SCAPI Pseudocode Specification

**Cryptography and Computer Security Research Group**

**Department of Computer Science**

**Bar-Ilan University**

**11/16/2010**

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 | 15/11/10 | Yehuda Lindell | Overall review, additions and corrections |
| 1.2 | 02/05/12 | Yehuda Lindell | Additions, changes and review |
|  |  |  |  |

Table of Contents

[2 Template and Document Explanation 5](#_Toc277574564)

[3 Sigma protocols 8](#_Toc277574565)

[3.1 Schnorr’s Sigma-Protocol for DLOG (SIGMA\_DLOG) 8](#_Toc277574566)

[3.2 Sigma-Protocol for Diffie-Hellman Tuples (SIGMA\_DH) 10](#_Toc277574567)

[3.3 Sigma Protocol for Pedersen Commitment Knowledge (SIGMA\_PEDERSEN) 11](#_Toc277574568)

[3.4 Sigma-Protocol that a Pedersen-Committed Value is x (SIGMA\_COMMITTED\_VALUE\_PEDERSEN) 12](#_Toc277574569)

[3.5 Sigma Protocol for El Gamal Commitment Knowledge (SIGMA\_ELGAMAL\_COMMIT) 13](#_Toc277574570)

[3.6 Sigma Protocol that an ElGamal-Committed Value is x (SIGMA\_COMMITTED\_VALUE\_ELGAMAL) 14](#_Toc277574571)

[3.7 Sigma Protocol of ElGamal Secret Key (SIGMA\_SK\_ELGAMAL) 15](#_Toc277574572)

[3.8 Sigma Protocol that ElGamal-Encrypted Value is x (SIGMA\_ENCRYPTED\_VALUE\_ELGAMAL) 16](#_Toc277574573)

[3.8.1 Version 1 – using knowledge of the secret key 16](#_Toc277574574)

[3.8.2 Version 2 – using knowledge of the randomness used to encrypt 17](#_Toc277574575)

[3.9 Sigma Protocol – AND of Multiple Statements (AND\_SIGMA) 18](#_Toc277574576)

[3.10 Sigma Protocol – OR of 2 Statements (OR\_2\_SIGMA) 19](#_Toc277574577)

[3.11 Sigma Protocol – OR of Multiple Statements (OR\_MANY\_SIGMA) [TBD: BENNY] 20](#_Toc277574578)

[3.12 Sigma Protocols for Paillier (Damgard-Jurik Version) [TBD: BENNY] 17](#_Toc277574579)

[4 Zero-knowledge 21](#_Toc277574580)

[4.1 Zero-Knowledge from any Sigma Protocol (ZK\_FROM\_SIGMA) 21](#_Toc277574581)

[4.2 Zero-Knowledge Proof of Knowledge from any Sigma Protocol (ZKPOK\_FROM\_SIGMA) 23](#_Toc277574582)

[4.3 ZKPOK from any Sigma-Protocol – ROM (Fiat-Shamir) (ZKPOK\_FS\_SIGMA) 25](#_Toc277574583)

[4.4 UC-Secure ZKPOK from any Sigma-Protocol (UCZKPOK\_FROM\_SIGMA) 27](#_Toc277574584)

[5 Commitment schemes 29](#_Toc277574585)

[5.1 Pedersen Commitment (COMMIT\_PEDERSEN) 29](#_Toc277574586)

[5.2 Pedersen-Hash Commitment (COMMIT\_HASH\_PEDERSEN) 31](#_Toc277574587)

[5.3 ElGamal Commitment(COMMIT\_ELGAMAL) 32](#_Toc277574588)

[5.4 ElGamal-Hash Commitment (COMMIT\_HASH\_ELGAMAL) 34](#_Toc277574589)

[5.5 Hash-Based Commitment (Basic) (COMMIT\_HASH\_BASIC) 35](#_Toc277574590)

[5.6 Hash-Based Statistically-Hiding Commitment (COMMIT\_HASH) 37](#_Toc277574591)

[5.7 Equivocal Commitments (COMMIT\_EQUIVOCAL) 39](#_Toc277574592)

[5.8 Extractable Commitments(COMMIT\_EXTRACT) 40](#_Toc277574593)

[5.9 Fully Trapdoor Commitments 42](#_Toc277574594)

[5.9.1 Fully Trapdoor using Sigma Protocols(COMMIT\_DOUBLE\_TRAPDOOR) 42](#_Toc277574595)

[5.9.2 Fully Trapdoor Hash (ROM) (COMMIT\_DOUBLE\_TRAPDOOR\_HASH\_ROM) 44](#_Toc277574596)

[5.9.3 Fully Trapdoor in the CRS Model (COMMIT\_DOUBLE\_TRAPDOOR\_DDH\_CRS) 44](#_Toc277574597)

[5.10 Additive Homomorphic Operation on Pedersen Commitments (PEDERSEN\_ADD) 45](#_Toc277574598)

[5.11 Multiplicative Homomorphic Operation on ElGamal Commitments (ELGAMAL\_MULT) 46](#_Toc277574599)

[5.12 UC-Secure Commitments [TBD: YEHUDA] 47](#_Toc277574600)

[5.12.1 UC-Secure Commitments with a Random Oracle (COMMIT\_UC\_ROM) 47](#_Toc277574601)

[5.12.2 UC-Secure Commitments from DDH (COMMIT\_UC\_DDH) 47](#_Toc277574602)

[5.13 Non-Malleable Commitments 48](#_Toc277574603)

[5.13.1 Non-Malleable Hash Commitments (heuristic) (COMMIT\_NON\_MALLEABLE\_HASH\_HEUR) 48](#_Toc277574604)

[5.13.2 Non-Malleable Commitments from Encryption (CRS) (COMMIT\_NON\_MALLEABLE\_CRS) 49](#_Toc277574605)

[6 Oblivious Transfer 51](#_Toc277574606)

[6.1 Private OT (Naor-Pinkas) (OT\_DDH\_PRIVATE) 51](#_Toc277574607)

[6.2 Oblivious Transfer with One-Sided Simulation (OT\_DDH\_ONESIDEDSIM) 53](#_Toc277574608)

[6.3 Oblivious Transfer with Full Simulation (OT\_DDH\_FULLSIM) 56](#_Toc277574609)

[6.4 Oblivious Transfer with Full Simulation – ROM (OT\_DDH\_FULLSIM\_ROM) 58](#_Toc277574610)

[6.5 Oblivious Transfer with UC Security – DDH (OT\_DDH\_UC\_PVW) 60](#_Toc277574611)

[6.6 Oblivious Transfer with UC Security – N-Residuosity (OT\_NRES\_UC\_PVW) 62](#_Toc277574612)

[7 Batch Oblivious Transfer 64](#_Toc277574613)

[7.1 Batch OT – Full Simulation from DDH (OT\_DDH\_FULLSIM\_BATCH) 64](#_Toc277574614)

[8 Coin Tossing 67](#_Toc277574615)

[8.1 Coin-Tossing of a Single Bit (COIN\_TOSSING\_BLUM) 67](#_Toc277574616)

[8.2 Coin-Tossing of a String (COIN\_TOSSING\_STRING) 69](#_Toc277574617)

[8.3 Semi-Simulatable Coin-Tossing of a String (COIN\_TOSSING\_SEMI) 71](#_Toc277574618)

[9 Secure Pseudorandom Function Evaluation [TO BE EDITED] 73](#_Toc277574619)

[10 Oblivious Polynomial Evaluation (OPE) 77](#_Toc277574620)

[10.1 OPE from OT [TBD: BENNY] 77](#_Toc277574621)

[10.2 OPE from Homomorphic Encryption [TBD: BENNY] 77](#_Toc277574622)

[10.3 Fully Secure OPE [TBD: YEHUDA] 77](#_Toc277574623)

[11 Key Exchange Protocols 78](#_Toc277574624)

[11.1 InitKey Protocol (KEY\_EXCHANGE\_INITKEY) 78](#_Toc277574625)

[11.2 Passive Diffie-Hellman (KEY\_EXCHANGE\_DH) 79](#_Toc277574626)

[11.3 UC-Secure Key Exchange (KEY\_EXCHANGE\_UCDH) 81](#_Toc277574627)

[12 Secure Channel 83](#_Toc277574628)

[12.1 Basic Secure Channel 83](#_Toc277574629)

[12.2 Adaptive Secure Channel (with Erasures) [TBD: YEHUDA] 83](#_Toc277574630)

[13 Special Encryption Schemes 83](#_Toc277574631)

[13.1 Paillier Encryption and Homomorphic Operations [TBD: BENNY, including Sigma protocols for Damgard-Jurik] 83](#_Toc277574632)

[13.2 Attribute-Based Encryption [TBD: MATT and AVI] 83](#_Toc277574633)

[13.3 Identity-Based Encryption [TBD: MATT and AVI] 88](#_Toc277574634)

[14 Non-Interactive Primitives 89](#_Toc277574635)

[14.1 Get Random 89](#_Toc277574636)

[14.2 Pseudorandom Function with Arbitrary Input-Output Lengths 90](#_Toc277574637)

[14.3 Pseudorandom Permutation with Arbitrary Input-Output Lengths 92](#_Toc277574638)

[14.4 Universal One-Way Hashing 92](#_Toc277574639)

[14.5 Perfect Universal Hash Functions 93](#_Toc277574640)

[14.6 Information-Theoretic MAC 94](#_Toc277574641)

[14.7 Key Derivation (HKDF) 95](#_Toc277574642)

# 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

Sigma protocols are a basic building block for zero-knowledge, zero-knowledge proofs of knowledge and more. A sigma protocol is a 3-round proof, comprised of a first message from the prover to the verifier, a random challenge from the verifier and a second message from the prover. See Hazay-Lindell (chapter 6) for more information.

We begin by describing Sigma protocols for a number of tasks. Our description includes the specification of the 3 messages, and the verification check of V. We also include the simulator description since this is used in some constructions. Later, we show the automatic transformations from Sigma protocols to zero-knowledge and so on.

## 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 | |
| Prover message 1 (a): | SAMPLE a random ***r*∈ *Zq*** and COMPUTE ***a* = *gr*** |
| Verifier challenge (e): | SAMPLE a random challenge ***e* ∈{0, 1}*t*** |
| Prover message 2 (z): | COMPUTE ***z* = *r* + *ew* mod *q*** |
| Verifier check: | ACC IFF VALID\_PARAMS(***G,q,g***)=TRUE AND ***h*** ∈ ***G*** AND ***gz* = *ahe*** |

We write the actual messages in parentheses to make this explicit. In the above case, the messages are ***a,e,*** and ***z***.

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

**IMPORTANT NOTE**: In this and all the coming simulators, if ***e*** is not given as input then it should be chosen uniformly at random.

## 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-Hellman 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 | |
| Prover message 1 (a,b): | SAMPLE a random ***r*∈ *Zq*** and COMPUTE ***a* = *gr*** and ***b* = *hr*** |
| Verifier challenge (e): | SAMPLE a random challenge ***e* ∈{0, 1}*t*** |
| Prover message 2 (z): | COMPUTE ***z* = *r* + *ew* mod *q*** |
| Verifier check: | ACC IFF VALID\_PARAMS(***G,q,g***)=TRUE AND ***h*** ∈ ***G*** AND ***gz* = *aue*** AND ***hz* = *bve*** |

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

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

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol for Extended Diffie-Hellman Tuples |
| Protocol Reference: | SIGMA\_EXTEND\_DH |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a prover to convince a verifier that the input tuple **(*g1,…,gm,h1,…,hm)*** is an ***extended*** Diffie-Hellman tuple, meaning that there exists a single ***w*∈ *Zq*** such that ***hi*=*giw*** for all ***i***. |
| References: | Straightforward extension from SIGMA\_DH |

|  |  |
| --- | --- |
| SIGMA\_EXTEND\_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: **(*g1,…,gm,h1,…,hm)***and ***t*** such that ***2t* < *q*** * P’s private input: a value ***w*∈ *Zq*** such that ***hi*=*giw*** for all ***i*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_EXTEND\_DH Protocol Specification | |
| Prover message 1 (a): | SAMPLE a random ***r*∈ *Zq*** and COMPUTE ***ai* = *gir*** for all ***i***  SET ***a=(a1,…,am)***. |
| Verifier challenge (e): | SAMPLE a random challenge ***e* ∈{0, 1}*t*** |
| Prover message 2 (z): | COMPUTE ***z* = *r* + *ew* mod *q*** |
| Verifier check: | ACC IFF VALID\_PARAMS(***G,q,g***)=TRUE AND all ***g1,…,gm*** ∈ ***G*** AND for all ***i=1,…,m*** it holds that ***giz* = *ai⋅hie*** |

|  |  |
| --- | --- |
| SIGMA\_EXTEND\_DH Simulator (*M*) Specification | |
| Input: | Parameters (***G,q,g***) and ***t***, input **(*h,u,v*)** and a challenge ***e* ∈{0, 1}*t*** |
| Computation: | SAMPLE a random ***z* ∈ *Zq***  For every ***i=1,…,m***, COMPUTE ***ai = giz⋅hi-e*** (where ***–e*** here means ***–e* mod *q***)  OUTPUT **(**(***a1,…,am***)***,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 ‎5.1 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 | |
| Prover message 1 (a): | SAMPLE random values ***α* ∈ *Zq*** and ***β* ∈ *Zq*** and COMPUTE ***a* =*****hα⋅ gβ*** |
| Verifier challenge (e): | SAMPLE a random challenge ***e* ∈{0, 1}*t*** |
| Prover message 2 (z): | COMPUTE ***u* = *α* + *ex*****mod *q***and ***v* = *β* + *er*****mod *q*** |
| Verifier check: | ACC IFF VALID\_PARAMS(***G,q,g***)=TRUE AND ***h*** ∈ ***G*** AND ***hu⋅gv=a⋅ce*** |

|  |  |
| --- | --- |
| SIGMA\_PEDERSEN Simulator (*M*) Specification | |
| Input: | Parameters (***G,q,g***) and ***t***, input **(*h,c*)** and a challenge ***e* ∈{0, 1}*t*** |
| Computation: | SAMPLE random values ***u* ∈ *Zq*** and ***v* ∈ *Zq***  COMPUTE ***a* = *hu⋅gv⋅c-e*** (where ***–e*** here means ***–e* mod *q***)  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 ‎5.1 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 | |
| Specification: | 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 ‎3.11): 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 | |
| Specification: | 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 ‎3.11): 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 | |
| Specification: | 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 ‎3.11): 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 | |
| Specification: | 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 ‎3.11): 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 | |
| Specification: | 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 ‎3.11): same as SIGMA\_DH 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 | |
| Specification: | 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 ‎3.11): same as SIGMA\_DH with inputs appropriately defined.

## Sigma Protocol that Cramer-Shoup-Encrypted Value is x (SIGMA\_ENCRYPTED\_VALUE\_CRAMERSHOUP)

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 a Cramer-Shoup Encrypted Value is x |
| Protocol Reference: | SIGMA\_ENCRYPTED\_VALUE\_CRAMERSHOUP |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used to prove that the value encrypted under Cramer-Shoup in the ciphertext **(*u1,u2,e,v*)** with public-key ***g1,g2,c,d,h*** is ***x***. The protocol is for the case that the prover knows the randomness used to encrypt |
| References: | None: this is actually an EXTEND\_DH Sigma Protocol |

|  |  |
| --- | --- |
| SIGMA\_ENCRYPTED\_VALUE\_CRAMERSHOUP 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: **(*u1,u2,e,v*), *g1,g2,c,d,h*** and ***x*** * P’s private input: a value ***r*∈ *Zq*** such that ***u1*=*g1r***, ***u2*=*g2r***, ***e*=*hr⋅x*** and ***v* = *(cdH(u1,u2,e))r*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_ENCRYPTED\_VALUE\_CRAMERSHOUP Protocol Specification | |
| Specification: | RUN SIGMA\_EXTEND\_DH with:   * Common parameters **(*G,q,g*)** * Common input: **(*g1,g2,g3,g4,h1,h2,h3,h4*) = (*g1,g2,h,cdw,u1,u2,e/x,cdw*)** * P’s private input: a value ***w*∈ *Zq*** such that ***hi=giw*** for all ***i*** |

**Sigma-Protocol Simulator** (see Section ‎3.11): same as SIGMA\_EXTEND\_DH with inputs appropriately defined.

## Sigma Protocols that a Damgard-Jurik Encrypted Value is 0 (SIGMA\_ZERO\_DAMGARD\_JURIK)

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol that a Damgard-Jurik encrypted value is 0 |
| Protocol Reference: | SIGMA\_ZERO\_DAMGARD\_JURIK |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a party to prove that a ciphertext is an encryption of 0 (or an ***N***th power) |
| References: | Damgard-Jurik |

|  |  |
| --- | --- |
| SIGMA\_ZERO\_DAMGARD\_JURIK Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common Parameters: | Length parameter ***s*** and soundness parameter ***t < |n|/3*** (i.e., ***t*** must be less than a third of the length of the public key **n**) |
| Parties’ Inputs: | * Common input statement: public key ***n*** and ciphertext ***c*** * P’s private input: a value ***r*∈ *Z\*N’*** such that ***c=rN mod N’*** * Note ***N=ns*** and ***N’=Ns+1*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_ ZERO\_DAMGARD\_JURIK Protocol Specification | |
| Prover message 1 (a): | SAMPLE a random value ***s* ∈ *Z\*n*** and COMPUTE ***a* =*****sN mod N’*** |
| Verifier challenge (e): | SAMPLE a random challenge ***e* ∈{0, 1}*t*** |
| Prover message 2 (z): | COMPUTE ***z* = *s⋅re*****mod *n*** |
| Verifier check: | ACC IFF ***c,a,z*** are relatively prime to ***n*** AND AND ***zN = a⋅ce mod N’*** |

|  |  |
| --- | --- |
| SIGMA\_ ZERO\_DAMGARD\_JURIK Simulator (*M*) Specification | |
| Input: | Parameter ***t***, input **(*n,c*)** and a challenge ***e* ∈{0, 1}*t*** |
| Computation: | SAMPLE a random value ***z* ∈ *Z\*n***  COMPUTE ***a* = *zN/ce mod N’***  OUTPUT **(*a,e,z*)** |

NOTE: This protocol assumes that the prover knows the randomness used to encrypt. If the prover knows the secret key, then it can compute (once) the value ***m=n-1 mod ∅(n)=n-1 mod (p-1)(q-1)***. Then, it can recover the randomness ***r*** from ***c*** by computing ***cm mod n*** (this equals ***rn/n mod n = r***). Once given ***r***, the prover can proceed with the above protocol.

## Sigma Protocols that a Damgard-Jurik Encrypted Value is x (SIGMA\_ENCRYPTED\_VALUE\_DAMGARD\_JURIK)

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol that a Damgard-Jurik encrypted value is x |
| Protocol Reference: | SIGMA\_ENCRYPTED\_VALUE\_DAMGARD\_JURIK |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a party who encrypted a value ***x*** to prove that it indeed encrypted ***x*** |
| References: | Damgard-Jurik paper |

|  |  |
| --- | --- |
| SIGMA\_ENCRYPTED\_VALUE\_DAMGARD\_JURIK Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common parameters: | Length parameter ***s*** and soundness parameter ***t < |n|/3*** |
| Parties’ Inputs: | * Common input statement: public key ***n***, ciphertext ***c***, plaintext ***x*** * P’s private input: a value ***r*∈ *Z\*N’*** such that ***c=(1+n)xrN mod N’*** * Note ***N=ns*** and ***N’=Ns+1*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_ENCRYPTED\_VALUE\_DAMGARD\_JURIK Protocol Specification | |
| Specification: | RUN SIGMA\_ZERO\_DAMGARD\_JURIK with:   * Common input: **(*n,c’)*** where ***c’=c⋅(1+n*)-x** * P’s private input: a value ***r*∈ *Zq*** such that ***c’=rN mod N’*** |

**Sigma-Protocol Simulator**: same as SIGMA\_ZERO\_DAMGARD\_JURIK with inputs appropriately defined.

## Sigma Protocols that 3 Damgard-Jurik Ciphertexts are a Product (SIGMA\_PRODUCT\_DAMGARD\_JURIK)

|  |  |
| --- | --- |
| Protocol Name: | Σ Protocol that 3 Damgard-Jurik ciphertexts are a product |
| Protocol Reference: | SIGMA\_PRODUCT\_DAMGARD\_JURIK |
| Protocol Type: | Sigma Protocol |
| Protocol Description: | This protocol is used for a party to prove that 3 ciphertexts ***c1,c2,c3*** are encryptions of values **x1,x2,x3** s.t. ***x1⋅x2=x3 mod N*** |
| References: | Damgard-Jurik |

|  |  |
| --- | --- |
| SIGMA\_ZERO\_DAMGARD\_JURIK Protocol Parameters | |
| Parties’ Identities: | Prover (P) and Verifier (V) |
| Common Parameters: | Length parameter ***s*** and soundness parameter ***t < |n|/3*** |
| Parties’ Inputs: | * Common input statement: public key ***n*** and ciphertexts ***c1,c2,c3*** * P’s private input: value ***r1,r2,r3*∈ *Z\*N’*** s.t. ***ci=(1+n)xi⋅riN mod N’*** * Note ***N=ns*** and ***N’=Ns+1*** |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |
| --- | --- |
| SIGMA\_ ZERO\_DAMGARD\_JURIK Protocol Specification | |
| Prover message 1 (a): | SAMPLE random values ***d* ∈ *ZN***, ***rd* ∈ *Z\*n***, ***rdb* ∈ *Z\*n***, COMPUTE ***a1=(1+n)drdN mod N’*** and ***a2=(1+n)d⋅x2rdbN mod N’*** and SET ***a* =*****(a1,a2)*** |
| Verifier challenge (e): | SAMPLE a random challenge ***e* ∈{0, 1}*t*** |
| Prover message 2 (z): | COMPUTE ***z1=e⋅x1+d mod N***, ***z2 = r1e⋅rd mod n***, ***z3=r2z1/(rdb⋅r3e) mod n***, and SET ***z=(z1,z2,z3)*** |
| Verifier check: | ACC IFF ***c1,c2,c3,a1,a2,z1,z2,z3*** are relatively prime to ***n*** AND ***c1e⋅a1 = (1+n)z1z2N mod N’*** AND ***c2z1/(a2⋅c3e)= z3N mod N’*** |

|  |  |
| --- | --- |
| SIGMA\_ ZERO\_DAMGARD\_JURIK Simulator (*M*) Specification | |
| Input: | Parameter ***s***, input **(*n,c1,c2,c3*)** and a challenge ***e* ∈{0, 1}*t*** |
| Computation: | SAMPLE random values ***z1*∈*ZN, z2*∈*Z\*n, z3*∈*Z\*n***  COMPUTE ***a1* = *(1+n)z1z2N*/c1e *mod N’*** AND ***a2 = c2z1/(z3N⋅c3e) mod N’***  OUTPUT **(*a,e,z*)** where ***a* =*****(a1,a2)*** AND ***z=(z1,z2,z3)*** |

NOTE: This protocol assumes that the prover knows the randomness used to encrypt. If the prover knows the secret key, then it can compute (once) the value ***m=n-1 mod ∅(n)=n-1 mod (p-1)(q-1)***. Then, it can recover the randomness ***r*** from ***c*** by computing ***cm mod n*** (this equals ***rn/n mod n = r***). Once given ***r***, the prover can proceed with the above protocol.

## 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 | |
| Prover message 1 (*a1,…,am*): | COMPUTE all first prover messages ***a1,…,am*** |
| Verifier challenge (e): | SAMPLE a single random challenge ***e* ∈{0, 1}*t*** |
| Prover message 2 (*z1,…,zm*): | COMPUTE all second prover messages ***z1,…,zm*** |
| Verifier check: | ACC IFF all verifier checks are ACC |

|  |  |
| --- | --- |
| AND\_SIGMA Simulator (*M*) Specification | |
| Input: | A series of ***m*** statements {***xi***} and a challenge ***e* ∈{0, 1}*t*** |
| Computation: | RUN each Sigma protocol simulator with ***e*** |

NOTE: The result of this transformation on Sigma protocols is a Sigma protocol.

## 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*** (***i*=0,1**) | |
| Prover message 1 (*a1,a2*): | | 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*)**  The message is **(a1,a2)**; ***e1-b,z1-b*** are stored for later |
| Verifier challenge (e): | | SAMPLE a single random challenge ***e* ∈{0, 1}*t*** |
| Prover message 2 (*e0,z0,e1,z1*): | | SET ***eb = e* ⊕ *e1−b***  COMPUTE the response ***zb***to **(*ab, eb*)** in **π*b***using input **(*xb,w*)**  The message is ***e0,z0,e1,z1*** |
| Verifier check: | | ACC IFF all verifier checks are ACC |

|  |  |
| --- | --- |
| OR\_2\_SIGMA Simulator (*M*) Specification | |
| Input: | A pair of statements ***x0***, ***x1*** and a challenge ***e* ∈{0, 1}*t*** |
| Computation: | SAMPLE a random ***e0***, COMPUTE ***e1 = e* ⊕ *e0*** and then run the Sigma protocol simulator for each protocol with the resulting ***e0,e1*** values. |

NOTE: The result of this transformation on Sigma protocols is a Sigma protocol.

## 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] |

|  |  |
| --- | --- |
| OR\_MANY\_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***  A field **GF[2t]** (let **1,…,n** be well defined elements of the field) |
| Parties’ Inputs: | * Common input: **(*x1,…, xn*)** * P’s private input: a set of ***k*** witnesses ***wi*** (***w***i is a witness that can be used to prove ***xi***) |
| Parties’ Outputs: | * P: nothing * V: ACC or REJ |

|  |  |  |
| --- | --- | --- |
| OR\_MANY\_SIGMA Protocol Specification | | |
|  | Let **(*ai,ei,zi*)** denote the steps of a Σ protocol **π*i*** for proving that ***xi* ∈ *LRi***  Let ***I*** denote the set of indices for which P has witnesses | |
| Prover message 1 (*a*): | | For every **j∉I**, SAMPLE a random element **ej∈GF[2t]**  For every **j∉I**, RUN the simulator on statement **xj** and challenge **ej** to get transcript **(aj,ej,zj)**  For every **i∈I**, RUN the prover P on statement **xi** to get first message **ai**  SET **a=(a1,…,an)** |
| Verifier challenge (e): | | WAIT for messages **a1,…,an**  SAMPLE a single random challenge ***e* ∈GF[2t]** |
| Prover msg 2 (Q,*e1,z1,…,en,zn*): | | INTERPOLATE the points **(0,e)** and **{(j,ej)}** for every **j∉I** to obtain a degree ***n-k*** polynomial **Q** (s.t. **Q(0)=e** and **Q(j)=ej** for every **j∉I**)  For every **i∈I**, SET ***ei = Q(i)***  For every **i∈I**, COMPUTE the response ***zi***to **(*ai, ei*)** in **π*i***using input **(*xi,wi*)**  The message is ***Q,e1,z1,…,en,zn*** (where by ***Q*** we mean its coefficients) |
| Verifier check: | | ACC IFF ***Q*** is of degree ***n-k*** AND **Q(i)=ei** for all **i=1,…,n** AND **Q(0)=e**, and the verifier output on **(ai,ei,zi)** for all **i=1,…,n** is ACC |

|  |  |
| --- | --- |
| OR\_MANY\_SIGMA Simulator (*M*) Specification | |
| Input: | A series of ***n*** statements {***xi***} and a challenge ***e* ∈GF[2t]** |
| Computation: | SAMPLE random points ***e1***,…,**en-k** **∈GF[2t].**  COMPUTE the polynomial ***Q*** and values **en-k,…,en** like in the protocol.  RUN the simulator on each statement/challenge pair **(xi,ei)** for all **i=1,…,n** to obtain **(ai,ei,zi)**.  OUTPUT **(a1,e1,z1),…, (an,en,zn)** |

NOTE: The result of this transformation on Sigma protocols is a Sigma protocol.

# 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 | |
| Let (*a*,*e,z*) denote the prover1, verifier challenge and prover2 messages of the Σ protocol π | |
| 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 prover 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. The best choices for this are the COMMIT\_HASH, COMMIT\_PEDERSEN or COMMIT\_HASH\_PEDERSEN schemes. 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 | |
| Let (*a*,*e,z*) denote the prover1, verifier challenge and prover2 messages of the Σ protocol π | |
| 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. 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***. The best choices for the commitment scheme are the COMMIT\_PEDERSEN and COMMIT\_HASH\_PEDERSEN schemes.

## 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*** and possible context information ***cont*** * 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 | |
| Let (*a*,*e,z*) denote the prover1, verifier challenge and prover2 messages of the Σ protocol π | |
| Step 1 (P): | COMPUTE the first message ***a***in **π**, using **(*x,w*)** as input  COMPUTE ***e*=H(*x*,*a,cont*)**  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,cont*)**, 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,cont*)**  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,cont*)**, 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. ***T*** should be 40 at least (this should be a default but certainly not enforced in code). |
| 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 | |
| Let (*a*,*e,z*) denote the prover1, verifier challenge and prover2 messages of the Σ protocol π | |
| 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 |

## 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

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

|  |  |
| --- | --- |
| Protocol Name: | UC-Secure commitment from ROM |
| Protocol Reference: | COMMIT\_UC\_HASH\_ROM |
| Protocol Type: | UC secure commitment |
| Protocol Description: | Universally-composable commitment scheme based on hash functions that is secure in the random oracle model. The scheme is secure in the presence of *adaptive adversaries*. |
| References: | 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 |

|  |  |
| --- | --- |
| COMMIT\_UC\_HASH Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | An agreed-upon hash function **H**, a security parameter ***n***, session and subsession identifiers ***sid*** and ***ssid***, unique identifiers for the committer and receiver ***pidC*** and ***pidR***, respectively.  Any non-interactive string commitment protocol (e.g., COMMIT\_BASIC\_HASH). |
| 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\_ UC\_HASH Protocol Specification | |
| Commit phase | |
| Step 1 (C): | SAMPLE two random values ***r1,r2* ∈ {0*,* 1}*n***  COMPUTE ***c1* = *COMMIT(H(sid,ssid,pidC,pidR,x,r1),r2)***  COMPUTE ***c2* = *COMMIT(H(sid,r2))***  STORE ***r1,r2***  SEND ***c=(sid,ssid,c1,c2)*** to R |
| Step 2 (R): | WAIT for a value ***c=(sid,ssid,c1,c2)***  STORE ***c*** (ABORT if ***sid,ssid*** are incorrect) |
| Decommit phase | |
| Step 1 (C): | SEND **(*x,r1,r2*)** to R |
| Step 2 (R): | IF **NOT**   * ***c1* = *COMMIT(H(sid,ssid,pidC,pidR,x,r1),r2)*** AND * ***c2* = *COMMIT(H(sid,r2))*** AND * ***x* ∈{0*,* 1}*t***   OUTPUT REJ  ELSE OUTPUT ACC and value ***x*** |

|  |  |
| --- | --- |
| COMMIT\_ UC\_HASH Committer (C) Specification | |
| Commit phase | |
| Step 1: | SAMPLE two random values ***r1,r2* ∈ {0*,* 1}*n***  COMPUTE ***c1* = *COMMIT(H(sid,ssid,pidC,pidR,x,r1),r2)***  COMPUTE ***c2* = *COMMIT(H(sid,r2))***  STORE ***r1,r2***  SEND ***c=(c1,c2)*** to R |
| Decommit phase | |
| Step 1: | SEND **(*x,r1,r2*)** to R |
| Step 2: | OUTPUT nothing |

|  |  |
| --- | --- |
| COMMIT\_ UC\_HASH Receiver (R) Specification | |
| Commit phase | |
| Step 1: | WAIT for a value ***c=(sid,ssid,c1,c2)***  STORE ***c*** (ABORT if ***sid,ssid*** are incorrect) |
| Decommit phase | |
| Step 1: | WAIT for **(*x,r1,r2*)** from C |
| Step 4: | IF **NOT**   * ***c1* = *COMMIT(H(sid,ssid,pidC,pidR,x,r1),r2)*** AND * ***c2* = *COMMIT(H(sid,r2))*** AND * ***x* ∈{0*,* 1}*t***   OUTPUT REJ  ELSE  OUTPUT ACC and value ***x*** |

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

|  |  |
| --- | --- |
| Protocol Name: | UC-Secure commitment from DDH |
| Protocol Reference: | COMMIT\_UC\_DDH |
| Protocol Type: | UC secure commitment |
| Protocol Description: | Universally-composable commitment scheme based on the DDH assumption. The scheme is secure in the presence of *static adversaries.* |
| References: | Yehuda Lindell: Highly-Efficient Universally Composable Commitments based on the DDH Assumption. EUROCRYPT 2011: 446-466 |

|  |  |
| --- | --- |
| COMMIT\_UC\_DDH Protocol Parameters | |
| Parties’ Identities: | Committer (C) and Receiver(R) |
| Common parameters: | A DLOG group ***(G,g,q)***, session and subsession identifiers ***sid*** and ***ssid***, unique identifiers for the committer and receiver ***pidC*** and ***pidR***, respectively.  A **common reference string** with random group elements ***(g1,g2,c,d,h,h1,h2)***  A statistical security parameter ***t’*** (for Sigma protocol security; error is **2-*t’***) |
| 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\_ UC\_DDH Protocol Specification | |
| Commit phase | |
| Step 1 (C): | MAP the string ***(x,sid,ssid,pidC,pidR)*** to the group ***G***; call the result ***m***  SAMPLE a random value ***r* ∈ *Zq*** and encrypt m using Cramer-Shoup with public key ***(g1,g2,c,d,h)***. Call the result ***(u1,u2,e,v)***.  SEND ***c=(sid,ssid,c1,c2)*** to R |
| Step 2 (R): | WAIT for a value ***c***  STORE ***c*** (ABORT if ***sid,ssid*** inside ***c*** are incorrect) |
| Decommit phase | |
| Step 1 (C): | SEND ***x*** to R |
| Step 2 (R): | MAP the string ***(x,sid,ssid,pidC,pidR)*** to the group ***G***; call the result ***m***  SAMPLE a random ***e∈{0,1}t’***and MAP ***e*** to an element ***e’∈G***  SAMPLE random ***R,S∈Zq***, and COMPUTE ***c’=(g1R⋅g2S, h1R⋅h2S⋅e’)***.  SEND ***c’*** to C |
| Step 3 (BOTH): | Run SIGMA\_ENCRYPTED\_VALUE\_CRAMERSHOUP with C as prover and R as verifier. R uses the challenge ***e*** chosen in the previous step. When R sends ***e*** to C, it also sends ***R,S*** and C verifies that ***c’=(g1R⋅g2S, h1R⋅h2S⋅e’)***. |
| Step 4 (R): | IF **NOT**   * ***c1 ,c2 ∈ G*** * ***x* ∈{0*,* 1}*t*** * Sigma protocol output of R was ACC   OUTPUT REJ  ELSE OUTPUT ACC and value ***x*** |

|  |  |
| --- | --- |
| COMMIT\_ UC\_DDH Committer (C) Specification | |
| Commit phase | |
| Step 1: | MAP the string ***(x,sid,ssid,pidC,pidR)*** to the group ***G***; call the result ***m***  SAMPLE a random value ***r* ∈ *Zq*** and encrypt m using Cramer-Shoup with public key ***(g1,g2,c,d,h)***. Call the result ***(u1,u2,e,v)***.  SEND ***c=(sid,ssid,c1,c2)*** to R |
| Decommit phase | |
| Step 1: | SEND ***x*** to R |
| Step 2: | WAIT for a value ***c’*** |
| Step 3: | RUN the protocol SIGMA\_ENCRYPTED\_VALUE\_CRAMERSHOUP as prover using value **r**.  Upon receiving ***e*** in the execution, verify that also receive ***R,S*** and that ***c’=(g1R⋅g2S, h1R⋅h2S⋅e’)***. If not, then ABORT. |

|  |  |
| --- | --- |
| COMMIT\_ UC\_DDH Receiver (R) Specification | |
| Commit phase | |
| Step 1: | WAIT for a value ***c***  STORE ***c*** (ABORT if ***sid,ssid*** inside ***c*** are incorrect) |
| Decommit phase | |
| Step 1: | WAIT for ***x*** from C |
| Step 2 (R): | MAP the string ***(x,sid,ssid,pidC,pidR)*** to the group ***G***; call the result ***m***  SAMPLE a random ***e∈{0,1}t’***and MAP ***e*** to an element ***e’∈G***  SAMPLE random ***R,S∈Zq***, and COMPUTE ***c’=(g1R⋅g2S, h1R⋅h2S⋅e’)***.  SEND ***c’*** to C |
| Step 3 (BOTH): | Run SIGMA\_ENCRYPTED\_VALUE\_CRAMERSHOUP as verifier. Use the challenge ***e*** chosen in the previous step. When sending the challenge ***e*** to C, also send ***R,S***. |
| Step 4 (R): | IF **NOT**   * ***c1 ,c2 ∈ G*** * ***x* ∈{0*,* 1}*t*** * Sigma protocol output was ACC   OUTPUT REJ  ELSE OUTPUT ACC and value ***x*** |

## 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 Transfer

## Semi-Honest OT (OT\_DDH\_SEMIHONEST)

|  |  |
| --- | --- |
| Protocol Name: | Semi-Honest OT assuming DDH |
| Protocol Reference: | OT\_DDH\_SEMIHONEST |
| Protocol Type: | Oblivious Transfer Protocol |
| Security Level: | Secure for semi-honest adversaries only |
| Protocol Description: | Two-round oblivious transfer based on the DDH assumption that achieves security in the presence of semi-honest adversaries |
| References: | Folklore |

|  |  |
| --- | --- |
| 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 | |
| Step1 (R): | SAMPLErandom values ***α∈Zq*** and ***h∈G***  COMPUTE ***h0,h1***as follows:   1. If ***σ* = 0** then ***h0* = *gα*** and ***h1=h*** 2. If ***σ* = 1** then ***h0=h*** and ***h1* = *gα***   SEND **(*h0,h1)***to S |
| Step 2 (S): | SAMPLE a random value ***r*** **∈ {0*, . . . , q-1*}**  COMPUTE:   * ***u* = *gr*** * **k0 = h0r** * ***v0 = x0* XOR *KDF*(|*x0*|*,k0*)** * **k1 = h1r** * ***v1* = *x1* XOR *KDF*(|*x1*|*,k1*)**   SEND **(*u,v*0*,v*1)** to R  OUTPUT nothing |
| Step 3 (R): | COMPUTE ***kσ* = (*u*)*α***  OUTPUT ***xσ* = *vσ* XOR *KDF*(|*cσ*|,*kσ*)** |

|  |  |
| --- | --- |
| 0T\_DDH\_PRIVATE Sender (S) Specification | |
| Step 1: | WAIT for message **(*h0,h1)***from R |
| Step 2: | SAMPLE a random value ***r*** **∈ {0*, . . . , q-1*}**  COMPUTE:   * ***u* = *gr*** * **k0 = h0r** * ***v0 = x0* XOR *KDF*(|*x0*|*,k0*)** * **k1 = h1r** * ***v1* = *x1* XOR *KDF*(|*x1*|*,k1*)**   SEND **(*u,v*0*,v*1)** to R |
| Step 3: | OUTPUT nothing |

Note: the computation of ***u*** can be carried out before receiving the message from R.

|  |  |
| --- | --- |
| 0T\_DDH\_PRIVATE Receiver (R) Specification | |
| Step 1: | SAMPLErandom values ***α∈Zq*** and ***h∈G***  COMPUTE ***h0,h1***as follows:   1. If ***σ* = 0** then ***h0* = *gα*** and ***h1=h*** 2. If ***σ* = 1** then ***h0=h*** and ***h1* = *gα***   SEND **(*h0,h1)***to S |
| Step 2: | WAIT for the message **(*u, v*0*,v*1)** from S |
| Step 3: | COMPUTE ***kσ* = (*u*)*α***  OUTPUT ***xσ* = *vσ* XOR *KDF*(|*cσ*|,*kσ*)** |

## Semi-Honest OT Extension (OT\_SEMIHONEST\_EXTENSION)

|  |  |
| --- | --- |
| Protocol Name: | Semi-Honest OT Extension |
| Protocol Reference: | OT\_SEMIHONEST\_EXTENSION |
| Protocol Type: | Oblivious Transfer Protocol |
| Security Level: | Secure for semi-honest adversaries only |
| Protocol Description: | This is a protocol that takes any semi-honest OT protocol and correlation robust hash function, and provides multiple OTs at the cost of a small number of actual OT invocations. |
| References: | Y. Ishai, J. Kilian, K. Nissim and E. Petrank. Extending Oblivious Transfers Efficiently. *CRYPTO 2003*, pages 145-161. |

|  |  |
| --- | --- |
| 0T\_SEMIHONEST\_EXTENSION Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Common Parameters: | * As needed for the semihonest OT protocol (denoted OT\_SEMIHONEST) and the hash function (denoted ***H***) * A security parameter ***n*** (**default *n*=128**) * The number ***m*** of OTs being run (***m* > n**) |
| Parties’ Inputs: | * S’s private input: ***m*** pairs ; within each pair the strings are 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 ***σ1,…, σm* ∈ {0*,* 1}** |
| Parties’ Outputs: | * S: nothing * R: |

|  |  |
| --- | --- |
| 0T\_ SEMIHONEST\_EXTENSION Protocol Specification | |
| Step 1 (Both): | S: SAMPLE a random string ***s*∈{0,1}*n*** of length ***n***; denote it ***s1,…,sn***  R: SAMPLE ***n*** random strings ***T1,…,Tn*∈{0,1}*m*** each of length ***m*** |
| Step 2 (Both): | For ***i* = 1 to *n***, RUN OT\_SEMIHONEST where S plays the receiver and R plays the sender, with the following inputs   * R (playing sender): ***(Ti,Ti XOR σ)***, where ***σ=σ1,…, σm*** * S (playing receiver): ***si*** |
| Step 3 (S): | * Denote the output of S in the OT executions by **Q1,…,Qn** * Let **Q** be the matrix with ***m*** rows and ***n*** columns: **[*Q1*|Q2|…|*Qn*]** and denote by ***Q[i]*** the ***i***th row of ***Q*** (of length ***n***) |
| Step 4 (R): | * Let **T** be the matrix with ***m*** rows and ***n*** columns: **[*T1*|T2|…|*Tn*]** and denote by ***T[i]*** the ***i***th row of ***Q*** (of length ***n***) |
| The *i*th Transfer (with inputs and *σi*) | |
| Step 1 (S): | SEND and |
| Step 2 (R): | WAIT for from S  OUTPUT |

|  |  |
| --- | --- |
| 0T\_ SEMIHONEST\_EXTENSION Sender (S) Specification | |
| Step 1: | SAMPLE a random string ***s*∈{0,1}*n*** of length ***n***; denote it ***s1,…,sn*** |
| Step 2: | For ***i* = 1 to *n***, RUN OT\_SEMIHONEST as the receiver with input ***si*** |
| Step 3: | * Denote the output of S in the OT executions by **Q1,…,Qn** * Let **Q** be the matrix with ***m*** rows and ***n*** columns: **[*Q1*|Q2|…|*Qn*]** and denote by ***Q[i]*** the ***i***th row of ***Q*** (of length ***n***) |
| The *i*th Transfer (with inputs and *σi*) | |
| Step 1: | SEND and  OUTPUT nothing |

|  |  |
| --- | --- |
| 0T\_ SEMIHONEST\_EXTENSION Receiver (R) Specification | |
| Step 1: | SAMPLE ***n*** random strings ***T1,…,Tn*∈{0,1}*m*** each of length ***m*** |
| Step 2: | For ***i* = 1 to *n***, RUN OT\_SEMIHONEST as sender, with input ***(Ti,Ti XOR*** σ***)*** |
| Step 3: | * Let **T** be the matrix with ***m*** rows and ***n*** columns: **[*T1*|T2|…|*Tn*]** and denote by ***T[i]*** the ***i***th row of ***Q*** (of length ***n***) |
| The *i*th Transfer (with inputs and *σi*) | |
| Step 2: | WAIT for from S  OUTPUT |

## Private OT (Naor-Pinkas) (OT\_DDH\_PRIVATE)

|  |  |
| --- | --- |
| 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 that achieves privacy |
| 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): | IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT |
| Step 2 (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 3 (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  OUTPUT nothing |
| Step 4 (R): | IF NOT   * ***w*0*, w*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,*kσ*)** |

|  |  |
| --- | --- |
| 0T\_DDH\_PRIVATE Sender (S) Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT  WAIT for message ***a*** from R |
| Step 2: | 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: | OUTPUT nothing |

|  |  |
| --- | --- |
| 0T\_DDH\_PRIVATE Receiver (R) Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT  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: | WAIT for message pairs **(*w*0*, c*0)** and **(*w*1*, c*1)** from S |
| Step 3: | IF NOT   * ***w*0*, w*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,*kσ*)** |

## Oblivious Transfer with One-Sided Simulation (OT\_DDH\_ONESIDEDSIM)

|  |  |
| --- | --- |
| Protocol Name: | Oblivious Transfer with one-sided simulation |
| Protocol Reference: | OT\_DDH\_ONESIDEDSIM |
| Protocol Type: | Oblivious Transfer Protocol |
| Protocol Description: | Oblivious transfer based on the DDH assumption that achieves privacy for the case that the sender is corrupted and simulation in the case that the receiver is corrupted |
| References: | Protocol 7.3 page 185 of Hazay-Lindell |

|  |  |
| --- | --- |
| 0T\_DDH\_ONESIDEDSIM 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\_ONESIDEDSIM Protocol Specification | |
| Step 1 (Both): | IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT |
| Step 2 (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 3 (Both): | DENOTE the tuple ***a***received by Sby **(*x, y, z*0*, z*1)**  Run ZKPOK\_FROM\_SIGMA with Sigma protocol SIGMA\_DLOG with R as the prover and S as the verifier. Use common input ***x*** and private input for the prover ***α***.  If verifier-output is REJ, REPORT ERROR (cheat attempt) and HALT |
| Step 4 (S): | 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  OUTPUT nothing |
| Step 5 (R): | IF NOT   * ***w*0*, w*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,*kσ*)** |

|  |  |
| --- | --- |
| 0T\_DDH\_ ONESIDEDSIM Sender (S) Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT  WAIT for message ***a*** from R |
| Step 2: | DENOTE the tuple ***a***receivedby **(*x, y, z*0*, z*1)**  Run the verifier in ZKPOK\_FROM\_SIGMA with Sigma protocol SIGMA\_DLOG. Use common input ***x***.  If output is REJ, REPORT ERROR (cheat attempt) and HALT |
| Step 3: | 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 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| 0T\_DDH\_ ONESIDEDSIM Receiver (R) Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g***)  REPORT ERROR and HALT  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: | Run the prover in ZKPOK\_FROM\_SIGMA with Sigma protocol SIGMA\_DLOG. Use common input ***x*** and private input ***α***. |
| Step 3: | WAIT for message pairs **(*w*0*, c*0)** and **(*w*1*, c*1)** from S |
| Step 4: | IF NOT   * ***w*0*, w*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  COMPUTE ***kσ* = (*wσ*)*β***  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,*kσ*)** |

## Oblivious Transfer with Full Simulation (OT\_DDH\_FULLSIM)

|  |  |
| --- | --- |
| Protocol Name: | Oblivious Transfer with full simulation |
| Protocol Reference: | OT\_DDH\_FULLSIM |
| Protocol Type: | Oblivious Transfer Protocol |
| Protocol Description: | Oblivious transfer based on the DDH assumption that achieves full simulation |
| References: | Protocol 7.5.1 page 201 of Hazay-Lindell; this is the protocol of [PVW] adapted to the stand-alone setting |

|  |  |
| --- | --- |
| 0T\_DDH\_FULLSIM Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g0***) where (***G,q,g0***) 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*** |

The protocol below uses the function **RAND(w,x,y,z)** defined as follows:

* 1. SAMPLE random values ***s,t*** **∈{0*, . . . , q-1*}**
  2. COMPUTE ***u* = *ws*⋅*yt***
  3. COMPUTE ***v* = *xs*⋅*zt***
  4. OUTPUT **(*u,v*)**

|  |  |
| --- | --- |
| 0T\_DDH\_FULLSIM Protocol Specification | |
| Step 1 (Both): | IF **NOT** VALID\_PARAMS(***G,q,g0***)  REPORT ERROR and HALT |
| Step 2 (R): | SAMPLErandom values ***y,α0,r* ∈{0*, . . . , q-1*}**  SET ***α1*** **= *α0* + 1**  COMPUTE   * 1. ***g*1 = (*g*0)*y***   2. ***h*0 = (*g*0)*α0***   3. ***h*1 = (*g*1)*α1***   4. ***g* = (*g*σ)*r***   5. ***h* = (*h*σ)*r***   SEND **(*g*1*,h*0*,h*1)** and **(*g,h*)** to S |
| Step 3 (Both): | Run ZKPOK\_FROM\_SIGMA with Sigma protocol SIGMA\_DH with R as the prover and S as the verifier. Use common input **(*g0,g1,h0,h1/g1*)** and private input for the prover ***α0***.  If verifier-output is REJ, REPORT ERROR (cheat attempt) and HALT |
| Step 4 (S): | COMPUTE **(*u0,v0*) = RAND(*g0,g,h0,h*)**  COMPUTE **(*u1,v1*) = RAND(*g1,g,h1,h*)**  COMPUTE ***c0*** = ***x0* XOR *KDF*(|*x0*|*,v0*)**  COMPUTE ***c1*** = ***x1* XOR *KDF*(|*x1*|*,v1*)**  SEND **(*u0,c0*)** and **(*u1,c1*)** to R  OUTPUT nothing |
| Step 5 (R): | WAIT for **(*u0,c0*)** and **(*u1,c1*)** from S  IF NOT   * ***u*0*, u*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,(*uσ*)*r*)** |

|  |  |
| --- | --- |
| 0T\_DDH\_ FULLSIM Sender (S) Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g0***)  REPORT ERROR and HALT  WAIT for message from R |
| Step 2: | DENOTE the valuesreceivedby **(*g*1*,h*0*,h*1)** and **(*g,h*)**  Run the verifier in ZKPOK\_FROM\_SIGMA with Sigma protocol SIGMA\_DH. Use common input **(*g0,g1,h0,h1/g1*)**.  If output is REJ, REPORT ERROR (cheat attempt) and HALT |
| Step 3: | COMPUTE **(*u0,v0*) = RAND(*g0,g,h0,h*)**  COMPUTE **(*u1,v1*) = RAND(*g1,g,h1,h*)**  COMPUTE ***c0*** = ***x0* XOR *KDF*(|*x0*|*,v0*)**  COMPUTE ***c1*** = ***x1* XOR *KDF*(|*x1*|*,v1*)**  SEND **(*u0,c0*)** and **(*u1,c1*)** to R |
| Step 4: | OUTPUT nothing |

|  |  |
| --- | --- |
| 0T\_DDH\_ FULLSIM Receiver (R) Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g0***)  REPORT ERROR and HALT  SAMPLErandom values ***y,α0,r* ∈{0*, . . . , q-1*}**  SET ***α1*** **= *α0* + 1**  COMPUTE   * 1. ***g*1 = (*g*0)*y***   2. ***h*0 = (*g*0)*α0***   3. ***h*1 = (*g*1)*α1***   4. ***g* = (*g*σ)*r***   5. ***h* = (*h*σ)*r***   SEND **(*g*1*,h*0*,h*1)** and **(*g,h*)** to S |
| Step 2: | Run the prover in ZKPOK\_FROM\_SIGMA with Sigma protocol SIGMA\_DH. Use common input **(*g0,g1,h0,h1/g1*)** and private input ***α0***. |
| Step 3: | WAIT for messages **(*u0,c0*)** and **(*u1,c1*)** from S |
| Step 4: | IF NOT   * ***u*0*, u*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,(*uσ*)*r*)** |

## Oblivious Transfer with Full Simulation – ROM (OT\_DDH\_FULLSIM\_ROM)

The protocol below uses the function **RAND(w,x,y,z)** defined for OT\_DDH\_FULLSIM.

|  |  |
| --- | --- |
| Protocol Name: | Oblivious Transfer with full simulation |
| Protocol Reference: | OT\_DDH\_FULLSIM\_ROM |
| Protocol Type: | Oblivious Transfer Protocol |
| Protocol Description: | A two-round oblivious transfer based on the DDH assumption that achieves full simulation in the random oracle model |
| References: | Protocol 7.5.1 page 201 of Hazay-Lindell; this is the protocol of [PVW] adapted to the stand-alone setting and using a Fiat-Shamir proof instead of interactive zero-knowledge. |

|  |  |
| --- | --- |
| 0T\_DDH\_FULLSIM\_ROM Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g0***) where (***G,q,g0***) 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\_FULLSIM\_ROM Protocol Specification | |
| Step 1 (Both): | IF **NOT** VALID\_PARAMS(***G,q,g0***)  REPORT ERROR and HALT |
| Step 2 (R): | SAMPLErandom values ***y,α0,r* ∈{0*, . . . , q-1*}**  SET ***α1*** **= *α0* + 1**  COMPUTE   * 1. ***g*1 = (*g*0)*y***   2. ***h*0 = (*g*0)*α0***   3. ***h*1 = (*g*1)*α1***   4. ***g* = (*g*σ)*r***   5. ***h* = (*h*σ)*r***   Run ZKPOK\_FS\_SIGMA with Sigma protocol SIGMA\_DH using common input **(*g0,g1,h0,h1/g1*)** and private input ***α0***. Let ***tP*** denote the resulting proof transcript.  SEND **(*g*1*,h*0*,h*1),** **(*g,h*)** and ***tP*** to S |
| Step 3 (S): | Verify ***tP*** using common input **(*g0,g1,h0,h1/g1*)**.  If verifier-output is REJ, REPORT ERROR (cheat attempt) and HALT  COMPUTE **(*u0,v0*) = RAND(*g0,g,h0,h*)**  COMPUTE **(*u1,v1*) = RAND(*g1,g,h1,h*)**  COMPUTE ***c0*** = ***x0* XOR *KDF*(|*x0*|*,v0*)**  COMPUTE ***c1*** = ***x1* XOR *KDF*(|*x1*|*,v1*)**  SEND **(*u0,c0*)** and **(*u1,c1*)** to R  OUTPUT nothing |
| Step 4 (R): | WAIT for **(*u0,c0*)** and **(*u1,c1*)** from S  IF NOT   * ***u*0*, u*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,(*uσ*)*r*)** |

|  |  |
| --- | --- |
| 0T\_DDH\_ FULLSIM\_ROM Sender (S) Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g0***)  REPORT ERROR and HALT  WAIT for message from R |
| Step 2: | DENOTE the valuesreceivedby **(*g*1*,h*0*,h*1),** **(*g,h*)** and ***tP*** .  Verify ***tP*** using common input **(*g0,g1,h0,h1/g1*)**.  If output is REJ, REPORT ERROR (cheat attempt) and HALT  COMPUTE **(*u0,v0*) = RAND(*g0,g,h0,h*)**  COMPUTE **(*u1,v1*) = RAND(*g1,g,h1,h*)**  COMPUTE ***c0*** = ***x0* XOR *KDF*(|*x0*|*,v0*)**  COMPUTE ***c1*** = ***x1* XOR *KDF*(|*x1*|*,v1*)**  SEND **(*u0,c0*)** and **(*u1,c1*)** to R |
| Step 3: | OUTPUT nothing |

|  |  |
| --- | --- |
| 0T\_DDH\_ FULLSIM\_ROM Receiver (R) Specification | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g0***)  REPORT ERROR and HALT  SAMPLErandom values ***y,α0,r* ∈{0*, . . . , q-1*}**  SET ***α1*** **= *α0* + 1**  COMPUTE   * 1. ***g*1 = (*g*0)*y***   2. ***h*0 = (*g*0)*α0***   3. ***h*1 = (*g*1)*α1***   4. ***g* = (*g*σ)*r***   5. ***h* = (*h*σ)*r***   Run ZKPOK\_FS\_SIGMA with Sigma protocol SIGMA\_DH using common input **(*g0,g1,h0,h1/g1*)** and private input ***α0***. Let ***tP*** denote the resulting proof transcript.  SEND **(*g*1*,h*0*,h*1),** **(*g,h*)** and ***tP*** to S |
| Step 2: | WAIT for messages **(*u0,c0*)** and **(*u1,c1*)** from S |
| Step 3: | IF NOT   * ***u*0*, u*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,(*uσ*)*r*)** |

## Oblivious Transfer with UC Security – DDH (OT\_DDH\_UC\_PVW)

The protocol below uses the function **RAND(w,x,y,z)** defined for OT\_DDH\_FULLSIM. Note that we do not check the discrete log parameters in this protocol because they are part of the common reference string and thus are assumed to be reliably chosen.

|  |  |
| --- | --- |
| Protocol Name: | Oblivious Transfer with UC Security |
| Protocol Reference: | OT\_DDH\_UC\_PVW |
| Protocol Type: | Oblivious Transfer Protocol |
| Protocol Description: | A two-round oblivious transfer based on the DDH assumption that achieves UC security in the common reference string model. |
| References: | This is the protocol of Peikert, Vaikuntanathan and Waters (CRYPTO 2008) for achieving UC-secure OT. |

|  |  |
| --- | --- |
| 0T\_DDH\_UC\_PVW Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: a **common reference string** composed of a DLOG description (***G,q,g0***) and **(g0,g1,h0,h1)** which is a randomly chosen non-DDH tuple. * 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\_UC\_PVW Protocol Specification | |
| Step 1 (R): | SAMPLE arandom value ***r* ∈{0*, . . . , q-1*}**  COMPUTE ***g* = (*g***σ**)*r*** and ***h* = (*h***σ**)*r***  SEND **(*g,h*)** to S |
| Step 2 (S): | COMPUTE **(*u0,v0*) = RAND(*g0,g,h0,h*)**  COMPUTE **(*u1,v1*) = RAND(*g1,g,h1,h*)**  COMPUTE ***c0*** = ***x0* XOR *KDF*(|*x0*|*,v0*)**  COMPUTE ***c1*** = ***x1* XOR *KDF*(|*x1*|*,v1*)**  SEND **(*u0,c0*)** and **(*u1,c1*)** to R  OUTPUT nothing |
| Step 3 (R): | WAIT for **(*u0,c0*)** and **(*u1,c1*)** from S  IF NOT   * ***u*0*, u*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,(*uσ*)*r*)** |

|  |  |
| --- | --- |
| 0T\_DDH\_ UC\_PVW Sender (S) Specification | |
| Step 1: | WAIT for message **(*g,h*)** from R |
| Step 2: | COMPUTE **(*u0,v0*) = RAND(*g0,g,h0,h*)**  COMPUTE **(*u1,v1*) = RAND(*g1,g,h1,h*)**  COMPUTE ***c0*** = ***x0* XOR *KDF*(|*x0*|*,v0*)**  COMPUTE ***c1*** = ***x1* XOR *KDF*(|*x1*|*,v1*)**  SEND **(*u0,c0*)** and **(*u1,c1*)** to R  OUTPUT nothing |
| Step 3: | OUTPUT nothing |

|  |  |
| --- | --- |
| 0T\_DDH\_ UC\_PVW Receiver (R) Specification | |
| Step 1: | SAMPLE arandom value ***r* ∈{0*, . . . , q-1*}**  COMPUTE ***g* = (*g***σ**)*r*** and ***h* = (*h***σ**)*r***  SEND **(*g,h*)** to S |
| Step 2: | WAIT for messages **(*u0,c0*)** and **(*u1,c1*)** from S |
| Step 3: | IF NOT   * ***u*0*, u*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,(*uσ*)*r*)** |

# Batch Oblivious Transfer

## Batch OT – Full Simulation from DDH (OT\_DDH\_FULLSIM\_BATCH)

The protocol below uses the function **RAND(w,x,y,z)** defined for OT\_DDH\_FULLSIM.

|  |  |
| --- | --- |
| Protocol Name: | Oblivious Transfer with full simulation |
| Protocol Reference: | OT\_DDH\_FULLSIM\_BATCH |
| Protocol Type: | Batch Oblivious Transfer Protocol |
| Protocol Description: | Batch oblivious transfer based on the DDH assumption that achieves full simulation. In batch oblivious transfer, the parties run an initialization phase and then can carry out concrete OTs later whenever they have new inputs and wish to carry out an OT. |
| References: | Protocol 7.5.1 page 201 of Hazay-Lindell; this is the protocol of [PVW] adapted to the stand-alone setting |

|  |  |
| --- | --- |
| 0T\_DDH\_FULLSIM\_BATCH Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Parties’ Inputs: | * Common input: (***G,q,g0***) where (***G,q,g0***) is a DLOG description * S’s private inputs: a series of inputs ***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 inputs: a series of bits ***σ* ∈ {0*,* 1}** |
| Parties’ Outputs: | * S: nothing * R:  ***xσ*** |

|  |  |
| --- | --- |
| 0T\_DDH\_FULLSIM\_BATCH Protocol Specification | |
| Initialization Phase | |
| Step 1 (Both): | IF **NOT** VALID\_PARAMS(***G,q,g0***)  REPORT ERROR and HALT |
| Step 2 (R): | SAMPLErandom values ***y,α0* ∈{0*, . . . , q-1*}**  SET ***α1*** **= *α0* + 1**  COMPUTE   * 1. ***g*1 = (*g*0)*y***   2. ***h*0 = (*g*0)*α0***   3. ***h*1 = (*g*1)*α1***   SEND **(*g*1*,h*0*,h*1)** to S |
| Step 3 (Both): | Run ZKPOK\_FROM\_SIGMA with Sigma protocol SIGMA\_DH with R as the prover and S as the verifier. Use common input **(*g0,g1,h0,h1/g1*)** and private input for the prover ***α0***.  If verifier-output is REJ, REPORT ERROR (cheat attempt) and HALT |
| Transfer Phase (with inputs *x0,x1* and *σ*) | |
| Step 1 (R): | SAMPLE arandom value ***r* ∈{0*, . . . , q-1*}**  COMPUTE ***g* = (*g***σ**)*r*** and ***h* = (*h***σ**)*r***  SEND **(*g,h*)** to S |
| Step 2 (S): | COMPUTE **(*u0,v0*) = RAND(*g0,g,h0,h*)**  COMPUTE **(*u1,v1*) = RAND(*g1,g,h1,h*)**  COMPUTE ***c0*** = ***x0* XOR *KDF*(|*x0*|*,v0*)**  COMPUTE ***c1*** = ***x1* XOR *KDF*(|*x1*|*,v1*)**  SEND **(*u0,c0*)** and **(*u1,c1*)** to R  OUTPUT nothing |
| Step 3 (R): | WAIT for **(*u0,c0*)** and **(*u1,c1*)** from S  IF NOT   * ***u*0*, u*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,(*uσ*)*r*)** |

|  |  |
| --- | --- |
| 0T\_DDH\_ FULLSIM\_BATCH Sender (S) Specification | |
| Initialization Phase | |
| Step 1: | IF **NOT** VALID\_PARAMS(***G,q,g0***)  REPORT ERROR and HALT  WAIT for message from R |
| Step 2: | DENOTE the valuesreceivedby **(*g*1*,h*0*,h*1)**  Run the verifier in ZKPOK\_FROM\_SIGMA with Sigma protocol SIGMA\_DH. Use common input **(*g0,g1,h0,h1/g1*)**.  If output is REJ, REPORT ERROR (cheat attempt) and HALT |
| Transfer Phase | |
| Step 1: | WAIT for a message **(*g,h*)** from R |
| Step 2: | COMPUTE **(*u0,v0*) = RAND(*g0,g,h0,h*)**  COMPUTE **(*u1,v1*) = RAND(*g1,g,h1,h*)**  COMPUTE ***c0*** = ***x0* XOR *KDF*(|*x0*|*,v0*)**  COMPUTE ***c1*** = ***x1* XOR *KDF*(|*x1*|*,v1*)**  SEND **(*u0,c0*)** and **(*u1,c1*)** to R |
| Step 3: | OUTPUT nothing |

|  |  |
| --- | --- |
| 0T\_DDH\_ FULLSIM\_BATCH Receiver (R) Specification | |
| Initialization Phase | |
| Step 1: | SAMPLErandom values ***y,α0* ∈{0*, . . . , q-1*}**  SET ***α1*** **= *α0* + 1**  COMPUTE   * 1. ***g*1 = (*g*0)*y***   2. ***h*0 = (*g*0)*α0***   3. ***h*1 = (*g*1)*α1***   SEND **(*g*1*,h*0*,h*1)** to S |
| Step 2: | Run the prover in ZKPOK\_FROM\_SIGMA with Sigma protocol SIGMA\_DH. Use common input **(*g0,g1,h0,h1/g1*)** and private input ***α0***. |
| Transfer Phase | |
| Step 1: | SAMPLE arandom value ***r* ∈{0*, . . . , q-1*}**  COMPUTE ***g* = (*g***σ**)*r*** and ***h* = (*h***σ**)*r***  SEND **(*g,h*)** to S |
| Step 2: | WAIT for messages **(*u0,c0*)** and **(*u1,c1*)** from S |
| Step 3: | IF NOT   * ***u*0*, u*1**∈ ***G***, AND * ***c*0*, c*1** are binary strings of the same length   REPORT ERROR  OUTPUT ***xσ* = *cσ* XOR *KDF*(|*cσ*|,(*uσ*)*r*)** |

NOTE: This is the same as OT\_DDH\_FULLSIM. The only difference is that the first steps are separated here into a distinct initialization phase. Thus, this should be implemented and the other derived by combining the initialization phase and using it only once.

## Semi-Honest Batch OT Extension (OT\_SEMIHONEST\_BATCH\_EXTENSION)

This protocol differs from OT\_SEMIHONEST\_EXTENSION in that the receiver does not need to know its OT inputs in the first stage where the actual OT executions are run. (Note that in OT\_SEMIHONEST\_EXTENSION the sender already does not use its input in this first stage; however, the receiver does.)

|  |  |
| --- | --- |
| Protocol Name: | Semi-Honest OT Extension |
| Protocol Reference: | OT\_SEMIHONEST\_EXTENSION |
| Protocol Type: | Batch Oblivious Transfer Protocol |
| Security Level: | Secure for semi-honest adversaries only |
| Protocol Description: | This is a protocol that takes any semi-honest OT protocol and correlation robust hash function, and provides multiple OTs at the cost of a small number of actual OT invocations. The actual OTs can be run before the parties know their actual inputs. |
| References: | Y. Ishai, J. Kilian, K. Nissim and E. Petrank. Extending Oblivious Transfers Efficiently. *CRYPTO 2003*, pages 145-161. |

|  |  |
| --- | --- |
| 0T\_SEMIHONEST\_EXTENSION Protocol Parameters | |
| Parties’ Identities: | Sender (S) and Receiver (R) |
| Common Parameters: | * As needed for the semihonest OT protocol (denoted OT\_SEMIHONEST) and the hash function (denoted ***H***) * A security parameter ***n*** (**default *n*=128**) * The number ***m*** of OTs being run (***m* > n**) |
| Parties’ Inputs: | * S’s private input: ***m*** pairs ; within each pair the strings are 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 ***σ1,…, σm* ∈ {0*,* 1}** |
| Parties’ Outputs: | * S: nothing * R: |

|  |  |
| --- | --- |
| 0T\_ SEMIHONEST\_EXTENSION Protocol Specification | |
| Initialization Phase | |
| Step 1 (Both): | S: SAMPLE a random string ***s*∈{0,1}*n*** of length ***n***; denote it ***s1,…,sn***  R: SAMPLE ***n*** random strings ***T1,…,Tn*∈{0,1}*m*** each of length ***m*** and SAMPLE a random choice string ***τ=τ1,…, τm*** |
| Step 2 (Both): | For ***i* = 1 to *n***, RUN OT\_SEMIHONEST where S plays the receiver and R plays the sender, with the following inputs   * R (playing sender): ***(Ti,Ti XOR τ)*** * S (playing receiver): ***si*** |
| Step 3 (S): | * Denote the output of S in the OT executions by **Q1,…,Qn** * Let **Q** be the matrix with ***m*** rows and ***n*** columns: **[*Q1*|Q2|…|*Qn*]** and denote by ***Q[i]*** the ***i***th row of ***Q*** (of length ***n***) |
| Step 4 (R): | Let **T** be the matrix with ***m*** rows and ***n*** columns: **[*T1*|T2|…|*Tn*]** and denote by ***T[i]*** the ***i***th row of ***Q*** (of length ***n***) |
| Transfer Phase *i* (with inputs and *σi*) | |
| Step 1 (R): | If ***σi = τi*** then send ***b=0***; else send ***b=1*** |
| Step 2 (S): | WAIT to receive a bit ***b***  IF ***b=0*** then SEND and  IF ***b=1*** then SEND and |
| Step 3 (R): | WAIT for from S  OUTPUT |

|  |  |
| --- | --- |
| 0T\_ SEMIHONEST\_EXTENSION Sender (S) Specification | |
| Initialization Phase | |
| Step 1: | SAMPLE a random string ***s*∈{0,1}*n*** of length ***n***; denote it ***s1,…,sn*** |
| Step 2: | For ***i* = 1 to *n***, RUN OT\_SEMIHONEST as the receiver with input ***si*** |
| Step 3: | * Denote the output from the OT executions by **Q1,…,Qn** * Let **Q** be the matrix with ***m*** rows and ***n*** columns: **[*Q1*|Q2|…|*Qn*]** and denote by ***Q[i]*** the ***i***th row of ***Q*** (of length ***n***) |
| Transfer Phase *i* (with inputs and *σi*) | |
| Step 1: | WAIT to receive a bit ***b***  IF ***b=0*** then SEND and  IF ***b=1*** then SEND and  OUTPUT nothing |

|  |  |
| --- | --- |
| 0T\_ SEMIHONEST\_EXTENSION Receiver (R) Specification | |
| Initialization Phase | |
| Step 1: | SAMPLE ***n*** random strings ***T1,…,Tn*∈{0,1}*m*** each of length ***m*** and SAMPLE a random choice string ***τ=τ1,…, τm*** |
| Step 2: | For ***i* = 1 to *n***, RUN OT\_SEMIHONEST as the sender, with input ***(Ti,Ti XOR* τ*)*** |
| Step 3: | Let **T** be the matrix with ***m*** rows and ***n*** columns: **[*T1*|T2|…|*Tn*]** and denote by ***T[i]*** the ***i***th row of ***Q*** (of length ***n***) |
| Transfer Phase *i* (with inputs and *σi*) | |
| Step 1: | If ***σi = τi*** then send ***b=0***; else send ***b=1*** |
| Step 2: | WAIT for from S  OUTPUT |

# Coin Tossing

## Coin-Tossing of a Single Bit (COIN\_TOSSING\_BLUM)

|  |  |
| --- | --- |
| Protocol Name: | Blum single-coin tossing using any commitment scheme |
| Protocol Reference: | COIN\_TOSSING\_BLUM |
| Protocol Type: | Coin tossing Protocol |
| Protocol Description: | A protocol for tossing a single bit; this protocol is fully secure under the stand-alone simulation-based definitions |
| References: | M. Blum. Coin Flipping by Phone. *IEEE COMPCOM*, 1982. |

|  |  |
| --- | --- |
| COIN\_TOSSING\_BLUM Protocol Parameters | |
| Parties’ Identities: | Party P1 and Party P2 |
| Parties’ Inputs: | None |
| Parties’ Outputs: | The same bit ***b*** |

|  |  |
| --- | --- |
| COIN\_TOSSING\_BLUM 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 to reveal ***b1***  IF COMMIT.decommit returns INVALID  REPORT ERROR (cheat attempt) |
| Step 5 (Both) | OUTPUT ***b1***XOR***b2*** |

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

|  |  |
| --- | --- |
| COIN\_TOSSING\_BLUM Party P2 Specification | |
| Step 1: | SAMPLE a random bit ***b2* ∈ {*0,1*}** |
| Step 2: | WAIT for COMMIT.commit on ***b1*** |
| Step 3: | SEND ***b2*** to P1 |
| Step 4: | RUN subprotocol COMMIT.decommit to receive ***b1*** |
| Step 5 | IF COMMIT.decommit returns INVALID  REPORT ERROR (cheat attempt)  ELSE  OUTPUT ***b1***XOR***b2*** |

## Coin-Tossing of a String (COIN\_TOSSING\_STRING)

|  |  |
| --- | --- |
| Protocol Name: | String Coin Tossing |
| Protocol Reference: | COIN\_TOSSING\_STRING |
| Protocol Type: | Coin tossing Protocol |
| Protocol Description: | A protocol for tossing a string; this protocol is fully secure under the stand-alone simulation-based definitions |
| References: | This protocol is based on: Y. Lindell. Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation. *CRYPTO* 2001. |

|  |  |
| --- | --- |
| COIN\_TOSSING\_STRING Protocol Parameters | |
| Parties’ Identities: | Party P1 and Party P2 |
| Common Parameters: | A parameter ***L*** determining the length of the output |
| Parties’ Inputs: | None |
| Parties’ Outputs: | The same ***L***-bit string ***s*** |

The protocol uses any COMMIT protocol with a ZK-protocol for the commitment and a ZK-protocol of the commitment value. Currently this can be instantiated with:

* + 1. COMMIT\_PEDERSEN with ZKPOK\_FROM\_SIGMA applied to SIGMA\_PEDERSEN and with a ZK\_FROM\_SIGMA applied to SIGMA\_COMMITTED\_VALUE\_PEDERSEN.
    2. COMMIT\_ELGAMAL with ZKPOK\_FROM\_SIGMA applied to SIGMA\_ELGAMAL\_COMMIT and with a ZK\_FROM\_SIGMA applied to SIGMA\_COMMITTED\_VALUE\_ELGAMAL

In concrete instantiations, it may be necessary to apply a KDF to the output value. Concretely:

* + - 1. If ELGAMAL commit is used then the strings s1, s2 are actually random group elements and the KDF is then used to derive L-bit strings
      2. If PEDERSEN commit is used, then if ***L*** is less than the length of q, then nothing needs to be applied. Otherwise, ***s1*** and ***s2*** can be chosen randomly between **0** and **q-1** and afterwards a KDF can be used to extend the result to an ***L***-bit string.

|  |  |
| --- | --- |
| COIN\_TOSSING\_STRING Protocol Specification | |
| Step 1 (both): | P1: SAMPLE a random ***L***-bit string ***s1* ∈ {*0,1*}*L***  P2: SAMPLE a random ***L***-bit string ***s2* ∈ {*0,1*}*L*** |
| Step 2 (P1): | RUN subprotocol COMMIT.commit on ***s1*** |
| Step 3 (both): | RUN ZKPOK\_FROM\_SIGMA applied to a SIGMA protocol that P1 knows the committed value ***s1***.  If the verifier output is REJ, then P2 HALTS and REPORTS ERROR. |
| Step 4 (P2): | SEND ***s2*** to P1 |
| Step 5 (P1): | P1 sends ***s1*** to P2 (but does not run decommit) |
| Step 6 (both): | RUN ZK\_FROM\_SIGMA applied to a SIGMA protocol that the committed value was ***s1***.  If the verifier output is REJ, then P2 HALTS and REPORTS ERROR. |
| Step 7 (both): | OUTPUT ***s1***XOR***s2*** |

|  |  |
| --- | --- |
| COIN\_TOSSING\_STRING Party P1 Specification | |
| Step 1: | SAMPLE a random ***L***-bit string ***s1* ∈ {*0,1*}*L*** |
| Step 2: | RUN subprotocol COMMIT.commit on ***s1*** |
| Step 3: | RUN the prover in a ZKPOK\_FROM\_SIGMA applied to a SIGMA protocol that P1 knows the committed value ***s1*** |
| Step 4: | WAIT for an ***L***-bit string ***s2*** from P2 |
| Step 5: | SEND ***b1*** to P2 |
| Step 6: | RUN the prover in ZK\_FROM\_SIGMA applied to a SIGMA protocol that the committed value was ***s1*** |
| Step 7: | OUTPUT ***s1*** XOR***s2*** |

|  |  |
| --- | --- |
| COIN\_TOSSING\_STRING Party P2 Specification | |
| Step 1: | SAMPLE a random ***L***-bit string ***s2* ∈ {*0,1*}*L*** |
| Step 2: | WAIT to receive a COMMIT.commit from P1 |
| Step 3: | RUN the verifier in a ZKPOK\_FROM\_SIGMA applied to a SIGMA protocol that P1 knows the committed value.  If the verifier output is REJ, then HALT and REPORT ERROR. |
| Step 4: | SEND ***s2*** to P1 |
| Step 5: | WAIT for an ***L***-bit string ***s1*** from P1 |
| Step 6: | RUN the verifier in ZK\_FROM\_SIGMA applied to a SIGMA protocol that the committed value was ***s1***.  If the verifier output is REJ, then HALT and REPORT ERROR. |
| Step 7: | OUTPUT ***s1*** XOR***s2*** |

## Semi-Simulatable Coin-Tossing of a String (COIN\_TOSSING\_SEMI)

|  |  |
| --- | --- |
| Protocol Name: | Semi- Simulatable Coin Tossing |
| Protocol Reference: | COIN\_TOSSING\_SEMI |
| Protocol Type: | Coin tossing Protocol |
| Protocol Description: | A protocol for tossing a string; this protocol is fully secure (with simulation) when P1 is corrupted and fulfills a definition of “pseudorandomness” when P2 is corrupted. |
| References: | Implicit in [Goldreich-Kahan] |

|  |  |
| --- | --- |
| COIN\_TOSSING\_SEMI Protocol Parameters | |
| Parties’ Identities: | Party P1 and Party P2 |
| Common Parameters: | A parameter ***L*** determining the length of the output |
| Parties’ Inputs: | None |
| Parties’ Outputs: | The same ***L***-bit string ***s*** |

This protocol uses any perfectly-hiding commitment scheme (e.g., COMMIT\_PEDERSEN, COMMIT\_HASH\_PEDERSEN, COMMIT\_HASH) and any perfectly-binding commitment scheme (e.g., COMMIT\_ELGAMAL).

|  |  |
| --- | --- |
| COIN\_TOSSING\_SEMI Protocol Specification | |
| Step 1 (both): | P1: SAMPLE a random ***L***-bit string ***s1* ∈ {*0,1*}*L***  P2: SAMPLE a random ***L***-bit string ***s2* ∈ {*0,1*}*L*** |
| Step 2 (both): | RUN subprotocol COMMIT\_PERFECT\_HIDING.commit on ***s1*** with P1 as the committer |
| Step 3 (both): | RUN subprotocol COMMIT\_PERFECT\_BINDING.commit on ***s2*** with P2 as the committer |
| Step 4 (both): | RUN subprotocol COMMIT\_PERFECT\_HIDING.decommit to reveal ***s1*** |
| Step 5 (both): | RUN subprotocol COMMIT\_PERFECT\_BINDING.decommit to reveal ***s2*** |
| Step 6 (both): | OUTPUT ***s1***XOR***s2*** |

|  |  |
| --- | --- |
| COIN\_TOSSING\_STRING Party P1 Specification | |
| Step 1: | SAMPLE a random ***L***-bit string ***s1* ∈ {*0,1*}*L*** |
| Step 2: | RUN the committer in subprotocol COMMIT\_PERFECT\_HIDING.commit on ***s1*** |
| Step 3: | RUN the receiver in subprotocol COMMIT\_PERFECT\_BINDING.commit |
| Step 4: | RUN the committer in subprotocol COMMIT\_PERFECT\_HIDING.decommit to reveal ***s1*** |
| Step 5: | RUN the receiver in subprotocol COMMIT\_PERFECT\_BINDING.decommit to receive ***s2*** |
| Step 6: | OUTPUT ***s1*** XOR***s2*** |

|  |  |
| --- | --- |
| COIN\_TOSSING\_STRING Party P2 Specification | |
| Step 1: | SAMPLE a random ***L***-bit string ***s2* ∈ {*0,1*}*L*** |
| Step 2: | RUN the receiver in subprotocol COMMIT\_PERFECT\_HIDING.commit |
| Step 3: | RUN the committer in subprotocol COMMIT\_PERFECT\_BINDING.commit on ***s2*** |
| Step 4: | RUN the receiver in subprotocol COMMIT\_PERFECT\_HIDING.decommit to receive ***s1*** |
| Step 5: | RUN the committer in subprotocol COMMIT\_PERFECT\_BINDING.decommit to reveal ***s2*** |
| Step 6: | OUTPUT ***s1*** XOR***s2*** |

# Secure Pseudorandom Function Evaluation

|  |  |
| --- | --- |
| Protocol Name: | Private Pseudorandom Function Evaluation |
| Protocol Reference: | SECURE\_PSEUDORANDOM\_FUNCTION\_EVALUATION |
| Protocol Type: | Pseudorandom Function Evaluation Protocol |
| Protocol Description: | This is a secure protocol for computing the Naor-Reingold PRF where one party holds the secret key and the other holds the input. The security level of the protocol is derived directly from the OT used. That is, if the OT is private/one-sided simulatable/fully simulatable/UC then the protocol is the same. |
| References: | Protocol 7.6.3 page 206 of Hazay-Lindell |

|  |  |
| --- | --- |
| Protocol Parameters | |
| Parties’ Identities: | Party P1 and Party P2 |
| Parties’ Inputs: | * Common input: (***G,q,g***) where (***G,q, g*)** is a DLOG description and a parameter ***m*** determining the input length * P1 ’s private input: ***k = (ga0 , a1, . . . , am)*** where   ***a0, a1,…,am ∈ Zq*** are random   * P2's private input: ***x = x1,…, xm*** of length *m* |
| Parties’ Outputs: | * P1: nothing * P2**:** |

|  |  |
| --- | --- |
| Protocol Specification | |
| Step 1 (Both): | BOTH: If NOT VALID\_PARAMS(***G,q,g***) then REPORT ERROR  P1: SAMPLE random values ***r1,…, rm ∈ Zq*** |
| Step 2 (Both): | RUN ***m*** OT executions for any OT (for efficiency this should be a BATCH\_OT). The ***i***th execution for ***i=1,…,m*** is run as follows:  P1 runs the sender (S) with input pair ***(ri, ri⋅ai)*** (with multiplication in ***Z\*q***)  P2 runs the Receiver (R) input bit ***xi*** |
| Step 3 (Both): | IF the output of any of the oblivious transfers is ⊥  REPORT ERROR |
| Step 4 (P2): | LET ***y1x1,…, ymxm***be the outputs of the OT executions.  If there exists an***i*** such that ***yixi* *∉ Z\*q*** thenSET ***yixi* = 1** |
| Step 5 (P1): | COMPUTE  SEND to P2 |
| Step 6 (Both): | P1: OUTPUT nothing  P2:  IF is not of order ***q*** (for a prime order group, this is equivalent to saying that and ), then REPORT ERROR  COMPUTE  OUTPUT |

|  |  |
| --- | --- |
| Party P1 Specification | |
| Step 1: | If NOT VALID\_PARAMS(***G,q,g***) then REPORT ERROR  P1: SAMPLE random values ***r1,…, rm ∈ Zq*** |
| Step 2: | RUN ***m*** OT executions for any OT (for efficiency this should be a BATCH\_OT). The ***i***th execution for ***i=1,…,m*** is run playing the receiver (R) input bit ***xi*** |
| Step 3: | IF the output of any of the oblivious transfers is ⊥  REPORT ERROR |
| Step 4: | COMPUTE  SEND to P2 |
| Step 5: | OUTPUT nothing |

|  |  |
| --- | --- |
| Party P2 Specification | |
| Step 1: | If NOT VALID\_PARAMS(***G,q,g***) then REPORT ERROR |
| Step 2: | RUN ***m*** OT executions for any OT (for efficiency this should be a BATCH\_OT). The ***i***th execution for ***i=1,…,m*** is run playing the sender (S) with input pair ***(ri, ri⋅ai)*** (with multiplication in ***Z\*q***) |
| Step 3: | IF the output of any of the oblivious transfers is ⊥  REPORT ERROR |
| Step 4: | LET ***y1x1,…, ymxm***be the outputs of the OT executions.  If there exists an***i*** such that ***yixi* *∉ Z\*q*** thenSET ***yixi* = 1** |
| Step 5: | WAIT for message from P1  IF is not of order ***q*** (for a prime order group, this is equivalent to saying that and ), then REPORT ERROR |
| Step 6: | COMPUTE  OUTPUT |

# 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

Encrypt-then-authenticate using any CPA secure encryption scheme and any secure MAC.

## Adaptive Secure Channel (with Erasures)

Release 2

# Special Encryption Schemes

## Paillier Encryption and Homomorphic Operations

|  |  |
| --- | --- |
| **Protocol Name:** | **Damgard-Jurik key generation** |
| **Protocol Reference:** | DJ\_KEY\_GENERATION |
| **Protocol Type:** | Key generation |
| **Protocol Description:** | Key generation for the Damgard Jurik encryption scheme. |
| **References:** | [DJ00] |

|  |  |
| --- | --- |
| **DJ\_KEY\_GENERATION Protocol Parameters** | |
| **Parties’ Identities:** | Party P1 (this is a non-interactive function) |
| **Common parameters:** | Security parameter ***k*** |
| **Inputs:** | None |
| **Outputs:** | * Public key ***n*** * Private key ***t*** |

|  |  |
| --- | --- |
| **DJ\_KEY\_GENERATION Protocol Specification** | |
| **Step 1:** | CHOOSE an RSA modulus ***n=pq*** of length ***k*** bits  COMPUTE ***t=lcm(p-1,q-1)***, where lcm is the least common multiple (and can be computed as lcm(a,b) = a\*b/gcd(a,b) )  OUTPUT   * Public key ***n*** * Private key ***t*** |

|  |  |
| --- | --- |
| **Protocol Name:** | **Damgard-Jurik encryption** |
| **Protocol Reference:** | DJ\_ENCRYPTION |
| **Protocol Type:** | Encryption |
| **Protocol Description:** | Encrypt using the Damgard Jurik encryption scheme |
| **References:** | [DJ00] |

|  |  |
| --- | --- |
| **DJ\_ ENCRYPTION Protocol Parameters** | |
| **Parties’ Identities:** | Party P1 (this is a non-interactive function) |
| **Parameters:** | Length parameter ***s***  (Comment: We do not have to define ***s*** in advance. Rather, it can be computed from the length of the plaintext ***x***, as  ***s=|x|/n***  rounded up.) |
| **Inputs:** | * Public key ***n***   We use the notation ***N=ns***, and ***N’*** = ***ns+1***.   * Plaintext ***x ∈ ZN*** |
| **Outputs:** | Ciphertext ***c∈ ZN’*** |

|  |  |
| --- | --- |
| **DJ\_ENCRYPTION Protocol Specification** | |
| **Step 1:** | CHOOSE a random ***r*** in ***ZN’\**** (This can be done by choosing a random value between 1 and ***N’-1***, which is with overwhelming probability in ***ZN’\****.)  COMPUTE ***c = (1+n)xrN mod N’***.  OUTPUT ***c*** |

|  |  |
| --- | --- |
| **Protocol Name:** | **Damgard-Jurik decryption** |
| **Protocol Reference:** | DJ\_DECRYPTION |
| **Protocol Type:** | Decryption |
| **Protocol Description:** | Decrypt using the Damgard Jurik encryption scheme |
| **References:** | [DJ00] |

|  |  |
| --- | --- |
| **DJ\_ DECRYPTION Protocol Parameters** | |
| **Parties’ Identities:** | Party P1 (this is a non-interactive function) |
| **Parameters:** | Length parameter ***s***  (Comment: We do not have to define ***s*** in advance. Rather, it can be computed from the length of ***c***, as ***s=(|c|/n)-1***  rounded up.) |
| **Inputs:** | * Public key ***n***   We use the notation ***N=ns***, and ***N’*** = ***ns+1***.   * Private key ***t*** * Ciphertext ***c ∈ ZN’*** |
| **Outputs:** | Plaintext ***x∈ ZN*** |

|  |  |
| --- | --- |
| **DJ\_DECRYPTION Protocol Specification** | |
| **Step 1:** | COMPUTE using the Chinese Remainder Theorem a value ***d***, such that ***d = 1 mod N,*** and ***d=0 mod t.*** (Comment: if we know ***s*** in advance then ***d*** can be precomputed.)  COMPUTE ***cd mod N’.***  COMPUTE x as the discrete logarithm of ***cd*** to the base ***(1+n)*** modulo ***N’***. This is done by the following computation  ***a=cd***  ***x=0***  ***for j = 1 to s do***  ***begin***  ***t1= ((a mod nj+1 ) - 1) / n***  ***t2 = x***  ***for k = 2 to j do***  ***begin***  ***x = x – 1***  ***t2 = t2 \* x mod nj***  ***t1 = (t1 – (t2 \* nk-1) / factorial(k) ) mod nj***  ***end***  ***x = t1***  ***end***  ***OUTPUT*** ***x*** |

|  |  |
| --- | --- |
| **Protocol Name:** | **Damgard-Jurik homomorphic addition** |
| **Protocol Reference:** | DJ\_ADD |
| **Protocol Type:** | homomorphic addition |
| **Protocol Description:** | Homomorphic addition of ciphertexts in the Damgard Jurik encryption scheme |
| **References:** | [DJ00] |

|  |  |
| --- | --- |
| **DJ\_ ADD Protocol Parameters** | |
| **Parties’ Identities:** | Party P1 (this is a non-interactive function) |
| **Parameters:** | Length parameter ***s***  (Comment: We do not have to define ***s*** in advance. Rather, it can be computed from the length of the plaintext ***c***, as  ***s=|c|/n -1***  rounded up.) |
| **Inputs:** | * Public key ***n***   We use the notation ***N=ns***, and ***N’*** = ***ns+1***.   * Ciphertexts ***c1,c2 ∈ ZN’*** |
| **Outputs:** | Ciphertext ***c∈ ZN’*** (such that ***c*** is a random encryption of the sum of the plaintexts of ***c1*** and ***c2*** modulo ***N***). |

|  |  |
| --- | --- |
| **DJ\_ADD Protocol Specification** | |
| **Step 1:** | COMPUTE ***c = c1\*c2 mod N’***  CHOOSE a random ***r*** in ***ZN’\**** (This can be done by choosing a random value between 1 and ***N’-1***, which is with overwhelming probability in ***ZN’\****.)  COMPUTE ***c = c \* rN mod N’***  OUTPUT ***c*** |

|  |  |
| --- | --- |
| **Protocol Name:** | **Damgard-Jurik deterministic homomorphic addition** |
| **Protocol Reference:** | DJ\_DADD |
| **Protocol Type:** | homomorphic addition |
| **Protocol Description:** | Deterministic homomorphic addition of ciphertexts in the Damgard Jurik encryption scheme. Namely, the resulting ciphertext is a deterministic function of the two input ciphertexts. |
| **References:** | [DJ00] |

|  |  |
| --- | --- |
| **DJ\_ DADD Protocol Parameters** | |
| **Parties’ Identities:** | Party P1 (this is a non-interactive function) |
| **Parameters:** | Length parameter ***s***  (Comment: We do not have to define ***s*** in advance. Rather, it can be computed from the length of the plaintext ***c***, as  ***s=|c|/n -1***  rounded up.) |
| **Inputs:** | * Public key ***n***   We use the notation ***N=ns***, and ***N’*** = ***ns+1***.   * Ciphertexts ***c1,c2 ∈ ZN’*** |
| **Outputs:** | Ciphertext ***c∈ ZN’*** (such that ***c*** is an encryption of the sum of the plaintexts of ***c1*** and ***c2*** modulo ***N***). |

|  |  |
| --- | --- |
| **DJ\_DADD Protocol Specification** | |
| **Step 1:** | COMPUTE ***c = c1\*c2 mod N’***  /\* the only difference from DJ\_ADD is that two lines of code were removed here \*/  OUTPUT ***c*** |

|  |  |
| --- | --- |
| **Protocol Name:** | **Damgard-Jurik multiplication by a constant** |
| **Protocol Reference:** | DJ\_MUL |
| **Protocol Type:** | homomorphic multiplication |
| **Protocol Description:** | Homomorphic multiplication by a constant of a ciphertext in the Damgard Jurik encryption scheme |
| **References:** | [DJ00] |

|  |  |
| --- | --- |
| **DJ\_ MUL Protocol Parameters** | |
| **Parties’ Identities:** | Party P1 (this is a non-interactive function) |
| **Parameters:** | Length parameter ***s***  (Comment: We do not have to define ***s*** in advance. Rather, it can be computed from the length of the plaintext ***c***, as  ***s=|c|/n -1***  rounded up.) |
| **Inputs:** | * Public key ***n***   We use the notation ***N=ns***, and ***N’*** = ***ns+1***.   * Ciphertext ***c1 ∈ ZN’*** * ***a ∈ ZN*** |
| **Outputs:** | Ciphertext ***c∈ ZN’*** (such that ***c*** is a random encryption of the ***a*** multiplied by the plaintext of ***c1*** modulo ***N***). |

|  |  |
| --- | --- |
| **DJ\_MUL Protocol Specification** | |
| **Step 1:** | COMPUTE ***c = (c1)a mod N’***  CHOOSE a random ***r*** in ***ZN’\**** (This can be done by choosing a random value between 1 and ***N’-1***, which is with overwhelming probability in ***ZN’\****.)  COMPUTE ***c = c \* rN mod N’***  OUTPUT ***c*** |

|  |  |
| --- | --- |
| **Protocol Name:** | **Damgard-Jurik deterministic multiplication by a constant** |
| **Protocol Reference:** | DJ\_DMUL |
| **Protocol Type:** | Deterministic homomorphic multiplication |
| **Protocol Description:** | Homomorphic multiplication by a constant of a ciphertext in the Damgard Jurik encryption scheme |
| **References:** | [DJ00] |

|  |  |
| --- | --- |
| **DJ\_ DMUL Protocol Parameters** | |
| **Parties’ Identities:** | Party P1 (this is a non-interactive function) |
| **Parameters:** | Length parameter ***s***  (Comment: We do not have to define ***s*** in advance. Rather, it can be computed from the length of the plaintext ***c***, as  ***s=|c|/n -1***  rounded up.) |
| **Inputs:** | * Public key ***n***   We use the notation ***N=ns***, and ***N’*** = ***ns+1***.   * Ciphertext ***c1 ∈ ZN’*** * ***a ∈ ZN*** |
| **Outputs:** | Ciphertext ***c∈ ZN’*** (such that ***c*** is a random encryption of the ***a*** multiplied by the plaintext of ***c1*** modulo ***N***). |

|  |  |
| --- | --- |
| **DJ\_DMUL Protocol Specification** | |
| **Step 1:** | COMPUTE ***c = (c1)a mod N’***  /\* the only difference from DJ\_ADD is that two lines of code were removed here \*/  OUTPUT ***c*** |

## Attribute-Based Encryption (Release 2)

## Identity-Based Encryption (Release 2)

# 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: |  |

|  |  |
| --- | --- |
| GET\_RANDOM 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*** |

|  |  |
| --- | --- |
| GET\_RANDOM 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** |

## HMAC-Based PRF with Varying Input-Output Lengths

|  |  |
| --- | --- |
| Protocol Name: | PRF with Varying Input-Output Length from PRF with Varying Input |
| Protocol Reference: | PRF\_VARY\_INOUT |
| Protocol Type: | Pseudorandom function |
| Protocol Description: | This is a pseudorandom function with varying input/output lengths, based on any PRF with varying input length (e.g., HMAC). 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. |
| References: | None |

|  |  |
| --- | --- |
| PRF\_VARY\_INOUT Protocol Parameters | |
| Parties’ Identities: | Party P1 (this is a non-interactive function) |
| Common parameters: | * A concrete PRF ***F*** with varying input length; let ***L*** be the (fixed) output length of the HMAC function |
| Parties’ Inputs: | * A secret key ***k*** * An input ***x*** * An output length parameter ***outlen*** |
| Parties’ Outputs: | * A string ***y*** of length ***outlen*** |

|  |  |
| --- | --- |
| PRF\_VARY\_INOUT Protocol Specification | |
| Step 1: | Let ***m*** be the smallest integer for which ***L⋅m*** **> *outlen***, where ***L*** is the output length of ***F***.  FOR ***i*** **= 1** to ***m***  COMPUTE ***Yi* = *F*(*k*,(x,*outlen,i*))** [key=k, data=(x,outlen,i)]  OUTPUT the first ***outlen*** bits of ***Y1,…,Ym*** |

## Hash-Based One-Time Signatures

|  |  |
| --- | --- |
| Protocol Name: | Hash-Based One-Time Signatures |
| Protocol Reference: | HASH\_ONE-TIME\_SIG |
| Protocol Type: | One-time signature scheme |
| Protocol Description: | This is the Lamport one-time signature scheme, using a hash function as a one-way function. In addition, in order to be fixed-length, we use the hash-and-sign paradigm. |
| References: | E.g., Katz-Lindell, page 433 |

|  |  |
| --- | --- |
| HASH\_ONE-TIME\_SIG Key Generation | |
| Parties’ Identities: | Party P1 (this is a non-interactive function) |
| Common parameters: | * A collision-resistant hash function H; let ***L*** be the (fixed) output length of H |
| Key generation alg: | * Choose **2*L*** random ***L***-bit strings **x1,0,x1,1,…,xL,0,xL,1** * For ***i*=1,…,*L***, compute **yi,0=H(xi,0)** and **yi,1=H(xi,1)** |
| Output (keys): | * Public-key **pk**: **y1,0,y1,1,…,yL,0,yL,1** * Private-key **sk**: **x1,0,x1,1,…,xL,0,xL,1** |

|  |  |
| --- | --- |
| HASH\_ONE-TIME\_SIG Sign | |
| Parties’ Identities: | Party P1 (this is a non-interactive function) |
| Common parameters: | * A collision-resistant hash function H; let ***L*** be the (fixed) output length of H |
| Parties’ Inputs: | * A private key ***sk =* x1,0,x1,1,…,xL,0,xL,1** * An input message ***m*** (of any length) |
| Signing alg: | * Compute **z=H(m)**; let **z1,…,zL** be the bits of **z** * Output the signature **x1,z1,…,xL,zL** |

|  |  |
| --- | --- |
| HASH\_ONE-TIME\_SIG Verify | |
| Parties’ Identities: | Party P1 (this is a non-interactive function) |
| Common parameters: | * A collision-resistant hash function H; let ***L*** be the (fixed) output length of H |
| Parties’ Inputs: | * A public key ***pk =* y1,0,y1,1,…,yL,0,yL,1** * An input message ***m*** (of any length) * A signature **x1,z1,…,xL,zL** |
| Signing alg: | * Compute **z=H(m)**; let **z1,…,zL** be the bits of **z** * For ***i*=1,…,*L***, verify that **yi,zi=H(xi,zi)** * Output **ACCEPT** if all are equal; else output **REJECT** |

## PRF-Based PRP with Varying Input-Output Length

|  |  |
| --- | --- |
| Protocol Name: | PRF-Based PRP with Varying Input-Output Length |
| Protocol Reference: | PRFBased\_PRP\_VARY\_INOUT |
| Protocol Type: | Pseudorandom permutation |
| Protocol Description: | This is a pseudorandom permutation with varying input/output lengths (but of course input length = output length), based on any PRF with a variable input/output length (as long as input length = output length). We take the interpretation that there is essentially a different random permutation for every input/output length. |
| References: | None |

|  |  |
| --- | --- |
| PRFBased\_PRP\_VARY\_INOUT Protocol Parameters | |
| Parties’ Identities: | Party P1 (this is a non-interactive function) |
| Parties’ Inputs: | * A secret key ***k*** * An input ***x*** of ***even*** length |
| Parties’ Outputs: | * A string ***y*** of length **|*x*|** |

|  |  |
| --- | --- |
| PRFBased\_PRP\_VARY\_INOUT Protocol Specification | |
| Step 1: | Let **|*x*|=*2L*** (i.e., the length of the input is ***2L***)  Let ***L0*** be the first **|*x*|/2** bits of ***x***  Let ***R0*** be the second **|*x*|/2** bits of ***x***  FOR **i = 1 to 4**  SET ***Li* = *Ri-1***  COMPUTE ***Ri* = *Li-1* ⊕** **PRF\_VARY\_INOUT(*k*,(*Ri-1,i*),*L*)**  [key=k, data=(Ri-1,i), outlen = L]  OUTPUT **(*L4,R4*)** |

## Naor-Reingold Pseudorandom Function

Function based on dlog (because anyway need for oblivious prf)

## Universal One-Way Hashing

## Universal Hash Functions

|  |  |
| --- | --- |
| Protocol Name: | Universal hash function |
| Protocol Reference: | Evaluation Hash Function |
| Protocol Type: | ε -AXU family over GF[28] and . These parameters give a value of ε of *t*/*2n* (at most 2-40 but smaller for smaller *t*) |
| Protocol Description: | * **Universal family** : A universal hash function is a mapping from a finite set A with size a to a finite set B with size b. When a random choice of a hash function h is made, then for any two distinct inputs x and x', the probability that these two inputs yield a collision equals 1/(the size of the family). * **An ε -almost XOR universal family:** Let ***ε*** be any positive real number. An ε -almost XOR universal family (or ***ε -AXU*** family) H of hash functions from a set ***A*** to a set ***B*** is a family of functions from ***A*** to ***B*** such that for any distinct elements ***x, x' ∈ A*** and for any ***b ∈ B***   **|{*h ∈ H : h(x) XOR h(x') = b*}| *≤ ε /(the size of the family)*** |
| References: | [1] Software Performance of Universal Hash Functions :  <http://www.cosic.esat.kuleuven.be/publications/article-73.ps>  [2] On fast and provably secure message authentication based on universal hashing  <http://www.shoup.net/papers/macs.pdf> |

|  |  |
| --- | --- |
| Universal hash function | |
| Parties’ Identities: | Party P1 (this is a non-interactive function) |
| Common parameters: | * A finite field **GF(264)** and a parameter ***t=32,768*** |
| Parties’ Inputs: | * The message: an input ***m*** of length at most **64t** bits (note that this is 128 megabytes) * The key : a random element ***α ∈ GF(264)*** |
| Parties’ Outputs: | * The hash result of ***M(α)⋅α ∈ GF(264)*** |

|  |  |
| --- | --- |
| Universal hash function | |
| Step 1: | * The field **GF(264)** is represented as ***GF(2)[x]/f(x)***, with ***f(x) = x64 + x4 + x3 + x + 1***. ***f(x)*** is a good 64 degree irreducible polynomial that will be fixed for all computations. * The input ***m*** (of length ***≤ 64t*** bits) is viewed as a polynomial ***M(x)*** of degree *<* ***t*** over ***GF(264)*** as follows.   Every 64 bits of ***m*** are viewed as an element in ***GF(264)***. Every such element is a coefficient of the polynomial ***M(x)***. The total of at most ***t*** coefficients gives a polynomial of degree at most **t*.***   * COMPUTE ***M(α)⋅α ∈ GF(264)*** [key=***α***, data= ***M(x)***] as follows   Evaluate the polynomial ***M(x)*** on ***α*** to get ***M(α)∈ GF(264)***  Multiply by ***α ∈ GF(264)*** to get ***M(α)⋅α ∈ GF(264)*** |

## 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*,*1*))** [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-1)]  OUTPUT the first ***L*** bits of ***K*(*1*),…,*K*(*t*)** |