q***, given a Pedersen commitment to ***x*** and a Pedersen commitment to ***y*** (without knowing ***x*** or ***y***) |
| References: | None |

|  |  |
| --- | --- |
| PEDERSEN\_ADD Protocol Parameters | |
| Party’s Identity: | A single party P |
| Common parameters: | A DLOG group description (***G,q,g***) |
| Party’s Input: | Two Pedersen commitment ***c1,c2* ∈*G*** |
| Party’s Output: | A single commitment ***c*** such that if ***c1*** is a commitment to ***x*** and ***c2*** is a commitment to ***y***, then ***c*** is a random commitment to ***x*+*y*** |

|  |  |
| --- | --- |
| PEDERSEN\_ADD Protocol Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g***), REPORT ERROR and HALT  SAMPLE a random value ***u* ∈ *Zq***  COMPUTE ***c* = *gu*⋅*c1*⋅*c2***  OUTPUT ***c*** |

## Multiplicative Homomorphic Operation on ElGamal Commitments (ELGAMAL\_MULT)

The ElGamal commitment scheme is mulitplicatively homomorphic in ***G***, as follows. Given ***c1* = (*gr, hr⋅x*)** and ***c2* =(*gs, hs⋅y*)**, observe that ***c1⋅c2*** (with multiplication of each element separately) equals **(*gr+s, hr+s⋅x⋅y*)**. In order to rerandomize, multiply again by **(*gw ,hw*)** for a random ***w***.

|  |  |
| --- | --- |
| Protocol Name: | Homomorphic multiplication for ElGamal |
| Protocol Reference: | ELGAMAL\_MULT |
| Protocol Type: | Homomorphic operation on commitments |
| Protocol Description: | A method for constructing a random ElGamal commitment to ***x*⋅*y*** (with multiplication in ***G***), given an ElGamal commitment to ***x*** and an ElGamal commitment to ***y*** (without knowing ***x*** or ***y***) |
| References: | None |

|  |  |
| --- | --- |
| ELGAMAL\_MULT Protocol Parameters | |
| Party’s Identity: | A single party P |
| Common parameters: | A DLOG group description (***G,q,g***) |
| Party’s Input: | Two ElGamal commitments ***c1* =(*h*,*u1,v1*)*, c2* =(*h,u2,v2*)**  OBSERVE: the same ***h*** value must appear in both |
| Party’s Output: | A single commitment ***c*** such that if ***c1*** is a commitment to ***x*** and ***c2*** is a commitment to ***y***, then ***c*** is a random commitment to ***x*⋅*y*** |

|  |  |
| --- | --- |
| ELGAMAL\_MULT Protocol Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g***), AND the same ***h*** value appears in ***c1,c2***,  REPORT ERROR and HALT  SAMPLE a random value ***w* ∈ *Zq***  COMPUTE ***u* = *gw*⋅*u1*⋅*u2***  COMPUTE ***v* = *hw*⋅*v1*⋅*v2***  OUTPUT ***c* = (*u,v*)** |

## UC-Secure Commitments

It seems that there is no really efficient UC-secure commitment protocol out there! This is very disconcerting, and also makes the transformation from Sigma protocols to UCZK irrelevant. So, this is an important question.

Actually, maybe this is OK, will look again…

Take <http://eprint.iacr.org/2001/091> as well as doing fully trapdoor commitment with UC-secure ZKPOKs for ElGamal and Pedersen ???

### UC-Secure Commitments with a Random Oracle (COMMIT\_UC\_ROM)

Use: Dennis Hofheinz, [Jörn Müller-Quade](http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/m/M=uuml=ller=Quade:J=ouml=rn.html): Universally Composable Commitments Using Random Oracles. [TCC 2004](http://www.informatik.uni-trier.de/~ley/db/conf/tcc/tcc2004.html#HofheinzM04): 58-76

### UC-Secure Commitments from DDH (COMMIT\_UC\_DDH)

## Non-Malleable Commitments

### Non-Malleable Hash Commitments (heuristic) (COMMIT\_NON\_MALLEABLE\_HASH\_HEUR)

|  |  |
| --- | --- |
| Protocol Name: | Non-Malleable Commitment via Hash (ROM) |
| Protocol Reference: | COMMIT\_NON\_MALLEABLE\_ HASH\_HEUR |
| Protocol Type: | A non-malleable commitment scheme |
| Protocol Description: | This is a commitment scheme that is non-malleable when viewing the hash function as a random oracle. However, this can actually be viewed as a standard assumption; no special random oracle properties are necessary. |
| References: | Folklore |
| The Protocol: | Run COMMIT\_HASH\_BASIC |

### Non-Malleable Commitments from Encryption (CRS) (COMMIT\_NON\_MALLEABLE\_CRS)

|  |  |
| --- | --- |
| Protocol Name: | Non-Malleable Commitment from Encryption (CRS) |
| Protocol Reference: | COMMIT\_NON\_MALLEABLE \_CRS |
| Protocol Type: | Non-malleable commitment |
| Protocol Description: | This is a non-malleable commitment scheme in the common reference string model. It uses any public-key encryption sceme that is NM-CPA (and so in particular CCA2 is fine). |
| References: | Folklore |

|  |  |
| --- | --- |
| COMMIT\_NON\_MALLEABLE\_CRS Protocol Parameters | |
| Let (G,E,D) be a public-key encryption scheme that is non-malleable under chosen plaintext attacks (NM-CPA); note CCA2-secure encryption is NM-CPA | |
| Party’s Identity: | Committer (C) and Receiver(R) |
| Common parameters: | A public key ***pk*** for the NM-CPA encryption scheme |
| Party’s Input: | C’s private input: a value ***x*** |
| Party’s Output: | * C: nothing * R’s output from the COMMIT phase:   + nothing * R’s output from the DECOMMIT phase:   + ACC or REJ, and if ACC then a value ***x* ∈{0*,* 1}*t*** |

|  |  |
| --- | --- |
| COMMIT\_NON\_MALLEABLE\_CRS Protocol Specification | |
| Commit phase | |
| Step 1 (C): | SAMPLE a random value ***r* ∈ {0*,* 1}*n***  COMPUTE ***c* = *Epk*(*x;r*)** (encrypt ***x*** with randomness ***r***)  SEND ***c*** to R |
| Step 2 (R): | WAIT for a value ***c***  STORE ***c*** |
| Decommit phase | |
| Step 1 (C): | SEND **(*x, r*)** to R |
| Step 2 (R): | IF **NOT** ***c* = *Epk*(*x;r*)**  OUTPUT REJ  ELSE  OUTPUT ACC and value ***x*** |

|  |  |
| --- | --- |
| COMMIT\_NON\_MALLEABLE\_CRS Committer (C) Specification | |
| Commit phase | |
| Step 1: | SAMPLE a random value ***r* ∈ {0*,* 1}*n***  COMPUTE ***c* = *Epk*(*x;r*)** (encrypt ***x*** with randomness ***r***)  SEND ***c*** to R |
| Decommit phase | |
| Step 1: | SEND **(*x, r*)** to R |
| Step 2: | OUTPUT nothing |

|  |  |
| --- | --- |
| COMMIT\_NON\_MALLEABLE\_CRS Receiver (R) Specification | |
| Commit phase | |
| Step 1: | WAIT for a value ***c***  STORE ***c*** |
| Decommit phase | |
| Step 1: | WAIT for **(*x, r*)** from C |
| Step 4: | IF **NOT** ***c* = *Epk*(*x;r*)**  OUTPUT REJ  ELSE  OUTPUT ACC and value ***x*** |

# Oblivious Transfers

|  |  |
| --- | --- |
| Protocol Name: | Naor-Pinkas |
| Protocol Reference: | OT\_DDH\_PRIVATE |
| Protocol Type: | Oblivious Transfer Protocol |
| Security Level: | Privacy only |
| Protocol Description: | Two-round oblivious transfer based on the DDH assumption |
| References: | Protocol 7.2.1 page 179 of Hazay-Lindell |

|  |  |
| --- | --- |
| 0T\_DDH\_PRIVATE Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q,g***) is a DLOG description * 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*** |

|  |  |
| --- | --- |
| 0T\_DDH\_PRIVATE Protocol Specification | |
| Step 1 (Both): | Both: IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT  R: SAMPLErandom values ***α,β,γ* ∈ {0*, . . . , q-1*}**  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 (cheat attempt)  SAMPLE random values ***u0,u1,v0,v1*** **∈ {0*, . . . , q-1*}**  COMPUTE:   * ***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** * ***c0* = *x0* XOR *KDF*(|*x0*|*,k0*)** * ***c1* = *x1* XOR *KDF*(|*x1*|*,k1*)**   SEND **(*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*(|*cσ*|,*kσ*)**  S: output nothing |
|  |  |

|  |  |
| --- | --- |
| 0T\_DDH\_PRIVACY 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 |

|  |  |
| --- | --- |
| 0T\_DDH\_PRIVACY 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*)** |

**WE SAID NOT TO DO THIS.**

|  |  |
| --- | --- |
| 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*** |
|  |  |

PVW plain using Fiat-Shamir

PVW\_UC based on DDH

PVW\_UC based on N-residuosity

# 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*** |

**WE SAID NOT TO DO THIS.**

|  |  |
| --- | --- |
| 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

|  |  |
| --- | --- |
| 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*** |

# Oblivious Polynomial Evaluation (OPE)

## OPE from OT

## OPE from Homomorphic Encryption

## Fully Secure OPE (???)

# Key Exchange Protocols

## InitKey Protocol (KEY\_EXCHANGE\_INITKEY)

|  |  |
| --- | --- |
| Protocol Name: | InitKey empty key exchange |
| Protocol Reference: | KEY\_EXCHANGE\_INITKEY |
| Protocol Type: | Key Exchange Protocol |
| Protocol Description: | This is an empty key-exchange protocol for the case that the long-term shared symmetric keys are used for encryption and MAC directly. |
| References: |  |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_INITKEY Protocol Parameters | |
| Parties’ Identities: | Parties P1 and P2 |
| Common parameters: | None |
| Parties’ Inputs: | Both parties have a pairs of symmetric keys ***K1,K2***, where ***K1*** is for encryption and ***K2*** is for message authentication |
| Parties’ Outputs: | * An encryption key ***K*ENC** * A MAC key ***K*MAC** |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_INITKEY Protocol Specification | |
| Step 1 (both): | OUTPUT ***K*ENC = *K1*** and ***K*MAC = *K2*** |

## Passive Diffie-Hellman (KEY\_EXCHANGE\_DH)

|  |  |
| --- | --- |
| Protocol Name: | Passive Diffie-Hellman key exchange |
| Protocol Reference: | KEY\_EXCHANGE\_DH |
| Protocol Type: | Key Exchange Protocol |
| Protocol Description: | This is the Diffie-Hellman key exchange protocol that is secure in the presence of eavesdropping adversaries only. |
| References: |  |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_DH Protocol Parameters | |
| Parties’ Identities: | Parties P1 and P2 |
| Common parameters: | A DLOG group description (***G,q,g***) and integers ***L*ENC** and ***L*MAC** which are the respective lengths of the encryption and MAC keys to be generated. |
| Parties’ Inputs: | None |
| Parties’ Outputs: | * An encryption key KENC * A MAC key KMAC |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_DH Protocol Specification | |
| Step 1 (P1): | SAMPLE a random ***a* ← *Zq***  COMPUTE ***h1* = *ga***  SEND ***h1*** to P2 |
| Step 2 (P2): | SAMPLE a random ***b* ← *Zq***  COMPUTE ***h2* = *gb***  SEND ***h2*** to P1 |
| Step 3 (P1): | WAIT for ***h2*** from P2  COMPUTE **(*K1 ,K2*) = KDF(*L*ENC+*L*MAC, (*h2*)*a*)**  OUTPUT ***K*ENC = *K1*** and ***K*MAC = *K2*** |
| Step 4 (P2): | WAIT for ***h1*** from P1  COMPUTE **(*K1 ,K2*) = KDF(*L*ENC+*L*MAC, (*h1*)*b*)**  OUTPUT ***K*ENC = *K1*** and ***K*MAC = *K2*** |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_DH Party P1 Specification | |
| Step 1: | SAMPLE a random ***a* ← *Zq***  COMPUTE ***h1* = *ga***  SEND ***h1*** to P2 |
| Step 2: | WAIT for ***h2*** from P2  COMPUTE **(*K1 ,K2*) = KDF(*L*ENC+*L*MAC, (*h2*)*a*)**  OUTPUT ***K*ENC = *K1*** and ***K*MAC = *K2*** |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_DH Party P2 Specification | |
| Step 1: | SAMPLE a random ***b* ← *Zq***  COMPUTE ***h2* = *gb***  SEND ***h2*** to P1 |
| Step 2: | WAIT for ***h1*** from P1  COMPUTE **(*K1 ,K2*) = KDF(*L*ENC+*L*MAC, (*h1*)*b*)**  OUTPUT ***K*ENC = *K1*** and ***K*MAC = *K2*** |

## UC-Secure Key Exchange (KEY\_EXCHANGE\_UCDH)

|  |  |
| --- | --- |
| Protocol Name: | UC-Secure Key Exchange |
| Protocol Reference: | KEY\_EXCHANGE\_UCDH |
| Protocol Type: | Key Exchange Protocol |
| Protocol Description: | This is a universally composable key exchange protocol that is secure under the DDH assumption in the ideal-signature hybrid model. |
| References: | This is the SIG-DH protocol in Canetti-Krawczyk, Eurocrypt 2001 |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_UCDH Protocol Parameters | |
| Parties’ Identities: | Parties P1 and P2 |
| Common parameters: | * Public keys **(*pk1*,*pk2*)** * A unique session identifier ***sid*** * A DLOG group description (***G,q,g***) and integers ***L*ENC** and ***L*MAC** which are the respective lengths of the encryption and MAC keys to be generated. |
| Parties’ Inputs: | * P1 has private key ***sk1*** (associated with ***pk1***) * P2 has private key ***sk2*** (associated with ***pk2***) |
| Parties’ Outputs: | * An encryption key KENC * A MAC key KMAC * The session identifier ***sid*** |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_UCDH Protocol Specification | |
| Step 1 (P1): | SAMPLE a random ***a* ← *Zq***  COMPUTE ***h1* = *ga***  SEND **(*P1*,*sid***,***h1*)** to P2 |
| Step 2 (P2): | WAIT for **(*P1*,*sid***,***h1*)** from P1  SAMPLE a random ***b* ← *Zq***  COMPUTE ***h2* = *gb***  COMPUTE **s =** **Sign(*sk2*,(*P2,sid,h2,h1,P1*))**  COMPUTE ***h3* = *h1b***  ERASE ***b***  SEND **(*P2*,*sid*,*h2,s*)** to P1 |
| Step 3 (P1): | WAIT for **(*P2*,*sid*,*h2,s*)** from P2  IF **Verify(pk2**,(***P2*,*sid*,*h2*,*h1,P1*),*s*) = false** [Verify(key,message,signature)]  OUTPUT FAIL (cheat attempt)  COMPUTE **s’ =** **Sign(*sk1*,(*P1,sid,h1,h2,P2*))**  COMPUTE ***h3* = *h2a***  ERASE ***a***  COMPUTE **(*K1 ,K2*) = KDF(*L*ENC+*L*MAC, *h3*)**  SEND **(*P1*,sid, *s’*)**  OUTPUT ***sid*, *K*ENC = *K1*** and ***K*MAC = *K2*** |
| Step 4 (P2): | WAIT for **(*P1*,*sid,s’*)** from P1  IF **Verify(pk1**,(***P1*,*sid*,*h1*,*h1*,*P2*),*s’*) = false** [Verify(key,message,signature)]  OUTPUT FAIL (cheat attempt)  COMPUTE **(*K1 ,K2*) = KDF(*L*ENC+*L*MAC, *h3*)**  OUTPUT ***sid, K*ENC = *K1*** and ***K*MAC = *K2*** |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_UCDH Party P1 Specification | |
| Step 1: | SAMPLE a random ***a* ← *Zq***  COMPUTE ***h1* = *ga***  SEND **(*P1*,*sid***,***h1*)** to P2 |
| Step 2: | WAIT for **(*P2*,*sid*,*h2,s*)** from P2  IF **Verify(pk2**,(***P2*,*sid*,*h2*,*h1,P1*),*s*) = false** [Verify(key,message,signature)]  OUTPUT FAIL (cheat attempt)  COMPUTE **s’ =** **Sign(*sk1*,(*P1,sid,h1,h2,P2*))**  COMPUTE ***h3* = *h2a***  ERASE ***a***  COMPUTE **(*K1 ,K2*) = KDF(*L*ENC+*L*MAC, *h3*)**  SEND **(P1,sid, *s’*)**  OUTPUT ***sid*, *K*ENC = *K1*** and ***K*MAC = *K2*** |

|  |  |
| --- | --- |
| KEY\_EXCHANGE\_UCDH Party P2 Specification | |
| Step 1: | WAIT for **(*P1*,*sid***,***h1*)** from P1  SAMPLE a random ***b* ← *Zq***  COMPUTE ***h2* = *gb***  COMPUTE **s =** **Sign(*sk2*,(*P2*,*sid*,*h2*,*h1*,*P1*))**  COMPUTE ***h3* = *h1b***  ERASE ***b***  SEND **(*P2*,*sid*,*h2,s*)** to P1 |
| Step 2: | WAIT for **(*P1*,*sid,s’*)** from P1  IF **Verify(pk1**,(***P1*,*sid*,*h1*,*h1*,*P2*),*s’*) = false** [Verify(key,message,signature)]  OUTPUT FAIL (cheat attempt)  COMPUTE **(*K1 ,K2*) = KDF(*L*ENC+*L*MAC, *h3*)**  OUTPUT ***sid, K*ENC = *K1*** and ***K*MAC = *K2*** |

Note: In the above Pseudocode specification, values ***P1*** and ***P2*** are used. These are identifiers of the parties (the IP address suffices).

# Secure Channel

## Basic Secure Channel

Just encrypt and authenticate using any secure encryption and mac. Actually, also need counters for replay…

## Adaptive Secure Channel (with Erasures)

Authentication is like usual. We only change the encryption.

Based on Beaver-Haber (but without generating the entire pad ahead of time). Let k be the key. Then, the first message is sent by computing (k1,k2)=PRG(k,L+128) where L is the length of the message needed, k1 is of length L, and k2 is of length 128 bits. Then, encrypt the message with k1 (XOR), ERASE k1, and set k=k2 (for the next message) erasing the original k. Note that if a message is lost, then it cannot be resent (how do we solve this?) and everything must be completely synchronized. We can solve the lost message problem by sending

# Special Encryption Schemes

Find and take existing implementations; check if possible

## Paillier Encryption and Homomorphic Operations

## Attribute-Based Encryption

## Format-Preserving Encryption

## Order-Preserving Encryption

## Identity-Based Encryption

# Non-Interactive Primitives

We include pseudocode here of primitives that are not found in standard industry cryptography libraries.

## Get Random

|  |  |
| --- | --- |
| Protocol Name: | Get Random |
| Protocol Reference: | GET\_RANDOM |
| Protocol Type: | Method for obtaining random bits |
| Protocol Description: |  |
| References: |  |

|  |  |
| --- | --- |
| HKDF Protocol Parameters | |
| Parties’ Identities: | Party P1 (this is a non-interactive function) |
| Common parameters: | * A method for obtaining random bits from the operating system * A SecureRandom method provided by the programming language |
| Parties’ Inputs: | * A parameter ***L*** determining how many bits to obtain |
| Parties’ Outputs: | * A string ***R*** of length ***L*** |

|  |  |
| --- | --- |
| HKDF Protocol Specification | |
| Step 1: | CALL SecureRandom() to obtain L bits of randomness; denote the output **R1**  CALL random generator of operating system; denote the output **R2**  Examples from Windows operating systems:   * In Windows XP, use CAPI and the CryptGenRandom() function * In Windows Vista and above, use CNG and BCryptGenRandom()   OUTPUT **R1 ⊕ R2** |

## Pseudorandom Function with Arbitrary Input-Output Lengths

We take the interpretation that there is essentially a different random function for every output length. This can be modeled by applying the random function to the input and the required output length (given as input to the oracle). The pseudorandom function must then be indistinguishable from this.

Define ***y* = *F*K(*x*,*outlen*) = HMACK(*x,outlen*,1),…,HMACK(*x,outlen,m*)** where ***m*** is the smallest integer for which ***L⋅m*** **> *outlen***, where ***L*** is the output length of HMAC. Then, output ***y*** truncated to length ***outlen***. Observe that in terms of input length this is fine because HMAC works for any input length already.

## Pseudorandom Permutation with Arbitrary Input-Output Lengths

Apply 4 rounds of Luby-Rackoff to a PRF with output length half of what is desired. Only accept output lengths of even size.

See also format-preserving encryption…

## Universal One-Way Hashing

## Perfect Universal Hash Functions

## Information-Theoretic MAC

## Key Derivation (HKDF)

|  |  |
| --- | --- |
| Protocol Name: | Hash Key Derivation |
| Protocol Reference: | HKDF |
| Protocol Type: | Key derivation function |
| Protocol Description: | This is a key derivation function that has a rigorous justification as to its security |
| References: | H. Krawczyk. Cryptographic Extraction and Key Derivation:  The HKDF Scheme. CRYPTO 2010. |

|  |  |
| --- | --- |
| HKDF Protocol Parameters | |
| Parties’ Identities: | Party P1 (this is a non-interactive function) |
| Common parameters: | * A concrete hash function * A constant, hardwired random value ***XTS*** of length that equals the output of the hash function |
| Parties’ Inputs: | * An input string, denoted ***SKM*** (source key material) * An optional string called ***CTXinfo***, that determines the context of the key derivation; if not given, this is null * An integer ***L*** denoting the desired length of output |
| Parties’ Outputs: | * A string ***K*** of length ***L*** |

|  |  |
| --- | --- |
| HKDF Protocol Specification | |
| Step 1: | COMPUTE ***PRK* = HMAC(*XTS, SKM*)** [key=XTS, data=SKM]  Let ***t*** be the smallest number so that ***t⋅*|*H*|>*L*** where **|*H*|** is the HMAC output length  ***K*(*1*) = HMAC(*PRK*,(*CTXinfo*,*0*))** [key=PRK, data=(CTXinfo,0)]  **FOR *i* = *2* TO *t***  ***K*(*i*) = HMAC(*PRK*,(*K*(*i-1*),*CTXinfo,i*))** [key=PRK, data=(K(i-1),CTXinfo,i)]  OUTPUT the first ***L*** bits from ***K*(*1*),…,*K*(*t*)** |